gefördert vom
Bundesministerium
für Forschung
und Technologie

Verbmobil
Verbundvorhaben

Lexikonformalismus LeX4

Gunter Gebhardi
Johannes Heinecke

Humboldt Universität zu Berlin

Technisches Dokument 19
März 1995

März 1995

Gunter Gebhardi
Johannes Heinecke

Humboldt-Universität zu Berlin
Philosophische Fakultät II
Institut für deutsche Sprache und Linguistik
Computerlinguistik
Jägerstr. 10/11
10099 Berlin

Tel.: (030) 20192 - 553
Fax: (030) 20192 - 560
e-mail: {gebhardi|heinecke}@compling.hu-berlin.de

Die vorliegende Arbeit wurde im Rahmen des Verbundvorhabens Verbmobil vom Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie (BMBF) unter dem Förderkennzeichen 01 IV 101 G gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei den Autoren.

Inhaltsverzeichnis

Vorwort

Der lexikalische Repräsentationsformalismus LEXI entstand im Rahmen der Arbeiten in dem vom Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie geförderten Verbundprojekt VERBMOBIL.

Die Intention des lexikalische Repräsentationsformalismus – kurz: Lexikonformalismus – ist das linguistische Wissen für die maschinelle Verarbeitung natürlicher Sprache, welches in einem Lexikon gespeichert ist, in geeigneter Weise zusammenzustellen, zu verwalten und zu warten und schließlich in einer für den Nutzer der Informationen geeigneten Form bereitzustellen. Zwei Betrachtungsweisen des Lexikons sind dabei markant: die Sicht der Lexikographen, derer, die das Lexikon aufbauen, und die Sicht der Lexikonnutzer. Beide Sichten sind meist schwerlich miteinander zur Deckung zu bringen; LEXI soll deshalb als Brücke dienen. Man kann natürlich auch noch unterschiedliche Sichten über die Art *der* Brücke diskutieren, ... LEXI ist *eine* Brücke.

LEXI ist *ein* Werkzeug für den Lexikonerstellenden und -nutzenden. Bei seiner Entwicklung sind wir davon ausgegangen, daß es eine Reihe anderer Werkzeuge gibt, derer sich der LEXI-Nutzer bedient. So ersetzt LEXI keine Standardprogramme wie `sed`, `awk` oder einfache *shell scripte*. LEXI setzt die Verfügbarkeit eines entsprechenden Verarbeitungssystems mit Entwicklungsumgebung für die vom Nutzer extrahierten Daten voraus und erweißt sich auch nicht als effizienter Parser. LEXI zielt auf die Unterstützung des Aufbaus großerer Lexika ab. In diesem Punkt läßt LEXI zwei Felder offen: Datenspeicherung und Datenakquisition. Und genau für diese Aufgaben befinden sich auf LEXI abgestimmte Erweiterungen und Ergänzungen in Entwicklung.

Als geeignetes Beschreibungsmittel dienen in LEXI die aus der Computerlinguistik stammenden **Attribut-Wert-Strukturen**. Eine Typisierung dieser Strukturen bietet eine Reihe von Vorteilen, etwa um linguistische Verallgemeinerungen zu zeigen und damit kompaktere Darstellungen zu gestatten oder um die Konsistenz der Darstellung zu unterstützen oder um eine effizientere Verwaltung der Daten zu ermöglichen, um nur einige zu nennen. Andererseits muß daß Typenkonzept genügend Spielraum bieten um Änderungen und Modifikationen am Lexikon ebenso flexibel vornehmen zu können wie Anpassungen an unterschiedliche Nutzererfordernisse. Wir haben daher ein typenähnliches **Klassenkonzept** LEXI zugrunde gelegt. Schließlich unterstützt LEXI das Konzept unterschiedlicher Sichten auf die Objekte im Lexikon, indem von einem Objekt unterschiedliche **Instanzen** gebildet werden können.

Die vorliegende Sprachbeschreibung stellt die einzelnen Elemente von LEXI in einer für den Nutzer der Sprache geeigneten Form dar. Die Beschreibung spiegelt dabei einen bestimmten Entwicklungsgrad von LEXI wider. Geringfügige Abweichnungen der ausgelieferten Version von der hier beschriebenen sollten sich nicht in der *Sprache* LEXI auswirken. So sind fast alle LEXI- Beschreibungen, die vor Version 1.00 erstellt wurden, auch noch heute lauffähig. Da im Lauf der Zeit bereits einige tausend Lexikoneinträge entstanden sind, werden zukünftig Sprachveränderungen, sollten sie denn notwendig werden, äußerst behutsam erfolgen. Um den Entwicklungsstand und damit einhergehende Veränderungen zu dokumentieren, haben wir die Anmerkungen zu älteren Version von LEXI angefügt.

Version 1.37

Die aktuelle Version von *LeXi4* bringt für den Nutzer keine unmittelbar sichtbaren Veränderungen, mit Ausnahme der überarbeiten Debuggerinteraktion. Es gibt einige interne Veränderungen, die der Portierbarkeit von *LeXi4* dienen.

Aufgrund der nun vorliegenden Erfahrungen soll möglichst rasch der Schritt zur Version 2 des Formalismus vollzogen werden, die dann auch eine Anbindung an ein Datenbanksystem beinhaltet.

Berlin, Februar 1995

Version 1.28

In der vorliegenden Version 1.28 des Lexikonformalismus *LeXi4* sind zwei Änderungen für den Nutzer herausragend: Zum einen ist die Verarbeitungsgeschwindigkeit deutlich erhöht worden und zum anderen ist der Debugger einfacher und effizienter. Eine Änderung, die für den Nutzer einerseits einen Vorteil bietet, ist die Modifikation der Verarbeitungsstrategie zur Bewahrung einfacher disjunktiver Ausdrücke, andererseits mußten wir die Variablenbindung auf die PROLOG–Ebene zurücknehmen, was wir in einer der folgenden Version so rasch wie möglich wieder ändern werden.

Eine weitere gravierende Änderung ist die Portierung von *LeXi4* nach Quintus–PROLOG Die Nutzung von *LeXi4* setzt jetzt das Vorhandensein dieser PROLOG–Implementation voraus. Leider ging mit der Portierung kein Geschwindigkeitszuwachs einher. Mit der Portierung entsprechen wir der Forderung der Projektpartner.

Die vorliegende Version von *LeXi4* diente zum Aufbau der von uns ausgelieferten Version des VERBMOBIL–Lexikons. Dies belegt zum einen die Stabilität dieser Implementation, zum anderen erwieß sich *LeXi4* damit bereits als ein nützliches Werkzeug zum Aufbau eine Lexikons und dabei insbesondere zum Aufspüren und der Beseitigung von Inkonsitenzen.

Berlin, Dezember 1994

Version 1.00

Die vorliegende Version 1.00 des Lexikonformalismus *LeXi4* ist die erste, weitgehend vollständige Version. In ihr sind bereits all die Konzepte realisiert, die wir als Notwendig erachten, um ein Lexikon aufzubauen und zu warten.

Die Implementation dieser Version mußte in kürzerer Zeit erfolgen, als ursprünglich geplant war, einige Systemfunktionen, die uns zur Verfügung gestellt werden sollten, mußten wir selbst quasi reimplementieren und schließlich ist die Programmiersprachbasis nicht die, mit der wir eigentlich arbeiten wollten und sollten, was uns zu einer baldigen Portierung zwingen wird.

Die nun vorliegende Version des Lexikonformalismus *LeXi4* ist in HU–PROLOG implementiert, einer PROLOG–Version die an der Humboldt–Universität zu Berlin entstand

und für die meisten Nutzer ohne Zusatzkosten für die unterschiedlichsten Plattformen verfügbar ist. Für all diese Nutzer ist auch LaTeX sofort nutzbar.

Da alle Funktionen von uns implementiert werden mußten, konnten wir konsequent darauf Einfluß nehmen, LaTeX neutral bezüglich der Vielfalt möglicher Formalismen zu gestalten, die die Datenstrukturen von LaTeX als Eingabe übernehmen. Damit konnten wir ein wenig der *Idee eines theorieneutralen Lexikons* näher kommen. Um dem Nutzer dennoch ein Schnittstelle zu seinem Formalismus zu geben, gibt es das LaTeX Interfacemodul LaTeXTrafo.

Berlin, Oktober 1994

1 Aufbau der Beschreibung

Die Beschreibung beginnt nach einigen allgemeinen Anmerkungen mit der Vorstellung der wichtigsten Ausdrucksmittel von *LEXY*. Diese Ausdrucksmittel gestatten es, die Grundfunktionen von *LEXY* anhand von Beispielen zu skizzieren. Damit ist dann die Basis geschaffen, weitere Operationen einzuführen und ihre Nutzung ebenfalls an Beispielen zu zeigen. Einen umfangreicheren Abschnitt nehmen dann die Templates ein, die die Arbeit mit *LEXY* ganz wesentlich unterstützen. Damit ist die Beschreibung der *LEXY*-Beschreibungsmittel abgeschlossen.

Um die erstellten Beschreibungen auszuwerten, einerseits um damit die Lexikoneinträge entsprechen den in der Beschreibung formulierten Kriterien zu evaluieren und andererseits um nutzerspezifische Lexika zu generieren, sind entsprechende Funktionen des *LEXY*-Interpreters verfügbar. Nach deren Vorstellung werden die Funktionen zur Fehlersuche erläutert.

Abschließend wird als Vorschau auf die in Entwicklung befindliche Version 2 von *LEXY* eine Vorschau auf die Erweiterungen zur Datenspeicherung gegeben.

Wir wollen hier noch auf die von uns ausgelieferte Applikationsbeispiele verweisen, die sich auch als Grundstein für die ersten eigenen Experimente eignen. Diese Applikationsbeispiele liegen als VERBMOBIL-Memos vor. (Gunter Gebhardi, Johannes Heinecke: Substantivflexion in *LEXY* — Ein Applikationsbericht. VERBMOBIL-Memo 62. Humboldt-Unversität zu Berlin 1995; Johannes Heinecke, Gunter Gebhardi: Verbkonjugation im Lexikonformalismus *LEXY*. VERBMOBIL-Memo 63. Humboldt-Universität zu Berlin 1995.)

2 Allgemeine Konventionen

2.1 Beziehung zu PROLOG

LEXY ist ein Programmsystem welches eng mit PROLOG verbunden ist. In einigen Funktionsbereichen bietet *LEXY* keine eigenen Funktionen, sondern greift auf die Funktionen des PROLOG–Systems zurück. Offensichtlich ist dies etwa bei der Interaktion mit dem System.

Diese enge Bindung hat ihre Ursache darin, daß wir das Rad nicht neu erfinden wollten und daher einige *LEXY* ergänzende Prädikate und Funktionen aus dem zugrundeliegenden PROLOG–Systems unmittelbar übernommen haben. Und für den Nutzer hat es den Vorteil, daß er nicht wieder andere Namen für *Dateiöffnen, Dateischließen* und ähnliches zu erlernen hat.

Um mit *LEXY* zu arbeiten, ist es dennoch nicht notwendig, detaillierte PROLOG–Kenntnisse zu besitzen. Wie Namen und Variablen bezeichnet werden, ist für jede Programmiersprache definiert. Wir lehnen uns weitgehend an die PROLOG–Konventionen an.

Aufgrund der engen Verbindung zu Prolog gelten bezüglich der Syntax von *LEXY* die PROLOG–üblichen Vereinbarungen. Auf folgende Aspekte sei hier besonders hingewiesen (zu weiteren Details: Nutzerhandbuch des jeweiligen PROLOG–Systems; für eine allgemeine Einführung in PROLOG wird das quasi Standard–Werk von CLOCKSIN,

W.F. und MELLISH, C.S.: *Programmieren in Prolog*, erschienen im Springer–Verlag Berlin (in mehreren Auflagen) empfohlen):

- Atome beginnen mit einem **kleinen** Buchstaben, andernfalls sind Ausdrücke, die als Atom betrachtet werden, in Quotes (') zu setzen
- natürliche Zahlen werden mit Ausnahme von einigen der eingebauten Funktionen wie Atome behandelt
- Variablen beginnen immer mit einem **Großbuchstaben**; in unmittelbarer Interaktion mit dem zugrundeliegenden PROLOG–System erscheinen sie bei der Ausgabe im allgemeinen als Zahlen mit einem unmittelbar vorangestellten Unterstreichungsstrich
- jeder vollständige Ausdruck muß mit einem **Punkt** beendet werden; das Fehlen des Punktes ist einer der häufigsten Fehler, der aber meist vom System erkannt und gemeldet wird
- Kommentare sind entsprechend PROLOG– Sprachvereinbarung zu verwenden (`/* ... */` oder `%` bis Zeilenende)

2.2 Namensvergabe, Konstanten und Variablen

Die Verwendung von Atomnamen ist beliebig, solange diese den obigen Konventionen genügen. Man sollte dennoch überlegt Namen wählen und möglichst LaTeX–Schlüsselwörtern ebenso vermeiden wie die Mehrfachverwendung eines Namens mit jeweils unterschiedlicher Bedeutung.

Konstanten, Atome und ganze Zahlen benennen spezifische Objekte oder Relationen. Eine Konstante ist global gültig. Konstanten werden durch Gebrauch eingeführt und bedürfen keiner Definition oder Deklaration.

Variablennamen können beliebig sein, solange sie den oben genannten Konventionen entsprechen. Auch Variablennamen werden durch Gebrauch eingeführt. Die Gültigkeit von Variablen ist lokal auf einen vollständigen LaTeX–Ausdruck begrenzt, das heißt, der Gültigkeitsbereich einer Variablen erstreckt sich bis zum . Punkt am Ende des aktuellen Ausdrucks.

Es folgen einige Beispiele (man beachte die Verwendung der zwei Kommentarformen)

(1)
```
a                % eine Konstante
12               /* ebenfalls eine Konstante, hier eine Zahl */
helmut           % eine Konstante
ist_dick         % eine Konstante
'Franz-Joseph'   % eine Konstante
Franz_Joseph     % eine Variable
X                % eine Variable
_                % eine anonyme Variable
```

2.3 Prozedurales Verhalten

Die Funktionsweise der einzelnen Ausdruckselemente von *LEX4* wird anhand von Beispielen gezeigt, wobei die prozedurale Semantik der Operatoren „informal“ angegeben wird. Damit wird die Nutzersicht in den Vordergrund gestellt.

Die Operatoren in *LEX4* beschreiben Relationen über den bezeichneten Objekten. Als Ergebnis einer Operation wird das Objekt bezeichnet, welches sich entsprechend der gegebenen Konstruktionsbeschreibung ergibt.

Als Ausdruck wird eine Variable, die Beschreibung einzelner Objekte oder die Verknüpfung von Objekten mit Hilfe der Operatoren bezeichnet. Ein Ausdruck hat ein Ergebnis durch die Variable, das genannte Objekt oder wenn sich entsprechend der angegebenen Operatoren und Objekte ein neues Objekt konstruieren läßt. Lassen sich eine Menge von Objekten konstruieren, hat der Ausdruck eine Menge von Ergebnissen.

3 Attribut-Wert–Strukturen und die Operatoren : und & sowie \

3.1 Klammern

In *LEX4* können die runden Klammern (und) verwendet werden, um die durch die Operatoren präjudizierte Berechnungsreihenfolge innerhalb von Ausdrücken zu verändern. Die Verwendung der Klammern erfolgt gleich der in der Algebra.

Die Bindungsstärke der Operatoren wird zunächst schrittweise mit den Operatoren eingeführt, bis schließlich ein Gesamtüberblick gegeben wird.

3.2 Attribut–Wert–Strukturen

Der wichtigste Datentyp neben Konstanten und Variablen in *LEX4* ist die *Attribut–Wert–Struktur*. Eine Attribut–Wert–Struktur ist eine Datenstruktur, die aufgebaut ist aus einer nichtleeren Menge von Attributen – durch Atome bezeichnet – und den Attributen zugeordneten Werten – Ergebnissen von Ausdrücken. Es ist nicht zulässig, eine Attribut–Wert–Struktur so zu beschreiben, daß ein und derselbe Wert als Wert eines seiner Attribute verwendet wird. Genügt eine Attribut–Wert–Struktur dieser Eigenschaft, wird sie als *azyklisch* bezeichnet.

Attribut–Wert–Stukturen sind in gewisser Weise den `record`–Strukturen in PASCAL bzw. `struct`–Strukturen in C vergleichbar.

3.3 Der Operator :

Eine Attribut–Wert–Struktur besteht aus Attribut–Wert–Paaren. Ein solches Paar wird dargestellt als

attribut : *wert*

Der : -Operator bindet stärker als alle anderen Operatoren von *LaV4*.

3.4 Der Operator &

Der & -Operator verknüpft Attribut-Wert-Paare, Variablen und Konstanten. Die Operation, die mit diesem Operator verbunden ist, ist die Unifikation. Der allgemeine Gebrauch erfolgt gemäß

ausdruck_1 & *ausdruck_2*

Der & -Operator besitzt eine schwächere Bindung als der : -Operator.

Die Unifikation zweier Objekte führt zu folgenden Ergebnissen:

- eine Variable wird mit jedem Ausdruck unifiziert; die Variable und alle ihre Vorkommen innerhalb des einen Ausdrucks werden an den anderen Wert (den anderen Ausdruck) gebunden (vereinfacht entspricht dies etwa dem Gleichsetzen)
- eine Konstante unifiziert mit einer Variablen oder derselben Konstanten; das Ergebnis ist immer die Konstante, eine Variable wird an die Konstante gebunden
- eine Attribut-Wert-Struktur unifiziert mit einer Variablen (Ergebnis: Attribut-Wert-Struktur, wobei die Variable an die Struktur gebunden ist) oder derselben Struktur (Ergebnis: dieselbe Struktur) oder einer anderen Struktur, wobei für jedes Attribut in der anderen Struktur gelten muß, daß
  - es in der gegebenen Struktur nicht enthalten ist (Ergebnis: das Attribut aus der anderen Struktur wird mit seinen Werten der ersten Struktur hinzugefügt)
  - es in der gegebenen Struktur enthalten ist und die Werte der Attribute aus den beiden Strukturen miteinander unifizierbar sind (Ergebnis: Attribut mit dem Wert des Ergebnisses der Unifikation der Werte aus den einzelnen Strukturen)
    Das Ergebnis der Unifikation zweier Attribut-Wert-Strukturen ist nur genau dann wieder eine Attribut-Wert-Struktur, wenn die resultierende Struktur azyklisch ist.

Eine Attribut-Wert-Struktur ist das Ergebnis eines Ausdrucks, welcher die Verknüpfung von Attribut-Wert-Paaren beschreibt.

Es werden einige Beispiele von Attribut-Wert-Paaren und Strukturen gegeben:

(2)
```
% Attribut-Wert-Paare

a:b              % Attribut a, Wert b
a:b:c:d          % Attribut a hat als Wert eine
                 % Attribut-Wert-Struktur, die Struktur hat
                 %  das Attribut b und als Wert eine
                 % Attribut-Wert-Struktur, diese ...
wetter:schoen    % Attribut wetter, Wert schoen
tag: X           % Attribut tag, Wert X - eine Variable
```

(3)
```
% Attribut-Wert-Strukturen

agr: case: Case & agr: num: Num
agr: (case: Case & num: Num)       % beschreibt dasselbe

syntax: ( verb: (num: (Num & sg) &
                      pers: Pers) &
              subjekt: (num: Num &
                        pers: Pers) )
```

Schließlich seien Beispiele für Ausdrücke gegeben, die keine Attribut–Wert–Strukturen sind:

(4)
```
schmeckt: lecker & schmeckt: eklig
fritz: (gross & klein)
```

3.5 Der Operator \

Der \ -Operator dient der Darstellung von Disjunktionen. Er entspricht dem logischen *oder*. Ein disjunktiv verknüpfter Ausdruck hat die allgemeine Form

ausdruck_1 \ *ausdruck_2*

Das Ergebnis der disjunktiv verknüpften Ausdrücke ist das Ergebnis jedes einzelnen Ausdrucks. Einem disjunktiven Ausdruck können zwei Ergebnisse zugeordnet werden, wenn es für jeden Ausdruck ein Ergebnis gibt.

(5)
```
case: (nom \ gen \ dat \ acc)        % Kasus: Nominativ oder Genitiv ...
num: sg \ num: pl                    % Numerus: Singular oder Plural
das: (ist & quatsch) \ kein: quatsch % ist & fuehrt zu keinem Ergebnis
                                     % Ergebnis insgesamt daher
                                     % kein:quatsch
```

Der \ -Operator besitzt eine schwächere Bindung als der `&` -Operator.

Disjunktive Ausdrücke werden als Bestandteil der Beschreibung einer Attribut-Wert-Struktur bewahrt. Daß heißt, es gibt Ergebnisse in Form von Attribut-Wert-Strukturen, in denen disjunktive Teilausdrücke enthalten sind. Dies wird durch die Verknüpfungsoperationen, in welche der \ -Ausdruck einbezogen wird, bestimmt. Gegenwärtig werden disjunktive Werte von Attributen beim Kopieren (Hinzufügen in der Unifikation) und bei der Unifikation mit Variablen bewahrt.

Anmerkung: Ab Version 2 von *LeX4* erfolgt die Auswertung von disjunktiven Ausdrücken innerhalb von Attribut–Wert–Strukturen noch stärker disjunktionserhaltend.

4 Klassen, Basen und Instanzen

4.1 Berechnungsreihenfolge

4.1.1 Operatorenauswertung

Klassen- und Basisdefinitionen werden grundsätzlich *von links nach rechts*, in Abhängigkeit der *Bindungsstärke* der Operatoren und der verwendeten *Klammern* berechnet.

4.1.2 Die Berechnungsreihenfolge für Klassen, Basen und Instanzen

Die Berechnung von Klassen ist die Grundlage für die Auswertung von Basisdefinitionen. Diese ist wiederum die Grundlage für die Berechnung von abgeleiteten Instanzen. Durch diese Reihenfolge wird die nachfolgende Beschreibung geprägt.

Die Berechnungen sind durch den Nutzer wie in Abschnitt 9 beschrieben aufzurufen.

4.1.3 Mehrfachdefinitonen

Die Definition von Klassen und Basen mit demselben Namen ist möglich. Solche Definitionen werden als disjunkte Formen einer Definition betrachtet.

Die Reihenfolge der Beschreibung von Klassen, Basen und der im Abschnitt 8 beschriebenen Templates ist beliebig.

4.2 Klassen

4.2.1 Das Schlüsselwort `top`

Das Schlüsselwort wird an den unterschiedlichsten Stellen in $\mathcal{LEV}$–Beschreibungen verwendet und hat dennoch immer dieselbe Bedeutung: Es bezeichnet die allgemeinste Klasse. Vereinfacht ausgedrückt ist das die Klasse, die die Eigenschaft hat, keine speziellen Eigenschaften zu haben.

Von der Klasse `top` werden alle anderen Klassen abgeleitet, entweder direkt oder indirekt über andere Klassen. Genauer: Alle Klassen *erben* von `top` Informationen.

`top` dient als anonyme Variable. Das sind – vereinfacht gesagt – solche Variablen, die innerhalb eines vollständigen Ausdrucks nur einmal auftreten.

4.2.2 `class`–Definitionen

Das Schlüsselwort `class` leitet eine Klassendefinition ein. Eine Klassendefinition wird vervollständigt durch:

- den Namen der Klasse, ein Atom
- das Trennzeichen `:<`

- eine Superklassenspezifikation
- das Trennzeichen `>:`
- die Klassenstrukturbeschreibung
- (einen Punkt am Ende)

Eine Klassendefinition hat damit folgende Form:

`class` *name* `:<` *superklassenspezifikation* `>:` *klassenstrukturbeschreibung*`.`

Superklassenspezifikationen sind Ausdrücke, die aus dem Namen einer einzelnen Klasse bestehen oder aus der Verknüpfung von mehreren Namen unter Einbeziehung der Operatoren `&` und `\`. Ein Klassenstrukturbeschreibung ist ein *LeVi*–Ausdruck.

Klassendefinitionen sind vollständige *LeVi*–Ausdrücke.

Klammerausdrücke dürfen nicht das `>:` -Symbol einschließen.

Diese Definition gestattet es nun, einen vollständigen *LeVi*–Ausdruck auch als Beispiel anzugeben:

(6)
```
class num :< top >: (sg \ pl).
class case :< top >: (nom \ gen \ dat \ acc).
class wetter :< top >: temperatur: 24.
```

4.2.3 Vererbung

Eine Klasse *erbt* alle Informationen ihrer Superklasse bzw. Superklassen. Jede Klasse erbt von einer Superklasse. Um dies zu ermöglichen, existiert die Superklasse `top`.

Im Beispiel

(7)
```
class aa :< top >: a:anton.
class bb :< aa >: b:berta.
```

erbt die Klasse `aa` von `top` (nichts). In der Klassendefinition wird zusätzlich das Attribut–Wert–Paar `a:anton` eingeführt. `class aa` hat damit den Inhalt `a:anton`. `class bb` erbt von `class aa` diese Informationen. Zusätzlich wird das Attribut–Wert–Paar `b:berta` hinzugefügt. Der Inhalt von `class bb` ist daher `a:anton & b:berta`.

Wie bereits im Beispiel offensichtlich wurde, werden die ererbten Informationen mit den Informationen der Klassenstrukturbeschreibung durch Unifikation verknüpft.

Die Bindungsstärke des `>:` -Operators ist identisch mit der des `&`. Wie Nutzerfehler zeigen, ist dies insbesondere bei Beschreibungen unter Einbeziehung des `\` -Operators zu beachten.

Es ist nicht zulässig, Klassen zyklisch zu definieren, wie in

(8)
```
class x :< x >: ... .
                % oder auch
```

```
class x :< y >: ... .
class y :< x >: ... .
                    % und so weiter
```

Das heißt jedoch nicht, daß die Verwendung rekursiver Beschreibungen nicht möglich ist.

4.2.4 Klassen als Bestandteil von Attribut–Wert– Strukturen

Jeder Klassenname, der eine Klasse bezeichnet, deren Auswertung zu einem Ergebnis führt, kann an die Stelle einer Konstanten treten. Genauer: Bei der Auswertung eines Ausdrucks, der aus einer Konstanten besteht, wird zuerst davon ausgegangen, daß es sich dabei nicht um eine Konstante, sondern um den Namen einer Klasse handelt.

(9)
```
class num :< top >: (sg \ pl).
class cas :< top >: (nom \ gen \ acc \ dat).
class numerus_c :< top >: numerus:num.
class kasus_c :< top >: kasus: cas.
class agr :< numerus_c & kasus_c >: top.
class numerus_singular :< numerus_c >: numerus: pl.
class numerus_plural :< numerus_c >: numerus: sg.
class numerus_falsch :< numerus_c >: numerus: ganz_viel.
```

Beispiel (9) zeigt eine Klassenhierarchie. In `class numerus_c` wird dem Attribut `numerus` der Wert `num` zugewiesen – eine Klasse. Das Ergebnis der Auswertung der Klassendefinition `numerus_c` ist genau das, was in der Definition steht: `numerus:num`. Das Ergebnis der Auswertung der Klassendefinition `agr` ist `numerus: num & kasus: cas`; `num` wird nicht aufgelöst. `num` wird aber ausgewertet, wenn es mit dem Wert `pl` bzw. `sg` in den Definitionen von `class numerus_singular` bzw. `class numerus_plural` verknüpft wird. Das Ergebnis ist `numerus: pl` bzw. `numerus: sg`. (Man beachte auch die Auflösung der Disjunktion!) Die Auswertung der Klassendefinition `class numerus_falsch` schließlich führt hier zu keinem Ergebnis, da `ganz_viel` weder mit `sg` noch `pl` unifiziert werden kann.

Die Anzahl rekursiver Klassenexpansionen kann theoretisch unendlich sein, etwa in der Definition von Listen. In der Praxis sind jedoch gerade endliche Strukturen von Interesse. Um einerseits rekursive Beschreibungen zu erlauben und andererseits unendliche Berechnungen zu verhindern, insbesondere im Zusammenhang mit den im Abschnitt 4.4 eingeführten Instanzen, ist eine Schranke für die Berechnungstiefe gesetzt. Wird diese Schranke überschritten, erfolgt eine Fehlermeldung. Die Schranke kann durch den Nutzer in der Parameterdatei verändert werden. (Es sollte genauestens geprüft werden, ob eine vergrößerte Berechnungstiefe erforderlich ist. Die Berechnungstiefe als Antwort auf beliebige Fehler zu erhöhen, hat im allgemeinen fatale Folgen.)

4.3 Basen

4.3.1 `base`–Definitionen

Das Schlüsselwort `base` leitet eine Basisdefinition ein. Eine Basisdefinition wird vervollständigt durch:

- den Namen der Basis, ein Atom
- das Trennzeichen `:<<`
- einer Superklassenspezifikation
- das Trennzeichen `>>:`
- die Basisstrukturbeschreibung
- einen Punkt am Ende (weil es ein vollständiger Ausdruck ist)

Eine Basisdefinition hat damit folgende Form:

`base` *name* `:<<` *superklassenspezifikation* `>>:` *basisstrukturbeschreibung*.

Man beachte den Unterschied der Operatoren in der Definition einer Klasse und einer Basis.

Superklassenspezifikationen sind Ausdrücke, die aus dem Namen einer einzelnen Klasse bestehen oder aus der Verknüpfung von mehreren Namen unter Einbeziehung der Operatoren `&` und `\`. Ein Basisstrukturbeschreibung ist ein *LeX4*–Ausdruck.

Basisdefinitionen sind vollständige *LeX4*–Ausdrücke.

Klammerausdrücke dürfen nicht das `>>:` -Symbol einschließen.

Auch diese Definition gestattet es nun wieder, einen vollständigen *LeX4*– Ausdruck als Beispiel anzugeben:

(10)
```
base sg_nom :<< agr >>: numerus: sg & kasus: nom.
base 'Haus' :<< substantiv_c >>: spell_out: 'Haus' & cat: n.
```

4.3.2 Die Auswertung von Basisdefinitionen

Die Auswertung von Basisausdrücken wird als die Berechnung von **Basisinstanzen** bezeichnet. Diese ist – ganz grob – der Deklarationen einer `struct` in C vergleichbar und ähnlich der Belegung eines *Slot*s in *Frame*–Sprachen.

Der Name der Basis ist der **Identifikator** der Basisinstanz. Dieser Identifikator wird oft auch allein als **Basis** einer Instanz bezeichnet.

Das Ergebnis der Berechnung von Basisausdrücken ergibt sich wie folgt:

- Berechnung der Superklasse(n)
- Berechnung der Basisstruktur gemäß Basisstrukturbeschreibung

- Verknüpfung der beiden Ergebnisse (in allen Varianten) gemäß folgender Regeln führt zu den Resultaten:

  - eine Variable in der Basisstruktur oder der Superklasse wird an den Ausdruck in der jeweils anderen Struktur gebunden
  - eine Konstante in der Basisstruktur oder der Superklasse unifiziert mit einer Variablen oder derselben Konstanten; das Ergebnis ist immer die Konstante, eine Variable wird an die Konstante gebunden
  - eine Attribut–Wert–Struktur in der Basisstruktur oder der Superklasse wird mit einer Variablen (Ergebnis: Attribut–Wert–Struktur, wobei die Variable an die Struktur gebunden ist) oder derselben Struktur (Ergebnis: dieselbe Struktur) oder einer anderen Struktur verbunden, wobei für jedes Attribut in der anderen Struktur gelten muß, daß

    * es in der gegebenen Basisstruktur nicht enthalten ist (Ergebnis: das Attribut aus der Superklasse wird mit seinen Werten der Struktur hinzugefügt)
    * es in der gegebenen Struktur enthalten ist und die Werte der Attribute aus den beiden Strukturen miteinander gemäß dieser Regeln verknüpfbar sind (Ergebnis: Attribut mit dem Wert des Ergebnisses der Verknüpfung der Werte aus den einzelnen Strukturen)
      Das Ergebnis der Verknüpfung zweier Attribut–Wert–Strukturen ist nur genau dann wieder eine Attribut–Wert–Struktur, wenn die resultierende Struktur azyklisch ist.

    Die Verknüpfungsoperation ist offensichtlich der Unifikation *ähnlich*. Man beachte daher besonders die feinen Unterschiede.
  - Zuordnung des Ergebnisses zu *einer* Klasse
    Bezieht eine Basisdefinition durch `&` verbundene Klassen ein, so wird das Ergebnis der Berechnung genau der Klasse zugeordnet, die direkte Subklasse der durch `&` verbundenen Klassen ist. (Das gilt sinngemäß für mehrere direkte Subklassen, wenn es diese gibt.)

Folgendes Beispiel dient der Erläuterung der Berechnung von Basisinstanzen:

(11)
```
class num :< top >: (sg \ pl).
class cas :< top >: (nom \ gen \ acc \ dat).
class numerus_c :< top >: numerus:num.
class kasus_c :< top >: kasus: cas.
class agr :< numerus_c & kasus_c >: top.
base sg_gen :<< agr >>: numerus: sg & kasus: gen.
base sg_dat :<< numerus_c & kasus_c >>: numerus: sg & kasus: dat.
```

Es können zwei Basisinstanzen berechnet werden. Eine Instanz besitzt den Basisidentifikator `sg_gen`, die Struktur `numerus: sg & kasus: gen` und ist der Klasse `agr` zugeordnet. Die andere Instanz besitzt den Basisidentifikator `sg_dat`, die Struktur `numerus: sg & kasus: dat` und ist ebenfalls der Klasse `agr` zugeordnet, obgleich diese Zuordnung in der Definition der Basis nicht explizit gemacht wurde.

Die Bindungsstärke des `>>:` -Operators liegt genau zwischen der des `&` und der des \ -Operators.

4.4 Instanzen

Die Berechnung von **Instanzen** ist **das** grundlegende Verarbeitungsprinzip von *LeX4*. Instanzen gibt es in zwei Möglichkeiten:

- Basisinstanzen
- abgeleitete Instanzen

Die Berechnung von Instanzen wird als **realisieren** von Instanzen bezeichnet.

Jeder Klasse ist mindestens eine Superklasse zugeordnet. Bestimmte Klassen besitzen mehrere durch `&` verknüpfte Superklassen. Darüber hinaus gibt es durch Mehrfachdefinition oder durch Verwendung von \ disjunkte Superklassen.

Aus Sicht der Superklassen sind diese Klassen Subklassen.

Auf diese Art ist eine Klassenhierarchie definiert.

4.4.1 Basisinstanzen

Die Berechnung der Basisinstanzen erfolgt auf der Grundlage der Basisdefinitionen wie in 4.3.2 beschrieben. Eine Basisinstanz ist *genau einer* Klasse zugeordnet, unabhängig davon, ob eine oder mehrere Superklassen in die Basisdefinition einbezogen wurden. Der Name der Basis dient als Identifikator der Instanz.

4.4.2 Abgeleitete Instanzen

Das Realisieren abgeleiteter Instanzen erfolgt auf der Grundlage der Basisinstanzen und der Klassenhierachie.

Eine abgeleitete Instanz bezüglich einer Klasse und einer Basis ist genau dann zu berechnen, wenn der Superklasse (bzw. den Superklassen) ebenfalls Instanzen derselben Basis zugeordnet werden können. Dabei ist es unerheblich, ob diese Instanzen der Superklasse(n) Basisinstanzen oder abgeleitete Instanzen sind.

Die Berechnung der abgeleiteten Instanzen erfolgt auf die gleiche Art wie die Berechnung abgeleiteter Klassen, mit dem Unterschied, daß an der Stelle der Superklasseninformation die Instanz einbezogen wird, die der Superklasse zugeordnet ist und dieselbe Basis besitzt wie die zu berechnende Instanz. Die Auswertung erfolgt auf dieselbe Art und der `>:` -Operator bezeichnet ebenso die Unifikation.

(12)
```
class num :< top >: (sg \ pl).
class cas :< top >: (nom \ gen \ acc \ dat).
class agr :< top >: numerus: num & kasus: cas.
class agr_sg_gen :< agr >: suffix: s.
base sg_gen :<< agr >>: numerus: sg & kasus: gen.
```

Im Beispiel (12) wird eine Basisinstanz mit dem Identifikator `sg_gen` in der Klasse `agr` mit dem Inhalt `numerus: sg & kasus: gen` berechnet. Eine abgeleitete Instanz der selben Basis in der Klasse `agr_sg_gen` umfaßt die Informationen `numerus: sg & kasus: gen & suffix: s`.

5 Weitere Operatoren zur Verknüpfung von $\mathcal{LEX}$-Ausdrücken

Bisher wurden die Operatoren `&` und \ zur Verknüpfung von $\mathcal{LEX}$-Ausdrücken eingeführt. Im folgenden sollen weitere $\mathcal{LEX}$-Operatoren vorgestellt und schließlich die Bindungsstärke aller Operatoren im Überblick gezeigt werden.

5.1 Der Operator -

Der Operator `-` bezeichnet die **Subtraktion**. Subtraktion ist für Variablen und in Basisdefinitionen nicht definiert.

Die Subtraktion zweier nicht–variabler Ausdrücke führt zu folgenden Ergebnissen, wobei eine Attribut-Wert-Struktur als Subtrahend grundsätzlich als Menge von Attribut-Wert-Paaren entsprechend dem Distributivgesetz behandelt wird:

- das Attribut des Minuenden wird aus der Struktur gelöscht, wenn
  - der Subtrahend ein atomarer Wert ist, welcher genau dem Attribut entspricht
  - der Subtrahend dasselbe Attribut bezeichnet und die Subtraktion der Werte der beiden Attribute dazu führt, daß der Wert des Minuenden gelöscht wird (rekursiv)
- das Attribut des Minuenden bleibt in der Struktur erhalten, wenn der Subtrahend dasselbe Attribut bezeichnet und die Subtraktionen der Werte der Attribute nicht als Ergebnis bewirkt, daß der Wert des Minuenden gelöscht wird (rekursiv); das ist genau dann der Fall, wenn der Wert des Minuenden eine Attribut-Wert-Struktur ist, aus der das durch den Subtrahenden bestimmte Attribut-Wert-Paar gelöscht wird
- das Attribut-Wert-Paar des Minuenden wird aus der Struktur gelöscht, wenn der Subtrahend genau das gleiche Attribut-Wert-Paar bezeichnet
- die Subtraktion ergibt eine anonyme Variable, wenn zwei Werte voneinander subtrahiert werden, die nicht als Werte von Attributen eingeführt wurden und identisch sind

Die Operation kann an einem einfachen Beispiel demonstriert werden:

(13)
```
class agr_1 :< top >: numerus: num & kasus: cas.
class agr_2 :< agr_1 >: person: pers - kasus.
```

`agr_2` erbt die Informationen aus der Klasse `agr_1`, fügt das Attribut–Wert–Paar `person: pers` hinzu und löscht das Attribut– Wert–Paar `kasus`, so daß das Ergebnis der Berechnung `numerus: num & person: pers` ergibt.

5.2 Konditional-Operatoren

Die folgenden Operatoren beschreiben Operationen, die kein Ergebnis im bisher üblichen Sinne haben, sondern lediglich dazu dienen, ob ein Ergebnis berechenbar wäre. Man kann sie entfernt mit `if` -Ausdrücken in anderen Sprachen vergleichen.

5.2.1 Der Operator &&

Der `&&` -Operator dient dazu, zu prüfen, ob die Unifikation zweier Ausdrücke möglich ist.

(14)
```
a:1 && b:2      % Ergebnis: a:1
a:1 && a:2      % scheitert
```

5.2.2 Der Operator &~

Der `&~` -Operator dient dazu, zu prüfen, ob die Unifikation zweier Ausdrücke nicht möglich ist.

(15)
```
a:1 &~ a:2      % Ergebnis: a:1
a:1 &~ b:2      % scheitert
```

5.2.3 Der Operator --

Der `--` -Operator dient dazu, zu prüfen, ob die Subtraktion zweier Ausdrücke möglich ist.

(16)
```
a:1 & b:2 -- a:1     % Ergebnis: a:1 & b:2
a:1 & b:2 -- a:3     % scheitert
```

5.2.4 Der Operator -~

Der `-~` -Operator dient dazu, zu prüfen, ob die Subtraktion zweier Ausdrücke nicht möglich ist.

(17)
```
a:1 & b:2 -~ a:3     % Ergebnis: a:1 & b:2
a:1 & b:2 -~ a:1     % scheitert
```

Bindung	Operator
stärkste Bindung	`@@`
	`@`
	`~`, `:`
	`>:`, `&`, `&&`, `-`, `--`, `&~`, `-~`
	`>>:`
	`\`
schwächste Bindung	`::`

Tabelle 1: Operatorenbindung

5.3 Operatorenbindung

Die Bindung der bisher eingeführten und einiger noch zu beschreibender Operatoren in fallender Reihenfolge ist der Tabelle 1 zu entnehmen.

6 Weitere Datentypen

6.1 Zeichenketten

LeX besitzt für Zeichenketten einen speziellen Datentyp. Dadurch können Restriktionen vermieden werden, die etwa bei der Darstellung von Zeichenketten durch Atome bestehen.

Eine Zeichenkette wird dagestellt durch

`$ "`*zeichenkette*`"`

Das `$` -Symbol dient zur Typmarkierung, die Zeichenketteselbst schließt sich in `"` eingebettet an. Die Darstellung `"`*zeichenkette*`"` enspricht unmittelbar der Darstellung von Zeichenketten in Prolog.

(18)
```
$ "Test"
$ "B""aren"       % Darstellungsmoeglichkeit fuer einen Umlaut
$ "Stra"se"       % Darstellungsmoeglichkeit fuer sz

/* weitere Beispiele */

$ "Ist das eine Frage?"
$ "8tung"
$ "B""a:r"
```

Zeichenketten werden von allen bisher behandelten Operationen wie Konstanten be-

handelt. Viele für die Zeichenkettenverarbeitung nützliche Operationen sind als eingebaute Templates vorhanden und werden in 8.2 beschrieben.

6.2 Negation

LeXi4 besitzt gegenwärtig nur äußerst eingeschränkte Möglichkeiten zur Beschreibung von Negation. Es ist möglich, einem Attribut ein Atom mit vorangestelltem ~ -Operator zuzuordnen. Ein entsprechender Ausdruck besitzt folgende Form:

attribut: ~ *wert*

Ein einfaches Beispiel verdeutlicht die Anwendung:

(19) `kasus: ~gen`

Es sei hier noch auf die Operatoren `&~` (5.2.2) und `-~` (5.2.4) hingewiesen.

Anmerkung: In Version 2 von *LeXi4* wird es deutlich erweiterte Möglichkeiten zur Behandlung der Negation geben.

6.3 Vorausschau: Nutzerdefinierte Datentypen

Eine Erweiterung, die ebenfalls in einer Version ab 2 von *LeXi4* umgesetzt werden soll, ist ein Integrationinterface für *nutzerdefinierte Datentypen* . Ein nutzerspezifiertes Datenobjekt kann dann als Wert eines Attributes von *LeXi4* verwaltet, bei Bereitstellung entsprechender Verarbeitungsprädikate auch verarbeitet werden.

7 Referenzen

Um bei der Berechnung von Instanzen einer Klasse auf Instanzen derselben oder einer anderen Basis in einer anderen Klassen zugreifen zu können, gibt es die Möglichkeit, die gewünschten Informationen mit Hilfe einer Referenz auf diese Instanz in der anderen Klasse verfügbar zu machen. Die Angabe geschieht durch

`@` *klasse*

auf eine Instanz derselben Basis in der bezeichneten Klasse oder durch

`@@` *basis* `@` *klasse*

auf eine Instanz einer anderen Basis in der bezeichneten Klasse.

Es ist zu beachten, daß keine Schleife durch die Referenzen und Sub- bzw. Superklassenrelationen entstehen. Andernfalls erscheint eine Fehlermeldung bei der Berechnung.

Durch Referenzen wird ein hoher Vernetzungsgrad der Instanzen erreicht, der sich in der Berechnungskomplexität niederschlägt.

8 Templates

8.1 Nutzerdefinierte Templates

Templates ermöglichen dem Nutzer, mehrfach verwendete Kodesequenzen als eigenständige Ausdrücke zu formulieren und zu nutzen. Als besonders nützlich erweisen sich dabei die sogenannten *parametrisierten Templates*.

Templates werden in LaTeX zur Laufzeit ausgewertet. Dies ist ein markanter Unterschied zu *Macros* in anderen Programmiersprachen, wie etwa C, in welchen derartige Sprachkonstrukte vor der Laufzeit ausgewertet werden.

Die Nutzung des Template–Mechanismus umfaßt zwei Teile:

- die Definition des Templates und
- den Aufruf des Templates.

8.1.1 Templatedefintion

Ein Template wird vollständig definiert durch

`#` *templatename* `::` *ausdruck*`.`

oder

`#` *templatename*`(`*parameterliste*`)` `::` *ausdruck*`.`

`#` leitet eine Templatedefinition ein; es folgt der Name des Templates. Handelt es sich bei dem Template um ein parametrisiertes Template, schließt sich unmittelbar – ohne Leerzeichen – an den Templatenamen die öffnende Klammer `(`, die Liste der Parameter, wobei diese durch Kommata `,` getrennt sind, und letztlich die schließende Klammer `)` an. Parameter sind LaTeX–Ausdrücke. Nach dem Templatenamen und der optionalen Parameterliste in Klammern folgt `::`. Alles sich daran anschließende ist der, gegebenenfalls nach Parametereinsetzung, an Stelle des Templatenamen auszuwertende LaTeX–Kode.

Templates werden unterschieden nach ihrem Namen und der Anzahl der Parameter.

Einige Beispiel können nur stellvertretend die Vielzahl von Definitionsformen zeigen

(20)
```
# nix :: top.
# fritz :: intern: modifikation: fritz.
# 'V' :: syn: sem: hd: dtr: v.
# sg_gen :: agr: (
                  num: sg &
                  cas: gen
                 ).

# wortform(Wortform) :: spell_out: Wortform.
# variante(sg_gen) :: agr: (
                            num: sg &
```

```
                              cas: gen
                             ).
# set(Num, Cas) :: agr: (
                          num: Num &
                          cas: Cas
                         ).
# umbenennen(Alt,Arg) :: Alt - name & name: Arg.
```

Das letzte Template **umbenennen** bezieht die Subtraktion ein. Damit wird natürlich indirekt ausgeschlossen, daß dieses Template in eine Basisdefinition einbezogen wird.

Es ist selbstverständlich möglich, mehrere Template mit demselben Namen und derselben Stelligkeit zu verwenden, die dann als Disjunktionen betrachtet werden.

8.1.2 Templateaufruf

Die Einbeziehung eines Templates erfolgt durch Angabe des Ausdrucks

`#` *templatename*

oder

`#` *templatename*`(`*parameterliste*`)`

Es wird genau das Template ausgewertet, welches denselben Namen und die selbe Anzahl von Parametern besitzt. Die optionalen Parameter in der Parameterliste sind durch Kommata zu trennen und in Klammern gleichfalls unmittelbar hinter den Templatenamen zu stellen.

Es ist zu beachten, daß ein Template als ein geklammerter Ausdruck ausgewertet wird. Es ist nicht möglich, Basis- oder Klassendefinitionen vollständig als Templates zu formulieren.

Korrespondierend zu den Beispielen (20) folgen einige der möglichen Templateaufrufe:

(21)
```
# nix
# fritz
# 'V'
# sg_gen

# wortform($ "Hallo")            /* oder */
          # wortform('Hallo')
# variante(sg_gen)               /* oder */
          # variante(X)
# set(Num, Cas)                  /* oder */
          # set(pl, Cas)         /* oder */
          # set(pl, gen)
# umbenennen(name:joseph, sepp)  /* oder */
          # umbenennen(Name, jupp)
```

Selbstverständlich ist es wie in Beispiel (22) auch möglich, innerhalb einer Templatedefinition einen Templateaufruf zu benutzen. Dies darf nicht zu Schleifen führen.

```
(22) # num(Num) :: numerus: Num.
     # sub_agr(Num, Pers) ::  #num(Num) &
                              person: Pers &
                              kasus: struct_case.
     # att_agr(Num, Gen, Cas) :: #num(Num) &
                                  genus: Genus &
                                  kasus: Cas.
```

8.2 Eingebaute Templates

LEXI umfaßt eine Menge sogenannter *eingebauter Templates*. Diese Menge umfaßt einerseits solche Templates, die von allgemeinem Interesse sind und andererseits solche, die für Beschreibungen dienen, die sich sonst nicht oder nur aufwendig in „purem“ *LEXI* erstellen lassen. Diese Templates bilden damit eine Schnittstelle zu anderen Beschreibungssystemen.

Die Menge dieser Templates wird, in Abhängigkeit von den Problembereichen, in den *LEXI* eingesetzt wird, erweitert. Darüber hinaus ist der hinter den eingebauten Templates stehende Mechanismus bereits dafür vorgesehen, Schnittstellen zu anderen Beschreibungs- und Programmiersprachen zu bilden.

Nutzerdefinierte Templates mit demselben Namen und derselben Stelligkeit wie eingebaute Templates werden von *LEXI* ignoriert.

8.2.1 Zeichenkettenoperationen

Die Menge der Templates für Zeichenkettenoperationen ist bereits sehr umfangreich. Die Namen der Templates wurden in Anlehnung an die Namen ähnlicher Funktionen in der Bibliothek der Programmiersprache C gewählt.

Die eingebauten Templates zur Zeichenkettenverarbeitung beginnen mit `str`. Die Operationen zum Abtrennen von Teilketten werden mit `spl` fortgesetzt, die zum Test auf Zeichenkettengleichheit mit `cmp` und die zum Test auf Zeichenkettenunterschied mit `diff`. Operationen, die sich auf den Anfang der Zeichenkette beziehen, enden auf `a`, die sich auf das Ende beziehen mit `z`. Die Übergabe numerischer Argument widerspiegelt sich gleichfalls im Namen. Als Parameter werden Zeichenketten übergeben.

Die Tests auf Gleichheit oder Unterschied von Zeichenketten besitzen ihre Bedeutung darin, erfolgreich zu sein bzw. fehlzuschlagen, wobei das Ergebnis selbst im allgemeinen nebensächlich ist.

Viele der hier genannten Templates lassen sich durch Konstruktionen unter Einbeziehung anderer Templates ersetzen, zum Teil auch unmittelbar durch andere Templates. Die Benutzung eines speziellen Templates besitzt dann vor allem dokumentarischen Wert.

- `strcat` – Verketten zweier Zeichenketten

  `# strcat(`*prefix*`, `*suffix*`)`

(23) `spell_out: # strcat($ "Haus", $ "es")`

`% ergibt`

`spellout: $ "Hauses"`

- `strspla` – Abtrennen eines Zeichenkettenpräfix
  `# strspla(`*kette*`, `*präfix*`)`

(24) `spell_out: # strspla($ "verschenken", $ "ver")`

`% ergibt`

`spell_out: $ "schenken"`

- `strsplz` – Abtrennen eines Zeichenkettensuffix
  `# strsplz(`*kette*`, `*suffix*`)`

(25) `spell_out: # strsplz($ "Butterseite", $ "seite")`

`% ergibt`

`spell_out: $ "Butter"`

- `strsplna` – Abtrennen eines Zeichenkettenpräfix bestimmter Länge
  `# strsplna(`*kette*`, `*länge*`)`

(26) `spell_out: # strsplna($ "verschenken", 3)`

`% ergibt`

`spell_out: $ "schenken"`

- `strsplnz` – Abtrennen eines Zeichenkettensuffix bestimmter Länge
  `# strsplnz(`*kette*`, `*länge*`)`

(27) `spell_out: # strsplnz($ "Butterseite", 5)`

`% ergibt`

`spell_out: $ "Butter"`

- `strsplma` – Herauslösen eines Zeichenkettenpräfix bestimmter Länge
  `# strsplma(`*kette*`, `*länge*`)`

(28) `spell_out: # strsplma($ "verschenken", 3)`

`% ergibt`

`spell_out: $ "ver"`

- `strsplmz` – Herauslösen eines Zeichenkettensuffix bestimmter Länge

  `# strsplmz(`*kette*`, `*länge*`)`

  (29) `spell_out: # strsplmz($ "Butterseite", 5)`

  `% ergibt`

  `spell_out: $ "seite"`

- `strcmp` – Vergleich von Zeichenketten

  `# strcmp(`*zeichenkette_1*`, `*zeichenkette_2*`)`

  (30) `spell_out: # strcmp($ "Marianne", $ "Marianne")`

  `% ergibt`

  `spell_out: $ "Marianne"`

  Die Funktion dieses Template ist beispielsweise vollständig in *LeVi* ohne Einbeziehung eines anderen Templates zu ersetzen.

- `strcmpa` – Vergleich eines Zeichenkettenpräfixes

  `# strcmpa(`*zeichenkette*`, `*zeichenkettenpraefix*`)`

  (31) `spell_out: # strcmpa($ "Stefan M""uller", $ "Stefan")`

  `% ergibt`

  `spell_out: $ "Stefan M""uller"`

- `strcmpz` – Vergleich eines Zeichenkettensuffixes

  `# strcmpz(`*zeichenkette*`, `*zeichenkettensuffix*`)`

  (32) `spell_out: # strcmpz($ "Peter Meier", $ "Meier")`

  `% ergibt`

  `spell_out: $ "Peter Meier"`

- `strdiff` – Unterschied von Zeichenketten

  `# strdiff(`*zeichenkette_1*`, `*zeichenkette_2*`)`

  (33) `spell_out: # strdiff($ "Helmut", $ "Erich")`

  `% ergibt`

  `spell_out: $ "Helmut"`

- `strdiffa` – Unterschied eines Zeichenkettenpräfixes

  `# strdiffa(`*zeichenkette*`,` *zeichenkettenpraefix*`)`

  (34)
  ```
  spell_out: # strdiffa($ "Osterei", $ "Weihnachts")

  % ergibt

  spell_out: $ "Osterei"
  ```

- `strdiffz` – Unterschied eines Zeichenkettensuffixes

  `# strdiffz(`*zeichenkette*`,` *zeichenkettensuffix*`)`

  (35)
  ```
  spell_out: # strdiffz($ "Weihnachtsmann", $ "Oster")

  % ergibt

  spell_out: $ "Weihnachtsmann"
  ```

- `strumlaut` – Umlautung des letzten umlautbaren Vokals bzw. Diphthongs in einer Zeichenkette durch Voranstellen eines " (exakter: "")

  `# strumlaut(`*zeichenkette*`)`

  (36)
  ```
  spell_out: # strumlaut($ "Dach")
  spell_out: # strumlaut($ "Haus")

  % ergibt

  spell_out: $ "D""ach"
  spell_out: $ "H""aus"
  ```

- `strlen` – Zeichenkettenlänge

  `# strlen(`*zeichenkette*`)`

  (37)
  ```
  laenge: # strlen($ "Fritz")

  % ergibt

  laenge: 5
  ```

- `strstr` – Teilzeichenkette

  `# strstr(`*zeichenkette*`,` *teilkette*`)`

  (38)
  ```
  orth: # strstr($ "Pfeiffer", $ "ff")

  % ergibt

  orth: $ "Pfeiffer"
  ```

8.2.2 Zeichenkettenbehandlung für Atome

Die Templates zur Zeichenkettenbehandlung sind auch in einer Variante für Atome verfügbar. Atomnamen werden dabei intern in Zeichenketten umgewandelt, danach wird das entsprechende Template für Zeichenketten ausgewertet und schließlich wird das Ergebnis wieder in ein Atom übersetzt (gilt natürlich nicht für das Ergebnis des Templates zur Bestimmung der Länge einer Zeichenkette).

Zur Vervollständigung folgt hier die Liste der Templates für Atome:

- `# atomcat(`*prefix*`, `*suffix*`)`
- `# atomspla(`*kette*`, `*präfix*`)`
- `# atomsplz(`*kette*`, `*suffix*`)`
- `# atomsplna(`*kette*`, `*länge*`)`
- `# atomsplnz(`*kette*`, `*länge*`)`
- `# atomsplma(`*kette*`, `*länge*`)`
- `# atomsplmz(`*kette*`, `*länge*`)`
- `# atomcmp(`*zeichenkette_1*`, `*zeichenkette_2*`)`
- `# atomcmpa(`*zeichenkette*`, `*zeichenkettenpraefix*`)`
- `# atomcmpz(`*zeichenkette*`, `*zeichenkettensuffix*`)`
- `# atomdiff(`*zeichenkette_1*`, `*zeichenkette_2*`)`
- `# atomdiffa(`*zeichenkette*`, `*zeichenkettenpraefix*`)`
- `# atomdiffz(`*zeichenkette*`, `*zeichenkettensuffix*`)`
- `# atomumlaut(`*zeichenkette*`)`
- `# atomlen(`*zeichenkette*`)`
- `# atomatom(`*zeichenkette*`, `*teilkette*`)`

8.2.3 Arithmetiktemplates

- `# intcmpgt(`x`, `y`)`
  ist erfolgreich wenn $X > Y$ gilt. Ergebnis ist X.
- `# intcmplt(`x`, `y`)`
  ist erfolgreich wenn $X < Y$ gilt. Ergebnis ist X.
- `# intcmpeq(`x`, `y`)`
  ist erfolgreich wenn X gleich Y ist. Das Template gilt nur für Zahlen.

- `# intalexind(`x`, `y`)`
  hat das Ergebnis von $X + Y$.
- `# intsub(`x`, `y`)`
  hat das Ergebnis von $X - Y$.
- `# intmul(`x`, `y`)`
  hat das Ergebnis von $X * Y$.
- `# intdiv(`x`, `y`)`
  hat das Ergebnis von $X : Y$.

8.2.4 Verschiedenes

- Konvertierung von Atomen in Zeichenketten und umgekehrt
  Für diese Aufgabe gibt es zwei Templates.
  `# str2atom(`*string*`)`
  wandelt die Zeichenkette *string* in ein Atom um, wie es Beispiel (39) zeigt.

  (39)
  ```
  xyz: # str2atom($"Test")          % Ergebnis ist xyz: 'Test'
  ```

  `# atom2str(`*atom*`)`
  wandelt das Atom *atom* in eine Zeichenkette um, wie es Beispiel (40) verdeutlicht.

  (40)
  ```
  xyz: # atom2str('Test')          % Ergebnis ist xyz: $ "Test"
  ```

- Kopieren
  Um einen Wert echt zu kopieren, gibt es das Template
  `# copy(`*wert*`)`
  Auch hier soll ein Beispiel die Funktionsweise zeigen.

  (41)
  ```
  agr: (num: Num &
        Val) &
      sub: #copy(Val) &
      agr: cas: gen

      /* Es ergibt sich folgende Struktur: */

      agr: (num: Num &
            cas: gen) &
      sub: num: Num
  ```

9 Interaktionsprädikate

In diesem Abschnitt werden die Prädikate eingeführt, die den Nutzer in die Lage versetzen, den LeX4-Interpreter zu starten und mit diesem in Interaktion zu treten.

9.1 Anmerkung zu den Datenstrukturen in der Interaktion

Der Nutzer wird in der Interaktion zum Teil (noch) mit einer anachronistischen Datendarstellung der Attribut–Wert–Struktur konfrontiert, die ihre Wurzeln in früheren Anforderungskatalogen hat. Allerdings bestätigten sich die erwarteten Vorteile dieser vereinfachten Schnittstelle in der späteren Nutzung nicht, vielmehr stifftete sie eher Verwirrung. Spätestens ab Version 2 von LeX4 wird diese Darstellung ersetzt durch die vollständige Darstellung von LeX4-Ausdrücken, wie sie in den vorangegangenen Kapitel beschrieben wurde.

In dieser vereinfachten Datendarstellung werden Attributmengen als Prolog–Listen repräsentiert. Ein Beispiel sollte zur Verdeutlichung genügen:

(42)
```
a:anton & b:berta          % LeX4
[a:anton, b:berta]         % veralteter Zwischencode
```

9.2 Prolog–Besonderheiten

Anmerkung: Mit Prolog vertraute LeX4-Nutzer können die folgenden Anmerkungen überspringen.

Die Interaktion mit dem LeX4-Interpreter erfolgt über den Prolog–Interpreter des Prolog–Systems, welches der jeweiligen LeX4-Version zugrundeliegt. Details sind dem Systemhandbuch zu entnehmen.

Nach dem Start des Prolog–Systems meldet sich dieses mit einem *Prompt*. Im allgemeinen hat dieses ein Form wie `?-`. Der Nutzer kann nun seine Eingabe in Form von Prolog–Prädikaten machen. Die Prädikate zur Interaktion mit LeX4 werden nachfolgenden im Einzelnen vorgestellt.

Alternativ kann der Nutzer seine Eingaben auch in eine Datei schreiben, die er nach dem Start oder beim Start des Prolog–Sytems lädt. Jeder Eingabe ist dabei die Zeichenkombination `:-` voranzustellen (quasi ein Ersatzprompt).

Bei der Beschreibung bzw. Eingabe der Prolog–Prädikate sind insbeondere zwei Besonderheit zu beachten:

- zwischen dem Namen des Prädikats und der bei bestimmten Prädikaten folgenden Klammer darf **kein** Leerzeichen stehen
- jede Eingabe ist mit einem Punkt `.` zu beenden

Für alle, die nicht mit Prolog vertraut sind, sei empfohlen, die notwendigen Befehlssequenzen aus einer der Beispielbeschreibungen zu übernehmen.

9.3 Elementare Prädikate

9.3.1 Laden von LeX4–Beschreibungen

Nach dem Start des PROLOG–Systems mit dem LeX4-Interpreter sind als erstes die auszuwertenden LeX4-Beschreibungen zu laden. Dies geschieht mit dem PROLOG–Prädikat

`consult('`*dateiname*`').`

oder abgekürzt mit

`['`*dateiname*`'].`

Eine entsprechende Zeile aus einer PROLOG–Sitzung mit dem Laden der LeX4-Datei `test.lx` ist

```
consult('test.lx').
```

In einer Datei sieht der entsprechende Befehl wie folgt aus

```
:-consult('test.lx').
```

(Das `:-` Symbol muß dabei vom Nutzer geschrieben werden.)

9.3.2 Berechnungsschritte

Um eine LeX4-Beschreibung nach dem Laden auszuwerten, sind folgende Prädikate vorhanden:

- `classes.`
  berechnet alle Klassenbeziehungen
- `bases.`
  berechnet alle Basisinstanzen
- `realize.`
  berechnet alle abgeleiteten Instanzen

Die Berechnungen bauen in dieser Reihenfolge aufeinander auf. Die Prädikate sollten daher in dieser Reihenfolge eingegeben werden.

Die Berechnungsschritte lassen sich aufgrund von Kontrollausschriften verfolgen. Die Ausschrift kann durch Veränderung der Parameterdatei von LeX4 unterdrückt werden.

9.3.3 Datenausgabe

Das Prädikat zur Datenausgabe erlaubt eine Reihe von Anpassungen. Dadurch ist es etwas komplexer. Die Grundstruktur des Prädikates ist

`frealize(`*datei*`, `*muster*`, `*nutzer*`, `*muster*`).`

oder

`frealize(`*datei*`, `*ausgabe*`, `*nutzer*`, `*muster*`, `*klasse*`).`

Die Bedeutung der einzelnen Argumente ergibt sich wie folgt:

- Datei (*datei*)

  Die Spezifikation der Ausgabedatei erfolgt durch ein Paar

  `file(`*dateiname*`,` *mode*`)`

  Der Dateiname ist ggf. in Quotes anzugeben. Das Argument *mode* korrespondiert zum entsprechenden Argument im Prädikat *open* des zugrundeliegenden PROLOG–Systems. Zu seiner Spezifikation ist das Handbuch heranzuziehen.

- Ausgabe (*ausgabe*)

  Die eigentliche Ausgabe besteht aus der Instanz einer bestimmten Klasse und Basis. Dies entspricht dem Tripel

  `realize(`*basis*`,` *instanz*`,` *class*`)`

  Bei der schrittweisen Abarbeitung des Prädikates `frealize/[4,5]` werden die Argumente nacheinander mit den möglichen Werten belegt. Diese können dann als Werte dem *muster*–Argument in `frealize/[4,5]` übergeben werden.

- Nutzer (*nutzer*)

  Der Nutzer kann an dieser Stelle ein Prädikat übergeben dessen erstes Argument eine Struktur besitzt, die der des Arguments *muster* in `frealize/[4,5]` entspricht. Damit kann der Nutzer noch direkter Einfluß auf die Ausgabe nehmen, als dies mit dem Argument *muster* möglich ist. Das ermöglicht aber auch, daß der Nutzer die Ausgabedaten der Berechnung bevor sie in die Ausgabedatei geschrieben werden, nach seinen Kriterien evaluieren kann.

  Benötigt der Nutzer diese Funktionalität nicht, ist anstelle des Prädikates eine Variable zu übergeben.

- Muster (*muster*)

  Muster ist eine Liste der auszugebenden Informationen. Die Elemente der Liste werden dabei nacheinander in die Ausgabedatei geschrieben. Sollen Elemente der Liste analog zur Ausgabe eines Terms mittels `write_term`–Prädikat des PROLOG–Systems konvertiert werden, ist es als Bestandteil des Paares

  `p(`*element*`,` *o_mode*`)`

  aufzunehmen und in der Liste durch dieses Paar zu ersetzen. Die Konvertierung wird mit Hilfe des Arguments *o_mode* definiert, welches dem entsprechenden Argument im Prädikat `write_term` des PROLOG–Systems entspricht. Zu seiner Beschreibung ist das PROLOG–Handbuch zu konsultieren.

- Klasse (*klasse*)

  Das optionale Argument *klasse* ist eine Liste der Klassen, die ausgegeben werden sollen.

9.3.4 Zusammenfassendes Beispiel

Die eingeführten Prädikate sollen anhand eines gemeinsamen Beispiels nochmals gezeigt werden. Nutzer, die sich mit den zuletzt diskutierten PROLOG–Prädikaten nicht auseinandersetzen wollen, können das Beispiel (mit Ausnahme der ersten Zeile zum Laden der Datei) einfach als letzte Sequenz an ihre *LEX4*–Beschreibung anfügen. Die Sequenz wird nach dem Laden sofort ausgeführt und erstellt ein Ausgabedatei mit den Instanzen der Klasse `test_c` entsprechend der zugrundeliegenden *LEX4*-Beschreibung (die natürlich diese Klasse dann als die gewünschte Zielklasse umfassen muß).

(43)
```
:-['lexikon.lx'].

:-classes.
:-bases.
:-realize.
:-frealize(file('lexikon.out', write),
                realize(Base, Content, _),
                _,
                ['lex_entry(',
                 p(Base,[quoted(true)]),
                 ', ',
                 p(Content, [quoted(true)]),
                 ').'
                ],
                [test_c]).
```

9.4 Prädikate zur Fehlersuche

*LEX4*bietet zur Fehlersuche zwei Arten von Prädikaten

- Berechnung einzelner Ausdrücke
- schrittweise Berechnung einzelner Ausdrücke

9.4.1 Berechnung einzelner Ausdrücke

Die Berechnungsprädikate gibt es in einer Version, die dem Nutzer unmittelbar das Ergebnis einer einzelnen Berechnung zeigt:

- `base(`*basis*`,` *inhalt*`,` *superklasse*`)`

  Das Argument *basis* ist zu spezifizieren. Dieses Prädikat kann zum Beispiel zur Inspektion der Basisinstanzen eingesetzt werden.

- `class(`*klasse*`,` *struktur*`,` *superklasse*`)`

  Das Prädikat kann nur bei nicht-eigenrekursiven Klassendefinitionen ohne Referenzen angewendet werden. Das Argument *klasse* ist zu spezifizieren. Damit kann etwa die Vererbung von Informationen verfolgt werden.

- `quick_class`(*klasse*, *superklasse*)

  Das Argument *klasse* sollte spezifiziert werden. Das Prädikat zeigt Klassenrelationen auf.

- `realize`(*basis*, *instanz*, *klasse*)

  Das Argument *basis* ist zu spezifizieren, *klasse* sollte spezifiziert sein.

9.4.2 Schrittweise Berechnungen

Zu den im vorigen Abschnitt eingeführten Prädikaten gibt es korrespondierende Prädikate, die bei einem Fehlschlagen der Berechung den zu berechnenden $\mathcal{QVI}$-Ausdruck solange zerlegen, bis eine Lösung für den verbliebenen Restausdruck möglich ist. Die Prädikate lauten:

- `base_debug`(*basis*, *inhalt*, *superklasse*)
- `class_debug`(*klasse*, *struktur*, *superklasse*)
- `quick_class_debug`(*klasse*, *superklasse*)
- `realize_debug`(*basis*, *instanz*, *klasse*)

Führt die Berechnung zu keinem Ergebnis, kann der Nutzer den weitern Ablauf durch Eingabe einzelner Zeichen wie folgt beeinflussen:

- Eingabetaste (*enter*, *return*, ...)

  Das letzte Element des zu berechnenden Ausdrucks wird abgespalten. Danach wird versucht den verbliebenen Restausdruck zu berechnen.

- `a`

  Die Berechnung wird abgebrochen.

- `f`

  Es werden alternative Ausdrücke zur Berechnung einbezogen.

Führt die Berechnung zu einem Ergebnis, kann der Nutzer den weiteren Ablauf wie folgt beeinflussen:

- Eingabetaste (*enter*, *return*, ...)

  Die Berechnung wird ausgeführt und das Ergebnis zurück gegeben.

- `f`

  Es werden alternative Ausdrücke zur Berechnung einbezogen.

10 Vorschau: Datenbankanbindung

LeXi4 bietet in der vorliegenden aktuellen Version keine Unterstützung zur Datenspeicherung. Der Bedarf nach einer systematischen Datenverwaltung bei der Arbeit mit LeXi4 wird jedoch an zwei Stellen besonders offensichtlich:

- Bei umfangreicheren Lexika wächst – bei systematischer Organisation – auch die Anzahl der LeXi4-Beschreibungsdateien. Dabei erhöht sich in erster Linie die Anzahl der Basisdefinitionen. Aus diesem Grund ist vorgesehen, ein Datenbanksystem zur Verwaltung dieser Definitionen einzusetzen. Ob die Integration dieses Verwaltungssystems als Bestandteil von LeXi4 erfolgt oder aber als Bestandteil eines Lexikographenarbeitsplatzes, ist noch zu bestimmen. Die letztere Lösung würde vom gegenwärtigen Stand der Entwicklung bevorzugt werden.

- Die Berechnungsdauer wächst mit dem Umfang der Lexika. Die vollständige Berechnung eines Lexikons mit deutlich mehr als 10 000 Einträgen führte zu Berechnungszeiten die sich in Richtung einer Stunde bewegten. Dies erscheint einerseits recht lang, andererseits: Kein Nutzer schreibt ein Lexikon mit 10 000 Einträgen aufeinmal, allenfalls übernimmt er soviel Einträge aus einem anderen Lexikon. Aus diesem Grund soll und kann die Berechnung des Lexikons *inkrementell* erfolgen, das heißt, es wird bei jedem LeXi4–Lauf nur das berechnet, was sich im Vergleich zum vorherigen Lauf verändert hat (etwa der Arbeitsweise von `make` vergleichbar). Das bedeutet, daß die Ergebnisse vom jeweils davor liegenden Lauf gespeichert werden müssen. Genau für diese Funktion wird in einer der Folgeversionen ein Datenbanksystem integriert.

A Indizes

A.1 Sprachelemente von LEX

A.2 Gesamtindex