Hausarbeit
zur Erlangung des Magistergrades
an der Ludwig-Maximilians-Universität München

Erstellen einer Grammatik für das Lateinische im „Grammatical Framework“

vorgelegt von Herbert Lange
Matrikelnummer 8063320

Fach: Computerlinguistik

Referent: Prof. Dr. Klaus U. Schulz

München, den 31.10.2013

Zusammenfassung

In dieser Arbeit sollen an einem konkreten Beispiel die nötigen Schritte gezeigt werden, um eine computergestützte Grammatik für eine natürliche Sprache zu entwerfen. Es wird anhand der lateinischen Sprache gezeigt, wie eine Grammatik, bestehend aus einem Lexikon, einem Morphologiesystem und einer Syntax, implementiert werden kann, die sich in einem größeren, multilingualen Grammatiksystem verwenden lässt. Dadurch können in der implementierten Sprache Sätze verarbeitet werden und auch in jede andere im System vorhandene Sprache übersetzt werden. Gezeigt wird ein rein regelbasierter Ansatz der sich von den statistischen Methoden durch seine Striktheit und Beschränktheit, aber auch durch seine Zuverlässigkeit abheben soll.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Im Bereich der Computerlinguistik haben sich im Laufe der Zeit zwei Lager gebildet, die jeweils ihren Ansatz zur Sprachverarbeitung vertreten. Der heute häufiger anzutreffende Ansatz ist der statistische Ansatz, denn nach aktuellem Stand kann man durch statistische Methoden in der Sprachverarbeitung mit relativ geringem Aufwand brauchbare bis gute Ergebnisse erzielen. Allerdings bedarf der statistische Ansatz möglichst guter Trainingsdaten, die nicht immer leicht zu beschaffen und zu bewerten sind.

Der zweite, ältere Ansatz, ist die regelbasierte Sprachverarbeitung. Er prägte die Anfänge der Computerlinguistik stark, wurde jedoch im Laufe der Zeit vom statistischen Ansatz verdrängt. Dies ist unter anderem auf die steigende Leistung heutiger Rechner zur Verarbeitung großer Datenmengen und vor allem auf die Fülle an Daten, die über das Internet verfügbar sind, zurückzuführen. Die Grundlagen regelbasierter Grammatiken sind in etwa so alt wie die Wissenschaft der Linguistik selbst und sollten auch weiterhin zum Wissen eines jeden Computerlinguisten gehören.

In dieser Arbeit werden an einem konkreten Beispiel die nötigen Schritte gezeigt, um eine computergestützte Grammatik für eine natürliche Sprache zu entwerfen. An der lateinischen Sprache wird exemplarisch gezeigt, wie ein Lexikon, ein Morphologiesystem und eine Syntax implementiert werden können, die sich in ein größeres, multilinguales Grammatiksystem einfügen lassen. Eine Sprache wie Latein, die zum einen für ihre Regelmäßigkeit aber auch für ihre linguistischen Besonderheiten bekannt ist, kann zu interessanten Erkenntnissen im Bereich der Grammatikentwicklung führen.

Ein weiterer Aspekt dieser Arbeit ist es, einen Beitrag zu einem größeren Projekt zu leisten. Denn die lateinische Grammatik, die im Rahmen dieser Arbeit entwickelt wurde, fließt direkt in das Grammatical Framework[1], das für diese Arbeit gewählten multilingualen Grammatik-, Parsing- und Übersetzungssystem ein.

[1] `http://www.grammaticalframework.org/`

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, zum einen ein soweit funktionstüchtiges Grammatiksystem zu entwickeln, dass es in der Lage ist, grundlegende Sätze zu verarbeiten und durch die Integration in ein multilinguales Grammatiksystem in andere, moderne, Sprachen zu übersetzen. Zum anderen sollen aber auch die allgemeinen Schritte einer Grammatikentwicklung exemplarisch an der lateinischen Sprache dargelegt werden.

Der Schwerpunkt soll dabei zunächst auf der Nähe zu einer gewöhnlichen, im bayerischen Gymnasialunterricht verwendeten, lateinischen Schulgrammatik liegen. Dies spiegelt sich in der Abfolge der Schritte und in einigen Entscheidungen beim Entwurf der Grammatik wieder. So ist eine lateinische Grammatik grob in folgende Abschnitte unterteilt: Phonologie, Wortarten und Wortbildung, Morphologie und Syntax. Zwar werden die ersten drei Teile in dieser Arbeit nur kurz angeschnitten, die logische Folge der zwei verbleibenden Teile wird hier aber beibehalten. Zusätzlich wird ihnen der üblicherweise in Grammatiken nicht in dieser Form auffindbare Teil über die Lexikonentwicklung vorangestellt.

Die Grammatik soll zusätzlich im bestehenden System des Grammatical Frameworks verwendet werden können. Deshalb müssen möglichst große Teile der für eine Sprache vom Grammatical Framework geforderten Schnittstellen zur Verfügung gestellt werden. Diese Schnittstellen werden im Grammatical Framework in der Form einer so genannten abstrakten Syntax definiert, die im folgenden noch genauer beschrieben wird. So wird eine Menge von Modulen, Funktionen und Regeln definiert, die in einer Grammatik enthalten sein müssen. Das geplante Ziel ist es zu ermöglichen, dass alle von dieser Grammatik beschriebene Sätze in andere Sprachen, die von der Ressource Grammar Library des Grammatical Frameworks unterstützt werden, übersetzt werden können.

1.3 Aufbau der Arbeit

Die Arbeit unterteilt sich in die drei Teile „Grundlagen“, „Grammatikentwicklung“ und ein Fazit. Die Grammatikentwicklung ist wiederum in die schon erwähnten Abschnitte über Lexikon, Morphologie und Syntax unterteilt.

Zu Beginn der Arbeit werden in Kapitel 2 die nötigen Grundlagen für die weiteren Teile der Arbeit erörtert. Die Grundlagen des Grammatical Frameworks werden in Abschnitt 2.1 erklärt. Diese bestehen aus einer kurzen Beschreibung des Umfangs und der Funktionen dieses Programmpakets. Es folgt eine Einführung in den Formalismus, der bei der Entwicklung der Grammatiken für das Grammatical Framework verwendet wird. Dieser Abschnitt wird mit einigen Informationen zur Ressource Grammar Library, der multilingualen Grammatikbibliothek des Grammatical Frameworks, abgeschlossen.

Nach den technischen Grundlagen folgen einige Informationen zur lateinischen Sprache in Abschnitt 2.2. Zunächst wird eine sprachwissenschaftliche Einordnung dieser Sprache unter besonderer Hervorhebung einiger interessanter Merkmale versucht. Abschließend wird die Relevanz dieser als tot geltenden Sprache in der heutigen Zeit diskutiert.

Nach dieser allgemeinen Einführung werden im nächsten Kapitel, dem Kapitel 3, die nötigen Schritte, die umgesetzt wurden, um eine Lateingrammatik im Grammatical Framework zu entwickeln, beschrieben. Dieses Kapitel ist in die drei Abschnitte Lexikon, Morphologie und Syntax unterteilt, da diese drei getrennte Module in der Ressource Grammar Library bilden. Bei der Entwicklung der Grammatik zeigten sich allerdings oft Abhängigkeiten zwischen den drei Bestandteilen, so dass im Laufe der Zeit auch Änderungen in anderen Komponenten nötig waren. Im Abschnitt 3.1 wird dargestellt, wie das Lexikon, das für eine Grammatik im Grammatical Framework nötig ist, erstellt wird. Darauf folgt in Abschnitt 3.2 die Beschreibung der lateinischen Wortflexion und wie sie im Grammatical Framework umgesetzt werden kann. Als letzter Teil dieses Kapitels wird in Abschnitt 3.3 erläutert, welche syntaktischen Regeln in der Grammatik nötig sind, um eine grundlegend funktionierende Grammatik zu erhalten, was zu den Hauptzielen dieser Arbeit gehört.

Abgerundet wird die Arbeit in Kapitel 4, in dem ein Fazit der Arbeit gezogen und ein Ausblick auf mögliche Erweiterung und Verwendung gegeben wird. So wird gezeigt, welchen Sprachumfang die Grammatik bisher umfasst, welche Erweiterungen gewinnbringend sein können und auch in welchen Bereichen das Ergebnis dieser Arbeit Anwendung finden kann.

1.4 Konventionen

Zum Schluss dieser Einleitung sollen noch kurz die in dieser Arbeit verwendeten typografischen Konventionen erläutert werden. Grammatik-Quelltexte des Grammatical Frameworks, so wie Teile daraus, Befehle auf der Betriebssystem-Eingabeaufforderung und Befehle in einer interaktiven Sitzung des Grammatical Framework-Systems werden in einer `schreibmaschinenähnlichen Schrift` gesetzt. Befehle, die in der Betriebssystem-Eingabeaufforderung einzugeben sind, sind an einer Eingabeaufforderung in Form von `$` vor dem Befehl und Befehle im Grammatical Framework-System an einem `>` zu erkennen. So ist `cat S;` ein Ausschnitt aus einem Quelltext, `$ gf` ein Befehl in der Betriebssystem-Eingabeaufforderung und `> generate_random` ein Befehl in einer interaktiven Grammatical Framework-Sitzung.

Alle Dateinamen und Pfade werden **fett** dargestellt. Ist bei einer Datei kein Pfad angegeben, so befindet sie sich im Verzeichnis, in dem die Lateingrammatik enthalten ist. Dieses Verzeichnis befindet sich entweder auf der beiliegenden CD oder im offiziellen Quelltext des Grammatical Frameworks im Unterverzeichnis **lib/src/latin**. Ist dagegen ein Ordnerpfad angegeben, so ist dieser relativ zu einem Verzeichnis, in dem sich der Quelltext des Grammatical Frameworks befindet. In Quelltext-Listings werden zur einfacheren Lesbarkeit auch Schlüsselwörter zusätzlich **`fett`** gedruckt.

Lateinische Wörter werden zur leichteren Erkennbarkeit *kursiv* gesetzt. Zusätzlich sind in Quelltext-Listings die Inhalte von Zeichenketten ebenfalls ***`kursiv`***.

Endet eine Zeile in einem Listing mit einem umgekehrten Schrägstrich („\“) so bedeutet dies, dass lediglich aus Platzgründen ein Zeilenumbruch eingefügt werden musste und die nächste Zeile eine Fortführung der Zeile an dieser Stelle darstellt.

2 Grundlagen

2.1 Das Grammatical Framework

Das Grammatical Framework ist ein Softwaresystem mit einer spezialisierten Programmiersprache zur Entwicklung von Grammatiken. Es benutzt Formalismen, wie sie auch in modernen funktionalen Programmiersprachen wie Haskell zu finden sind, unterstützt aber vor allem die Entwicklung von Anwendungen zur Verarbeitung natürlicher Sprachen.[1] Somit können einem manche Konzepte bereits vertraut sein, wenn man sich schon mit den Möglichkeiten der funktionalen Programmierung[2] auseinander gesetzt hat. Ein großer Vorteil des Grammatical Frameworks im Vergleich zu anderen Parsingsystemen ist, dass durch das Typsystem, das unter anderem auf der Typtheorie von Martin-Löf basiert, Grammatikfehler schon durch den Compiler erkannt werden können.[3]

Die große Stärke des Grammatical Frameworks ist die Multilingualität. Das Grundkonzept für die Unterstützung verschiedener Sprachen ist die Trennung in eine konkrete und eine abstrakte Repräsentation der Grammatik. Dabei haben die unterschiedlichen Sprachen eine gemeinsame abstrakte Struktur. Die konkrete Syntax dagegen beschreibt, wie aus einer sprachunabhängigen Repräsentation eines „Satzes“ eine für die jeweilige Sprache spezifische Zeichenkette erzeugt werden kann. Über diesen Schritt der abstrakten Repräsentation kann man eine Übersetzung zwischen verschiedensten Sprachen umsetzen, die eine gemeinsame abstrakte Syntax teilen.[4] Die Details dieses Formalismus sollen nun an einigen kleinen deutsch-englischen Beispielen genauer betrachtet werden.

[1] vgl. [Ran11] S. vii
[2] Beispiele für funktionale Programmiersprachen sind Lisp, SML und Haskell
[3] vgl. [Ran11] S. 127ff.
[4] vgl. [Ran11] S. 10ff.

2.1.1 Der Grammatikformalismus

Im Bereich der Computerlinguistik und Informatik werden hauptsächlich kontextfreie Grammatiken, also Grammatiken von Typ 2 der Chomsky-Hierarchie, verwendet.[5] Die zwei Hauptgründe dafür sind, dass die Mächtigkeit dieses Formalismuses meist ausreicht, die gewünschten Sprachen zu beschreiben. So sind in kontextfreien Sprachen geklammerte Ausdrücke möglich, die bei Programmiersprachen recht häufig sind.[6] Zudem ist der Verarbeitungsaufwand gering im Vergleich zu Grammatiken einer der höheren Stufen, und es existieren effiziente Algorithmen zum Parsen mit kontextfreien Grammatiken, so z.B. der Cocke-Younger-Kasami-Algorithmus, auch bekannt als CYK-Algorithmus.[7] Die in Beispiel 1 gegebene Grammatik ist ein

1	S	$\longrightarrow$	NP, VP
2	NP	$\longrightarrow$	Det, N
3	N	$\longrightarrow$	*Mann*
4	Det	$\longrightarrow$	*der*
5	VP	$\longrightarrow$	V
6	V	$\longrightarrow$	*schläft*

Beispiel 1: Kurzes Beispiel einer kontextfreien Grammatik

sehr minimalistisches Beispiel für eine kontextfreie Grammatik. Mit ihrer Hilfe kann nur der eine deutsche Satz „Der Mann schläft“ hergeleitet werden. Eine mögliche Ableitung hat die in Beispiel 2 gezeigte Form. Bei dieser Ableitung wurde top-down vorgegangen, also von der allgemeinsten Kategorie hinab bis zur spezifischen Zeichenkette. Alternativ wäre es auch möglich gewesen, eine bottom-up-Ableitung anzugeben, die lediglich dem Ablesen der gegebenen Ableitung von unten nach oben entspricht. In der Grammatik sind die Regeln zum einfacheren Bezug auf die Grammatik mit Regelnummern versehen. Diese Regelnummern sind deshalb auch in der Ableitung und dem entsprechenden Syntaxbaum zu sehen.

Im Formalismus des Grammatical Frameworks wird die oben gegebene Grammatik in eine abstrakte und eine konkrete Syntax zerlegt. So entspricht diese abstrakte Syntax in etwa dem Syntaxbaum in Abbildung 2.1 ohne die terminalen Blätter.

Die abstrakte Syntax der kontextfreien Grammatik aus Beispiel 1 ist in Listing 2.1 zu sehen. Zunächst gibt das Schlüsselwort `abstract` an, dass es sich um eine Datei mit einer abstrakten Syntaxbeschreibung handelt. Diesem Schlüsselwort folgt der Name und nach dem `=` der Inhalt der Grammatik.

[5] vgl. [Sch08] S. 9f
[6] vgl. [Sch08] S. 43
[7] vgl. [Sch08] S. 56f.

S
$\overset{1}{\Rightarrow}$ NP VP
$\overset{2}{\Rightarrow}$ Det N VP
$\overset{4}{\Rightarrow}$ *der* N VP
$\overset{3}{\Rightarrow}$ *der Mann* VP
$\overset{5}{\Rightarrow}$ *der Mann* V
$\overset{6}{\Rightarrow}$ *der Mann schläft*

Beispiel 2: Ableitung des Satzes mit Hilfe der Grammatik aus Beispiel 1

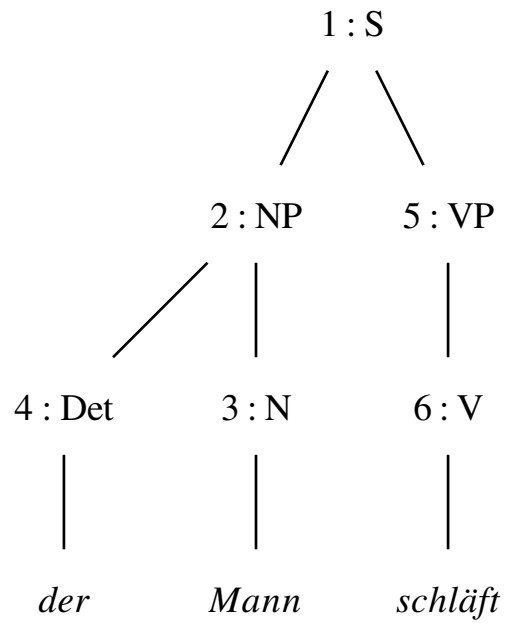

Abbildung 2.1: Syntaxbaum für die Ableitung in Beispiel 2

```
1 abstract MiniSatzAbs = {
2   flags startcat = S ;
3   cat S ; NP ; VP ; Det ; N ; V ;
4   fun
5     mkNP : Det -> N -> NP ;
6     mkVP : V -> VP ;
7     mkS : NP -> VP -> S ;
8     der_Det : Det ;
9     Mann_N : N ;
10    schlafen_V : V ;
11 }
```

Listing 2.1: Abstrakte Syntax der Grammatik aus Beispiel 1

Zuerst werden mit Hilfe der `flags`-Direktive einige mögliche Einstellungen vorgenommen. In diesem sehr kurzen Beispiel wird nur die Startkategorie für das Parsing gesetzt, also die Wurzel aller Parsebäume. Andere mögliche Optionen sind z.B. die Einstellungen des Encodings und der Lexer, also das Programm, das die Eingabe in lexikalische Tokens zerlegt.[8]

Nach dem Schlüsselwort `cat` folgt eine Liste der Kategorien oder auch Non-Terminal-Symbole. Sie entsprechen Datentypdefinition in (funktionalen) Programmiersprachen.

Hauptbestandteil der Grammatik sind die Syntaxregeln. Sie werden nach dem Schlüsselwort `fun` aufgelistet. Die Regeln ähneln der Form von Funktionssignaturen in Sprachen wie Standard ML oder Haskell, denn sie beschreiben lediglich die Bestandteile, aus denen ein Ausdruck einer neuen Kategorie zusammengesetzt werden soll. Das genaue Vorgehen für die Kombination dieser Bestandteile wird dann sprachspezifisch in jeder konkreten Grammatik, die diese abstrakte Grammatik implementiert, beschrieben. So sagt die erste Regel mit dem Namen `mkNP` aus, dass ein Ausdruck der Kategorie `NP` aus einem Ausdruck der Kategorie `Det` und aus einem Ausdruck der Kategorie `N` zusammengesetzt werden kann. Dabei ist aber noch keine Aussage über die endgültige Reihenfolge der Bestandteile in konkreten Zeichenketten getroffen. Die letzten drei Regeln führen lediglich die lexikalische Einheiten mit einer entsprechenden Kategorie ein.

Diese abstrakte Grammatik kann nun konkret umgesetzt werden. Zwei konkrete Implementierungen, für Deutsch und Englisch, sind in Listing 2.2 und 2.3 zu finden.

```
1  concrete MiniSatzGer of MiniSatzAbs = {
2    flags coding=utf8;
3    lincat S,NP,VP,Det,N,V = Str;
4    lin
5      mkNP det n = det ++ n ;
6      mkVP v = v ;
7      mkS np vp = np ++ vp ;
8      der_Det = "der" ;
9      Mann_N = "Mann" ;
10     schlafen_V = "schläft" ;
11 }
```

Listing 2.2: Konkrete deutsche Syntax für die abstrakte Syntax in Listing 2.1

Zunächst weist das Schlüsselwort `concrete` die Grammatik als eine konkrete

[8] vgl. [Ran11] S. 54f.

```
concrete MiniSatzEng of MiniSatzAbs = {
  lincat S,NP,VP,Det,N,V = Str;
  lin
    mkNP det n = det ++ n ;
    mkVP v = v ;
    mkS np vp = np ++ vp ;
    der_Det = "the" ;
    Mann_N = "man" ;
    schlafen_V = "sleeps" ;
}
```

Listing 2.3: Konkrete englische Syntax für die abstrakte Syntax in Listing 2.1

Grammatik aus. Es folgt wie bei einer abstrakten Syntax der Name der Grammatik, diesmal wird jedoch darauf folgend angegeben, welche abstrakte Grammatik die Grundlage bietet, hier unsere `MiniSatzAbs`-Grammatik.

Das in der deutschen, konkreten Grammatik verwendete Flag `coding` ermöglicht es, die Zeichenkodierung in den Zeichenketten festzulegen. In diesem Falle ist es für die deutschen Umlaute nötig, die Zeichenkodierung anzugeben. Für andere Sprachen mit komplett vom lateinischen unterschiedlichen Schriftsystemen gibt es auch die Möglichkeit, statt der direkten Zeichenkodierung eine Transliteration zu verwenden. Dafür wird eine bijektive Abbildung zwischen Unicode-Zeichen und Zeichenketten der Länge eins oder größer, die nur aus ASCII-Zeichen bestehen, verwendet.[9]

Das Schlüsselwort `lincat` ist die konkrete Entsprechung zum Schlüsselwort `cat` in der abstrakten Syntax. Hier muss für jede in der abstrakten Syntax angegebene Kategorie ein konkreter Datentyp angegeben werden. In diesem Falle wurde für alle Kategorien der einfache Datentyp `Str`, also eine einfache Zeichenkette[10], gewählt. Das Grammatical Framework unterstützt auch verschiedene Arten komplexer Datentypen, die später näher erläutert werden.

Auf den `lincat`-Block folgt, mit dem Schlüsselwort `lin` markiert, der Abschnitt, in dem die für jede abstrakte Syntaxregel beschrieben wird, wie diese in eine konkrete Zeichenkette zu übersetzen ist. Für die drei lexikalischen Regeln `Mann_N`, `der_Det` und `schlafen_N` ist dies lediglich die entsprechende Zeichenkette z.B. für `Mann_N` „Mann“ im Deutschen bzw. „man“ im Englischen. Die restlichen Syntaxregeln sind in diesem Beispiel nur geringfügig komplexer. Die Regel `mkVP` gibt lediglich den Parameter als Rückgabewert zurück, bildet also die gleiche Zeichenkette, der bereits

[9] vgl. [Ran11] S. 55 und S. 227f.

[10] Um genau zu sein, eine Liste von Token, die bei der Linearisierung mit Leerzeichen konkateniert werden

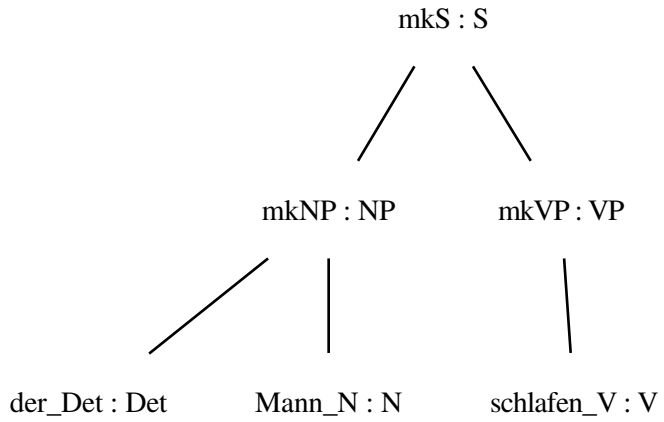

Abbildung 2.2: Baum der abstrakten Syntax für den Satz „der Mann schläft“

als Parameter übergeben wurde. Und die beiden verbleibenden Regeln `mkNP` und `mkS` konkatenieren einfach mit Hilfe des Operators `++` die beiden als Parameter übergebenen Zeichenketten.

Man kann diese sehr kurzen konkreten Grammatiken zusammen mit der gemeinsamen Grammatik in das Grammatical Framework laden und in einer der beiden Sprachen den Satz „der Mann schläft“ bzw. „the man sleeps“ parsen und die so erhaltene abstrakte Repräsentation `(mkS (mkNP der_Det Mann_N) (mkVP schlafen_V))` in die andere Sprache linearisieren, also mit Hilfe der konkreten Syntaxregeln die entsprechende Zeichenkette in der Sprache generieren.

Mit Hilfe des Grammatical Frameworks kann man sich sowohl den abstrakten Syntaxbaum und als auch den konkreten Parsebaum für eine der implementierten Sprachen grafisch darstellen lassen. Diese Bäume für die MiniSatz-Grammatik sind in Abb. 2.2 und 2.3 zu sehen.

Nun kann man diese doch sehr minimalistische Grammatik etwas erweitern, so dass man auch die Sätze „die Frauen schlafen“, „der Mann sieht die Frau“ und „der Mann liest das Buch“ erkennen kann. Die fertigen Quelltextdateien sind in Listing 2.4, 2.5 und 2.7 zu finden. Die Veränderungen in der abstrakten Syntax (Listing 2.4) betreffen hauptsächlich die Einführung von transitiven Verben mit der Kategorie `V2`. Mit der Funktion `mkVP2` wird aus einem transitiven Verb und einer Nominalphrase eine Verbalphrase mit Akkusativobjekt aufgebaut. Um auch Nominalphrasen im Plural zu ermöglichen, wird für den bestimmten Artikel eine Singular- und eine Pluralform benötigt. Auch das „Lexikon“, also die Liste der lexikalischen Einheiten, wird um die Wörter für „Frau“, „Buch“, „sehen“ und „lesen“ erweitert.

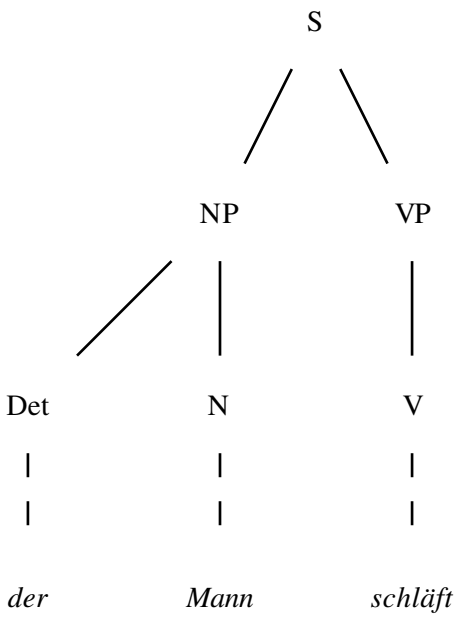

Abbildung 2.3: Parsebaum der konkreten deutschen Syntax

```
abstract SatzAbs = {
  flags startcat = S ;
  cat S ; NP ; VP ; Det ; N ; V ; V2 ;
  fun
    mkNP : Det -> N -> NP ;
    mkVP : V -> VP ;
    mkVP2 : V2 -> NP -> VP ;
    mkS : NP -> VP -> S ;
    defArtSg_Det : Det ;
    defArtPl_Det : Det ;
    Mann_N : N ;
    Frau_N : N ;
    Buch_N : N ;
    schlafen_V : V ;
    sehen_V2 : V2 ;
    lesen_V2 : V2 ;
}
```

Listing 2.4: Aus Listing 2.1 erweiterte abstrakte Syntax `SatzAbs`

```
concrete SatzGer of SatzAbs = {
  param Genus = Mask | Fem | Neutr ;
        Numerus = Sg | Pl ;
        Casus = Nom | Akk ;
  flags coding=utf8;
  lincat S = { s : Str } ;
         Det = { s : Genus => Casus => Str ; n : Numerus };
         N = { s : Numerus => Str ; g : Genus } ;
         NP = { s : Casus => Str ; n : Numerus} ;
         V,V2 = { s : Numerus => Str } ;
         VP = { s : Numerus => Str ; o : Str };
  lin
    mkNP det noun =
      { s = \\cas => det.s ! noun.g ! cas ++ noun.s ! det.n ;
                     n = det.n } ;
    mkVP v = v ** { o = "" } ;
    mkVP2 v2 np = v2 ** { o = np.s ! Akk } ;
    mkS np vp = { s = np.s ! Nom ++ vp.s ! np.n ++ vp.o } ;
    defArtSg_Det = { s = table { Mask => table {
                                   Nom => "der" ;
                                   Akk => "den"
                                   } ;
                                 Fem => table {
                                   Nom | Akk => "die"
                                   } ;
                                 Neutr => table {
                                   Nom | Akk => "das"
                                   }
                       } ;
                     n = Sg
      } ;
    defArtPl_Det = { s = \\_,_ => "die" ; n = Pl } ;
    Mann_N = { s = table { Sg => "Mann" ; Pl => "Männer" } ;
               g = Mask
      } ;
    Frau_N = { s = table { Sg => "Frau" ; Pl => "Frauen" } ;
               g = Fem
      } ;
    Buch_N = { s = table { Sg => "Buch" ; Pl => "Bücher" } ;
               g = Neutr
      } ;
    schlafen_V = {
      s = table { Sg => "schläft" ; Pl => "schlafen" }
      };
    sehen_V2 = {
      s = table { Sg => "sieht" ; Pl => "sehen" }
      };
    lesen_V2 = {
      s = table { Sg => "liest" ; Pl => "lesen" }
      };
}
```

Listing 2.5: Erweiterte konkrete deutsche Syntax `SatzGer`

In der konkreten Umsetzung sind nun aber größere Unterschiede, sowohl zur ursprünglichen Grammatik, als auch zwischen den unterschiedlichen Sprachen zu finden. Bei der konkreten deutschen Grammatik werden zuerst nach dem Schlüsselwort `param` drei neue Datentypen dadurch definiert, dass ihr Wertebereich aufgezählt wird. So wird definiert, dass im Deutschen der Genus die drei Werte Maskulin, Feminin, und Neutrum annehmen kann, der Numerus die Werte Singular und Plural und in diesem Beispiel Kasus die zwei Werte Nominativ und Akkusativ. Alle weiteren Fälle werden in diesem kleinen Beispiel nicht benötigt. Die nächste größere Änderung ist im `lincat`-Block zu finden. Denn statt allen Kategorien den selben einfachen Datentyp zu geben, haben nun alle Kategorien einen komplexen Typ. Zunächst sind all diese Typen Verbundtypen, in Englisch Records, zu erkennen an den geschweiften Klammern. Sie sammeln Objekte[11] möglicherweise verschiedenen Typs in einem einzigen Objekt zusammen. Jedes dieser inneren Objekte hat einen Typ und einen Bezeichner, über den darauf zugegriffen werden kann.[12] Auf diese Weise haben z.B. Nomen ein inhärentes Genus, gespeichert im Bezeichner `g`, und eine Form, die hier nur vom Numerus abhängig ist, gespeichert im Bezeichner `s`. Üblicherweise ist die Nomenform auch vom Kasus abhängig, allerdings betrachten wir nur die beiden Kasus Nominativ und Akkusativ, bei denen hier die Nomen immer die selbe Form bilden. Nach diesem Schema sind die Typen für alle Kategorien aufgebaut. Um die Abhängigkeit eines Wertes von gewissen anderen Werten auszudrücken, gibt es im Grammatical Framework die so genannten Tabellentypen[13], wie sie auch bei den `s`-Feldern von *Det*, *N*, *NP*, etc. zu sehen ist. Dies bedeutet, dass der Wert aus dem Bereich von Typ_2 vom Wert aus dem Bereich Typ_1 abhängt.[14]

Um es noch einmal zusammenzufassen, die Typen für die Kategorien im `licat`-Bereich sind nun statt des einfachen Typs `Str` Verbundtypen, bestehend aus mehreren Objekten, deren Werte wiederum von anderen Werten abhängig sein können. Wie das konkret funktioniert, wird im nächsten Abschnitt sichtbar. Denn es müssen sowohl für die Verbundtypen als auch für die Tabellentypen konkrete Objekte erzeugt werden. Dies geschieht im `lin`-Block, in dem nun die konkreten Grammatikregeln

[11] Der Begriff Objekt wird, obwohl es sich bei dem Grammatical Framework nicht um eine Objektorientierte Programmiersprache handelt, für Konstrukte eines bestimmten Typs verwendet

[12] Verbundtypen haben die Form $\{l_1 : T_1; \ldots; l_n : T_n\}$ mit $l_1, \ldots, l_n$ endlich viele verschiedenen Bezeichnern und $T_1, \ldots, T_n$ beliebige Typen. Objekte dieser Typen haben die Form $\{l_1 = v_1; \ldots; l_n = v_n\}$ mit v_i Werte vom Typ T_i für jedes i mit $1 \leq i \leq n$ (vgl. [Ran11] S. 279)

[13] Tabellentypen haben die Form $\{Typ_1 \texttt{ => } Typ_2\}$ mit $Typ_1, Typ, 2$ beliebige Typen und Tabellenobjekte die Form $\texttt{table \{ } k_1 \texttt{ => } V_1; \ldots; k_n \texttt{ => } V_n \texttt{ \}}$ mit k_i von Typ_1 und V_i vom Typ_2 (Tabellentypen sind im Grammatical Framework Sonderfälle von Funktionen, weswegen in dieser Arbeit nicht genauer auf die Funktionsweise eingegangen werden soll. Details sind bei [Ran11] S. 281f. zu finden)

[14] vgl. [Ran11] S. 59

„konstruiert“ werden. Fangen wir am besten von hinten, also von den lexikalischen Regeln her, an. Der Typ für Verben, ungeachtet ob es transitive, also `V2`-Verben, oder intransitive `V`-Verben sind, gibt an, dass sie aus einem einzigen Tabellenobjekt mit dem Bezeichner `s` bestehen. Diese Tabelle hat den Typ `Numerus => Str`, also eine `Str`-Zeichenkette, deren Wert von einem Numerus abhängt. Üblicherweise enthält das `s`-Feld in Grammatiken des Grammatical Frameworks das Paradigma, also eine Tabelle, die alle Wortformen enthält. So ein Verbund wird jeweils für die Verben `lesen_V2`, `sehen_V2` und `schlafen_V` konstruiert. Die geschweiften Klammern geben wieder an, dass es sich um einen Verbund handelt, der als Wert erzeugt werden soll. Dann folgt der Bezeichner `s` dem mit dem Zuweisungsoperator `=` ein Wert zugewiesen werden soll. Dieser Wert wiederum soll eine Tabelle des genannten Typs sein. Diese wird mit dem Schlüsselwort `table` erzeugt. In den darauf folgenden geschweiften Klammern muss nun jedem möglichen Wert des Datentyps, in diesem Falle `Numerus`, ein Wert des abhängigen Typs, hier `Str`, zugeordnet werden. Dazu wird der Operator `=>` verwendet. Der Typ `Numerus` hat die zwei Werte `Sg` und `Pl` und die Verben haben in dieser Grammatik zwei Formen. Verben hier nur in zwei Formen vor, der 3. Person-Präsens-Indikativ-Singular- und 3.-Person-Präsens-Indikativ-Plural-Form. So hat das Verb `lesen_V` die folgenden zwei Formen „liest“ und „lesen“.

Nach dem selben Schema funktionieren auch die restlichen Verben und die Nomen. Allerdings haben die Nomen ihr inhärentes Genus. Deshalb haben sie in ihrem Verbund zusätzlich den Bezeichner `g`, dem das grammatische Geschlecht des Nomens in der lexikalischen Regel `Mann_N`, `Frau_N` und `Buch_N` zugewiesen wird. Es ist recht offensichtlich, dass `Mann_N` maskulin, `Frau_N` feminin und `Buch_N` neutral sein sollte.

Gegenüber dem Typ der Nomen und Verben ist der Typ des Determinans, in diesem Falle der bestimmte Artikel, im Singular etwas komplizierter. Die Artikel haben ein inhärentes Numerusmerkmal, das den Numerus des Nomens in der Nominalphrase regiert. Deshalb gibt es für Singular- und Pluralartikel je eine eigene Regel.

Die Form des Artikels ist sowohl von Genus als auch von Kasus abhängig. Deshalb wird dem `s`-Feld bei `defArtSg_Det` eine Tabelle zugewiesen, in der jedem Wert für Genus erneut eine Tabelle über die Werte des Kasus zugewiesen wird. Man spricht hier von einer Tabelle von Tabellen. So wird bestimmt, dass bei maskulinem Geschlecht im Nominativ die Artikelform „der“, bei Akkusativ aber „den“ ist. Bei den anderen Genera sind für beide Kasus die Formen identisch. Dies wird durch das Zeichen `|` zwischen den Kasus ausgedrückt, das in etwa die selbe Bedeutung wie bei

regulären Ausdrücken hat. Also hat der bestimmte Artikel ungeachtet des Kasus bei Maskulina die Form „die“ und bei Neutra „das“.

Im Plural hat der bestimmte Artikel im Deutschen allerdings für jedes Genus und jeden Kasus immer die Form „die“. Deshalb kann man die Konstruktion der Tabelle von Tabellen, die beim Artikel im Singular stark vereinfachen. Zum ersten gibt es einen Platzhalter für beliebige Werte, das Zeichen `_`. In einer Tabelle kann es jeden Wert aus dem Wertebereich annehmen, für den kein eigener Eintrag in der Tabelle existiert. Ist er der einzige Eintrag in der Tabelle, so passt er für alle möglichen Werte. Wenn man also vom abstrakten Typ eine Tabelle über das Genus hat, aber jedem Genus die gleiche Zeichenkette `str` zuweisen will, so kann man `table { _ => str }` schreiben. Dies lässt sich im Grammatical Framework weiter verkürzen zu `\\_ => str`. Hat man zwei solche Tabellen ineinander geschachtelt, also `\\_ => \\_ => str`, so kann man diesen Ausdruck weiter verkürzen zu `\\_,_ => str`.[15]

Eine Tabelle dieser Form wird nun für die Form des bestimmten Artikels im Plural verwendet. Beide Determinans-Einträge haben, wie schon angedeutet, einen festen Numerus, der in einem Feld namens `n` festgehalten ist. Damit haben wir alle lexikalischen Einträge besprochen, die für unser Beispiel nötig sind.

Als nächstes folgen die syntaktischen Regeln, die aus den lexikalischen Einheiten komplexere Ausdrücke erzeugen. Beginnen wir mit der Regel `mkNP`, die aus einem Objekt `det` des Typs `Det` und einem Objekt `nom` des Typs `N` ein Objekt des Typs `NP` erzeugt. `NP`s, also Nominalphrasen, haben einen festen Numerus, der vom Artikel her stammt, und die linearisierte Form der Nominalphrase hängt von Kasus ab. Deshalb muss das *s*-Feld wieder den Tabellentyp `Genus => Str` haben. Dafür wird wieder eine Tabelle generiert, allerdings erneut in einer Variation der Kurzform, in der der Platzhalter (`_`) durch die Variable `cas` ersetzt wird, so dass der Wert, mit dem ein Eintrag aus der Tabelle ausgewählt werden soll, in dieser Variable verfügbar bleibt und für weitere Operationen verwendet werden kann.

Der Operator, um aus einem Objekt eines Tabellentyps einen Wert zu wählen, ist das Ausrufezeichen (`!`). Um auf die verschiedenen Felder in Verbundtypen zuzugreifen, wird der Punkt-Operator (`.`) verwendet. So liefert `det.n` den Numerus des in der Variable det gespeicherten Artikels. Und zusammen mit dem Selektionsoperator liefert der Ausdruck `noun.s ! det.s` den Wert aus der Tabelle im `s`-Feld, der durch diesen Numerus ausgewählt werden kann, also die konkrete Wortform von Typ `Str`. Um aus einem `Det`-Objekt einen `Str`-Wert zu erhalten, müssen wir zwei-

[15] vgl. [Ran11] S. 282

mal Werte aus Tabellen auswählen, zuerst ein Genus und anschließend einen Kasus. Dazu können wir zweimal den `!`-Operator verwenden. Das Geschlecht haben wir bereits im `g`-Feld des Nomens und der gewünschte Kasus ist in der Tabellenvariable `cas` gespeichert. Also bekommen wir per `det.s ! noun.g ! cas` einen Wert vom Typ `Str` und diese beiden String-Werte können nun wieder verkettet werden. Dieser zusammengesetzte Wert wird schließlich in die Tabelle eingefügt. Diese kompakte Tabelle ist die Kurzform für die ausführlichere Tabelle in Listing 2.6.

```
1 table {
2   Nom => det.s ! noun.g ! Nom ++ noun.s ! det.n ;
3   Akk => det.s ! noun.g ! Akk ++ noun.s ! det.n
4 }
```

Listing 2.6: Ausführliche Form der Tabelle in Zeile 13 des Listings 2.5

Verbalphrasen haben nahezu den gleichen Typ wie die Verben `V` und `V2`. Sie haben lediglich ein zusätzliches Feld für ein mögliches Akkusativobjekt bei transitiven Verben. Deshalb ist die Regel `mkVP` für die Konstruktion von Verbalphrasen aus intransitiven Verben sehr einfach. Sie übernimmt den Wert des Verbs und erweitert lediglich den Verbund um das `o`-Feld mit einer leeren Zeichenkette, denn bei intransitiven Verben ist kein Akkusativobjekt vorhanden. In der Regel `mkVP2` für transitive Verben mit Akkusativobjekt wird das `o`-Feld mit dem Wert des Objekts im Akkusativ gefüllt. Dazu wird aus der als Parameter übergebenen Nominalphrase der für den Akkusativ passende Wert ausgewählt. Die letzte Regel `mkS` konstruiert schließlich aus einer Nominalphrase und einer Verbalphrase einen Satz, in diesem Fall eine einfache Zeichenkette. Dazu wird aus der Subjekt-Nominalphrase der Nominativwert gewählt. Dieser Wert wird mit der Verbform verkettet, die dem Numerus des Subjekts entspricht, so wie mit dem Wert im Objektfeld. Da dieses Feld bei intransitiven Verben die leere Zeichenkette enthält, entfällt in diesem Falle im Endresultat das Akkusativobjekt. Mit dieser schon nicht mehr ganz einfachen Grammatik können nun die gewünschten Sätze erkannt werden.

Die entsprechende englische Grammatik ist etwas einfacher. Der Hauptgrund dafür ist, dass weder Kasus noch Genus eine Rolle spielen. Dadurch entfallen alle `g`-Felder bei Nomen und alle Genus- und Kasus-abhängige Tabellen. Übrig bleiben der Numerus als Merkmal und somit bei den Nomen und Verben die Tabellen für die Singular- und Pluralformen. Die Artikel haben weiterhin einen festen Numerus, aber das `s`-Feld besteht, nachdem alle auf Genus- und Kasus-Werten basierenden Tabellen wegfallen, nur noch aus einer einzigen Zeichenkette. Auch die syntaktischen

```
concrete SatzEng of SatzAbs = {
  param Numerus = Sg | Pl ;
  lincat S = { s : Str } ;
         NP = { s : Str ; n : Numerus } ;
         VP = {s : Numerus => Str ; o : Str } ;
         Det = { s : Str ; n : Numerus } ;
         N = { s : Numerus => Str } ;
         V,V2 = { s: Numerus => Str };
  lin
    mkNP det noun = { s = det.s ++ noun.s ! det.n ;
                      n = det.n } ;
    mkVP v = v ** { o = "" } ;
    mkVP2 v2 np = v2 ** { o = np.s } ;
    mkS np vp = { s = np.s ++ vp.s ! np.n ++ vp.o } ;
    defArtSg_Det = { s = "the" ; n = Sg } ;
    defArtPl_Det = { s = "the" ; n = Pl } ;
    Mann_N = {
      s = table { Sg => "man" ; Pl => "men" } } ;
    Frau_N = {
      s = table { Sg => "woman" ; Pl => "women" } } ;
    Buch_N = {
      s = table { Sg => "book" ; Pl => "books" } } ;
    schlafen_V = {
      s = table { Sg => "sleeps" ; Pl => "sleep" } } ;
    sehen_V2 = {
      s = table { Sg => "sees" ; Pl => "see" } } ;
    lesen_V2 = {
      s = table { Sg => "reads" ; Pl => "read" } } ;
}
```

Listing 2.7: Erweiterte konkrete englische Syntax `SatzEng`

Regeln sind hier viel einfacher. So entfällt auch in der `mkNP`-Regel die Tabelle und der Wert des Artikels wird einfach vor den Wert des Nomens unter Berücksichtigung des Numerus gehängt. Der Numerus wird weiterhin aus dem Artikel übernommen. Die `mkVP`-Regel bleibt komplett unverändert, bei der Regel `mkVP2` entfällt der Akkusativ zur Auswahl der Objekt-Zeichenkette. Ebenso entfällt der Nominativ in der `mkS`-Regel. Mit dieser Grammatik können nun auch die englischen Formen der gewünschten Sätze erkannt werden.

An diesem Beispiel kann man nun deutlicher die Vorteile der Trennung in abstrakte und konkrete Syntax sehen. Denn obwohl beide konkreten Grammatiken dieselbe abstrakte Syntax implementieren, so gibt es doch große Unterschiede in den zu berücksichtigenden Merkmalen.

Der gesamte Sprachumfang der Grammatiksprache im Grammatical Framework ist noch etwas umfangreicher, allerdings sollten nach diesen Beispielen die wichtigsten Sprachkonstrukte verständlich sein. Falls weitere Informationen benötigt werden, so bietet [Ran11] im Anhang C eine vollständige Sprachreferenz. Des weiteren bietet die offiziellen Webseite sowohl eine Kurzreferenz[16] als auch eine etwas ausführlichere Sprachbeschreibung[17]. Diese Ressourcen sind zwar hilfreich, allerdings ändert sich die Sprache in einem gewissen Rahmen schneller weiter, als die Dokumentation aktualisiert wird. So wurde im Juni 2013 mit dem`Maybe`-Typ eine neue Art von Datentypen eingeführt, die es ermöglicht in einem Paradigma fehlende Werte zu markieren. Dieser Datentyp ist in den genannten Quellen allerdings noch nicht dokumentiert.

[16] `http://www.grammaticalframework.org/doc/gf-reference.html`
[17] `http://www.grammaticalframework.org/doc/gf-refman.html`

2.1.2 Die Ressource Grammar Library

Was für allgemeine Programmiersprachen eine Standardbibliothek ist, ist im Grammatical Framework für die Multilingualität die Ressource Grammar Library, kurz RGL. Sie definiert eine gemeinsame abstrakte Syntax, die für verschiedenen Sprachen implementiert werden kann. Für mehr als 30 natürliche Sprachen ist diese abstrakte Syntax, mehr oder weniger komplett, umgesetzt.[18] Mit Hilfe dieser abstrakten Syntax und den konkreten Implementierungen ist eine grundlegende Übersetzung zwischen den unterstützten Sprachen direkt nach der Installation möglich, allerdings nur im Bereich des von Haus aus gegebenen Grundvokabulars. Meist muss also für eine eigene Anwendung mindestens das nötige Vokabular hinzugefügt werden, um die Möglichkeiten dieser Bibliothek ausschöpfen zu können.

Zunächst einmal bietet die Ressource Grammar Library die Definition verschiedener, so genannter geschlossener Wortkategorien, also Kategorien, deren Werte man zumindest theoretisch komplett aufzählen kann. Dazu gehören bestimmte Adverbien, Konjunktionen, Pronomen, Präpositionen und Subjunktionen. Weiterhin gibt es offene Wortkategorien, wie verschiedene Arten von Adverbien und verschiedenstellige Adjektive, Nomen und Verben. Darauf aufbauend gibt es Phrasenkategorien wie Verbalphrasen, Nominalphrasen, Sätze, Relativsätze, etc. und die nötigen Syntaxregeln um diese zu erzeugen. Insgesamt gibt es ca. 42 geschlossene Wort- und Phrasenkategorien und 22 offene Kategorien in der Ressource Grammar Library. Eine Übersicht über den Umfang der Ressource Grammar Library findet man in [Ran11] Anhang D so wie online[19].

Mit Hilfe der in der Ressource Grammar Library vorhandenen Grammatikregeln und Kategorien kann unser bereits gezeigtes Grammatikfragment noch einmal erheblich optimiert werden. Dabei ändert sich kaum etwas an der abstrakten Syntax der Grammatik. Aber bei den konkreten Umsetzungen reduziert sich der nötige Entwicklungsaufwand erheblich. Denn durch den höheren Abstraktionsgrad muss man sich über vieles keine Gedanken mehr machen, z.B. über die interne Struktur der Kategorien oder die Übereinstimmung zwischen den Merkmalen. Auch die Details der Wortbildung werden größtenteils ausgeblendet. Offensichtlich ist also nicht mehr so viel linguistisches Wissen nötig um mit Hilfe der Ressource Grammar Library natürlichsprachliche und vor allem multilinguale Anwendungen zu konstruieren.

Bei der abstrakten Syntax in Listing 2.8 gibt es nur eine kleine Änderung. Die

[18] Für den Stand verschiedener Sprachen vgl. `http://www.grammaticalframework.org/lib/doc/status.html`

[19] `http://www.grammaticalframework.org/lib/doc/synopsis.html`

```
1 --# -path=.:alltenses:prelude
2 abstract RglSatzAbs = {
3   flags startcat = S ;
4   cat S ; NP ; VP ; Det ; N ; V ; V2 ;
5   fun
6     mkNP : Det -> N -> NP ;
7     mkVP : V -> VP ;
8     mkVP2 : V2 -> NP -> VP ;
9     mkS : NP -> VP -> S ;
10    defArtSg_Det : Det ;
11    defArtPl_Det : Det ;
12    Mann_N : N ;
13    Frau_N : N ;
14    Buch_N : N ;
15    schlafen_V : V ;
16    sehen_V2 : V2 ;
17    lesen_V2 : V2 ;
18 }
```

Listing 2.8: Abstrakte Syntax mit Hilfe der RGL

darauf aufbauenden konkreten Grammatiken in Listing 2.9 und 2.10 sind diesmal einander sehr ähnlich.

In der ersten Zeile findet man nun das neue Schlüsselwort `open`. Dieses `open`, ein Modulname des gleichen Typs, wie die einbindende Datei, und ein `in` vor einem Codeblock, ist eine von zwei Möglichkeiten der Wiederverwendung von Modulen im Grammatical Framework. Die andere ist an der selben Stelle ein Modulname gefolgt von `**` und einem Codeblock.[20] Bei der deutschen Grammatik wird das konkrete Modul `SyntaxGer` geladen. Darin enthalten sind die syntaktischen Konstruktionsregeln für Phrasen. Zusätzlich wird das konkrete Modul `ParadigmsGer` eingebunden, das die Mittel bereitstellt, lexikalische Objekte für die Sprache korrekt zu erzeugen. Die Typen für alle Kategorien im `lincat`-Bereich werden einfach aus dem `SyntaxGer`-Modul übernommen.

Bei den lexikalischen Regeln werden zum einen Regeln aus `ParadigmsGer` benutzt, um die lexikalischen Objekte der verschiedenen Kategorien zu erzeugen. So z.B. die Funktion `mkN`, die aus der Nominativ-Singular- und der Nominativ-Plural-Form sowie dem Genus ein Nomen-Objekt erstellt.

Die Funktion `mkV` erstellt aus fünf Verbformen ein Verb-Objekt, welches dann das ganze Paradigma beinhaltet. Und die Funktion `mkV2` erzeugt dann aus einem

[20] vgl. [Ran11] S. 264f.

```
1 --# -path=.:alltenses:prelude
2 concrete RglSatzGer of RglSatzAbs =
3 open SyntaxGer, ParadigmsGer in {
4   flags coding=utf8;
5   lincat
6     S = SyntaxGer.S ;
7     NP = SyntaxGer.NP ;
8     VP = SyntaxGer.VP ;
9     V = SyntaxGer.V ;
10    V2 = SyntaxGer.V2 ;
11    N = SyntaxGer.N ;
12    Det = SyntaxGer.Det ;
13  lin
14    mkS np vp = SyntaxGer.mkS
15      presentTense simultaneousAnt positivePol
16      (mkCl np vp) ;
17    mkNP det n = SyntaxGer.mkNP det n ;
18    mkVP v = SyntaxGer.mkVP v ;
19    mkVP2 v2 np = SyntaxGer.mkVP v2 np ;
20    Mann_N = mkN "Mann" "Männer" masculine ;
21    Frau_N = mkN "Frau" "Frauen" feminine ;
22    Buch_N = mkN "Buch" "Bücher" neuter;
23    schlafen_V =
24      mkV "schlafen" "schläft" "schlief"
25        "schliefe" "geschlafen" ;
26    sehen_V2 =
27      mkV2 ( mkV "sehen" "sieht" "sah" "sähe" "gesehen" ) ;
28    lesen_V2 =
29      mkV2 ( mkV "lesen" "liest" "las" "läse" "gelesen" ) ;
30    defArtSg_Det = theSg_Det ;
31    defArtPl_Det = thePl_Det ;
32 }
```

Listing 2.9: Konkrete deutsche Syntax mit Hilfe der RGL

intransitiven Verb-Objekt ein transitives Verb.

Des weiteren sind einige Objekte geschlossener Kategorien bereits vordefiniert, so wie hier der bestimmte Artikel im Singular (`theSg_Det`) und Plural (`thePl_Det`).

Auch bei den syntaktischen Regeln kann man auf Bibliotheksfunktionen zurückgreifen. So gibt es in der Ressource Grammar Library bereits die Funktionen `mkVP` und `mkNP`, die genau das erledigen, was wir in unseren Funktionen `mkVP`, `mkVP2` und `mkNP` haben wollen. Lediglich die Funktion `mkS` der Ressource Grammar Library verhält sich etwas anders als unsere `mkS`-Regel, denn sie benötigt als zusätzliche Parameter ein Tempus, also Präsens, Präteritum oder Futur, Information über die Zeitigkeit, also ob der Vorgang bereits abgeschlossen ist oder noch andauert, und eine Polarität, also ob der Satz verneint ist oder nicht. Da wir nur positive, gleichzeitige Präsenssätze behandeln, können wir diese Parameter so festsetzen, dass genau diese Sätze gebildet werden. Die Regel `mkCl`, die im Rumpf der Regel `mkS` auftritt, erledigt fast, was die Regel `mkS` in der `MiniSatz`- und `Satz`-Grammatik übernommen hat, sie kombiniert eine Nominalphrase und eine Verbalphrase zu einem Satz, in diesem Fall von Typ `Cl`. Sätze dieses Typs unterscheiden sich von Sätzen des Typs `S` lediglich dadurch, dass die oben genannten Parameter zu den Tempusmerkmalen und Polarität noch nicht festgelegt sind.

In der englischen Grammatik sind die Unterschiede diesmal sehr gering und beziehen sich lediglich auf die Konstruktion der lexikalischen Objekte. Die Nomen können teilweise aus einer einzigen Form gebildet werden, wie bei `Buch_N`. Dies gilt genauso für die Verben hier, denn die Anzahl der nötigen Formen ist abhängig von der Regelmäßigkeit der Formenbildung, und die englische Sprache ist etwas regelmäßiger als das Deutsche. Der Rest der Grammatik ist identisch, abgesehen davon, dass natürlich die englische Version des konkreten `Syntax`- und `Paradigms`-Moduls benutzt wird.

Es ist offensichtlich, dass Grammatiken selbst für unterschiedliche Sprachen sehr ähnlich werden, wenn man zu ihrer Implementierung die Ressource Grammar Library benutzt, da man mit Hilfe der Ressource Grammar Library auf einer höheren Abstraktionsebene arbeiten kann.

```
1 --# -path=.:alltenses:prelude
2 concrete RglSatzEng of RglSatzAbs =
3 open SyntaxEng,ParadigmsEng in {
4   flags coding=utf8;
5   lincat
6     S = SyntaxEng.S ;
7     NP = SyntaxEng.NP ;
8     VP = SyntaxEng.VP ;
9     V = SyntaxEng.V ;
10    V2 = SyntaxEng.V2 ;
11    N = SyntaxEng.N ;
12    Det = SyntaxEng.Det ;
13  lin
14    mkS np vp = SyntaxEng.mkS
15      presentTense simultaneousAnt positivePol
16      (mkCl np vp) ;
17    mkNP det n = SyntaxEng.mkNP det n ;
18    mkVP v = SyntaxEng.mkVP v ;
19    mkVP2 v2 np = SyntaxEng.mkVP v2 np ;
20    Mann_N = mkN "man" "men" ;
21    Frau_N = mkN "woman" "woman" ;
22    Buch_N = mkN "book" ;
23    schlafen_V = mkV "sleep" ;
24    sehen_V2 = mkV2 "see" ;
25    lesen_V2 = mkV2 "read" ;
26    defArtSg_Det = theSg_Det ;
27    defArtPl_Det = thePl_Det ;
28 }
```

Listing 2.10: Konkrete englische Syntax mit Hilfe der RGL

2.2 Die Lateinische Sprache

2.2.1 Sprachwissenschaftliche Einordnung

Die lateinische Sprache, auch als oskisch-umbrische Sprache bezeichnet, gehört zur indogermanischen Sprachfamilie und dort zur Unterfamilie der italischen Sprachen. Durch diese Verwandtschaft kann man bei Wörtern und Wortformen oft Entsprechungen zwischen der lateinischen Sprache und verschiedensten anderen Sprachen Westeuropas bis hin zu Mittelasien finden (vgl. Tabelle 2.1).[21] Entstanden ist es als

lateinisch	altgriechisch	deutsch
pater	$\pi\alpha\tau\acute{\eta}\rho$ (=patēr)	Vater
ager	$\alpha\gamma\rho\acute{o}\varsigma$ (=agrós)	Acker
trēs	$\tau\rho\varepsilon\tilde{\imath}\varsigma$ (=treĩs)	drei
decem	$\delta\acute{\varepsilon}\kappa\alpha$ (=déka)	zehn

Tabelle 2.1: Wortentsprechungen in verschiedenen indogermanischen Sprachen (vgl. [BL94] S.1)

ein in der Stadt Rom üblicher Dialekt parallel zu anderen ländlicheren Dialekten im Latium, einer Region in Mittelitalien. Im Laufe der Zeit verdrängte es jedoch die weiteren italischen Sprachen im Zuge der Ausdehnung des römischen Reichs.[22]

Die Sprachgeschichte kann in mehrere Epochen unterteilt werden. Üblicherweise beginnt man diese Einordnung mit der Epoche des Altlateins, das von ca. 240 v. Chr. bis 80 v. Chr. angesiedelt wird. Es reicht von den frühesten nachgewiesenen lateinischen Sprachzeugnissen bis zum Beginn der Zeit des klassischen Lateins. Dessen Zeitraum wird von ca. 80 v. Chr. bis 117. n. Chr. gerechnet und beginnt in etwa mit den ersten öffentlichen Auftritten des M. Tullius Cicero. Die bekannten Gerichtsreden des berühmten römischen Anwalts und Schriftstellers von ca. 80 v. Chr. sind noch größtenteils erhalten. Die nach-klassische Phase kann wiederum in verschiedene Epochen unterteilt werden, in denen unter anderem die romanischen Volkssprachen entstanden sind, bis hin zum so genannten Neulatein, das vom 15. Jahrhundert bis hin zum Beginn des 20. Jahrhundert die Sprache der Wissenschaft darstellte und noch immer großen Einfluss auf Begriffe des Alltags ausübt.[23]

Auch heute noch am bedeutendsten ist wohl das klassische Latein, das weiterhin in Schulen unterrichtet wird und sich vor allem mit seinem großen überlieferten Textkorpus hervorhebt. Da sich die meisten Lateingrammatiken auf diese Sprachepoche

[21] vgl. [BL94] S.1
[22] vgl. [Glu04] Lateinisch: S. 5359
[23] vgl. [Mul06] S. 27ff.

stützen, wird in dieser Arbeit primär das klassische Latein betrachtet.

In der Sprachwissenschaft ist auch weiterhin umstritten, in welchem Verhältnis das klassische Latein zum so genannten Vulgärlatein steht. Heutzutage geht man davon aus, dass das klassische Latein eine kaum wirklich gesprochene Sprache war und das Vulgärlatein nicht nur eine nach-klassische Sprachvariante ist, sondern bereits parallel zum klassischen Schriftlatein als gesprochene Sprache verwendet wurde. Allerdings fand das klassische Latein noch bis in das 5. Jahrhundert n. Chr. Verwendung als eine Art Schreibnorm, während sich das Vulgärlatein langsam zu den romanischen Sprachen weiterentwickelte.[24]

Formal gehört Latein zu den stark flektierenden Sprachen. Das heißt, dass in der lateinischen Sprache, wie für synthetische Sprachen üblich, syntaktische Klassen und Verhältnisse über Wortsuffixe ausgedrückt werden.[25]. Allerdings drücken bei flektierenden Sprachen, im Gegensatz zu agglutinierenden Sprachen, die Affixe meist mehr als ein grammatisches Merkmal aus.[26] So ist bei der Verbform *audio* das *audi* der Verbstamm, um genau zu sein der Präsensstamm, des Verbs *audire* und das Suffix *-o* kodiert folgende Merkmale: 1. Person, Singular, Präsens, Indikativ, Aktiv.[27]

Es gibt für Nomen fünf zum Teil genusbasierte Flexionsklassen, also verschiedene Typen der Flexion innerhalb einer Wortart,[28], sechs verschiedene Kasus (Nominativ, Genitiv, Dativ, Akkusativ, Ablativ und Vokativ) und drei Genera (Maskulin, Feminin, Neutrum). Des weiteren gibt es ein voll flektierendes Pronomensystem und vier relativ stark synthetische Flexionsklassen für Verben.[29] Zu den Kasus sei anzumerken, dass der Ablativ im Lateinischen ein eigenständiger Kasus ist, jedoch der Vokativ oft mit dem Nominativ zusammenfällt.[30]

Die Wortstellung des Lateinischen wird oft als sehr frei beschrieben, allerdings gibt es eine klare Präferenz der SOV-Wortstellung im Satz, also dass das Objekt des Satzes direkt auf das Subjekt folgt, und das Verb den Satz abschließt. Die Möglichkeiten zur Positionierung des Adjektivs im Bezug auf das Nomen sind allerdings durch nichts beschränkt.[31]

[24] vgl. [Glu04] Lateinisch: S. 5359 und Vulgärlatein: S. 10719
[25] vgl. [Glu04] Synthetisch: S. 9690
[26] vgl. [Glu04] Flektierende Sprache: S. 3009
[27] vgl. [BL94] S. 75
[28] vgl. [Glu04] Flexion: S. 3011
[29] [Glu04] Lateinisch: S. 5359
[30] vgl. [BL94] S. 20f.
[31] [Glu04] Lateinisch: S. 5359

2.2.2 Bedeutung in der heutigen Zeit

Man kann sich natürlich über die Notwendigkeit streiten, sich in der heutigen Zeit noch mit der lateinischen Sprache zu beschäftigen. Es gibt aber auch ziemlich gute Gründe dafür, Latein nicht einfach als tote Sprache abzustempeln und nicht weiter zu betrachten.

Der am häufigsten, vor allem im Schulalter bei der Wahl einer zu lernenden Fremdsprache, vorgebrachte Grund ist, dass die lateinische Sprache als „Mutter aller romanischen Sprachen“ später einen einfacheren Einstieg in das Erlernen z.B von Französisch oder Spanisch bietet. Auch galt Latein seit Jahrhunderten, und gilt weiterhin, als produktive Quelle für Fachbegriffe aus Wissenschaft, Forschung und Technik. So haben viele moderne Begriffe wie Computer[32] und Monitor[33] lateinische Wurzeln. Auch im Universitätsalltag wird man oft mit lateinischen Lehnwörtern konfrontiert. Man trifft sich zum Essen in der Mensa[34] und studiert an Fakultäten[35].

Vor allem in der Sprachwissenschaft hat die lateinische Sprache eine besondere Bedeutung, da sie bei einem Vergleich verschiedener indogermanischer Sprachen als eine Art „default“-Sprache angesehen werden kann, denn sie bietet fast alle grammatischen Kategorien, die gewöhnlich benötigt werden. So kann Latein als Vergleichsparameter (*tertium comparationis*) verwendet werden. Diese Stellung der lateinischen Sprache spiegelt sich auch in der Fachterminologie moderner Schulgrammatiken wieder, die fast ausschließlich von lateinischen Fachausdrücken geprägt ist.[36]

Als etwas ungewöhnliche aber noch relativ moderne Verwendung der lateinischen Sprache kann in *latino sine flexione* gesehen werden. Diese von Giuseppe Peano, Anfang des 20. Jahrhunderts als Welthilfssprache entwickelte, vereinfachte Form der lateinischen Sprache fand bis ca. 1950 in mehreren wissenschaftlichen Veröffentlichungen Verwendung. Sie basiert auf dem üblichen lateinischen Wortschatz, der auch durch modernes romanisches Vokabular erweitert werden kann, und einer stark vereinfachten Morphologie.[37]

[32] von lat. *computere* - berechnen
[33] von lat. *monere* - mahnen
[34] von lat. *mensa* - Tisch, Tafel
[35] von lat. *facultas* - Vermögen, Fähigkeit
[36] vgl. [Mul06] S. 10
[37] vgl. [Glu04] Latino sine flexione S. 5374

3 Grammatikerstellung

Nach der Einführung in die Grundlagen in Kapitel 2, die nötig sind um im Grammatical Framework eine Grammatik zu entwickeln, folgt nun eine Schilderung der konkreten Schritte die angewendet wurden, um diese Lateingrammatik zu entwickeln.

Es sei noch anzumerken, dass es bereits einen Versuch von Aarne Ranta, einem der Autoren des Grammatical Frameworks, gab, eine Lateingrammatik für die Ressource Grammar Library des Grammatical Frameworks zu entwickeln. Die vorliegende Arbeit baut auf der Arbeit Rantas auf, kann aber insofern als selbständige und vollwertige Arbeit angesehen werden, da Rantas Implementierung sehr rudimentär und noch nicht funktionstüchtig war und seit ca. 2005 nicht mehr weiterentwickelt wurde.

Die Gliederung dieses Kapitels folgt dem gewählten Vorgehen bei der Implementierung. Die Begründung für die Reihenfolge der einzelnen Schritte wird jeweils zu Beginn der einzelnen Kapitel kurz dargelegt. In diesem Sinne wird zunächst die Erstellung des Lexikons, als zweites die Umsetzung der lateinischen Morphologie und schließlich die Syntaxregeln in dieser Lateingrammatik beschrieben.

3.1 Lexikon

Den Beginn dieser Grammatikimplementierung bildete die Erstellung des minimal nötigen Lexikons. Durch die abstrakte Syntax der Ressource Grammar Library für Lexika[1] ist eine Liste von etwas über 450 englischen Bezeichnern für Worte vorgegeben, die in jeder Sprache umgesetzt werden sollten.

Für die Erstellung eines Lexikon, wie es in einer Grammatik verwendet werden kann, sind zwei Schritte nötig. Einerseits müssen für jeden vorgegebenen Bezeichner, in diesem Falle alle lexikalischen Regeln aus dem abstrakten Lexikon, die entsprechenden Zeichenketten der gewünschten Sprache zugeordnet werden. Andererseits muss das Lexikon auch all jene Informationen enthalten, die zum Bilden der Vollformen und zur Konstruktion grammatischer Einheiten nötig sind. So z.B. bei Nomen das Geschlecht, wenn es nicht von der Grundform abgeleitet werden kann.

Der erste Schritt ist also, einfach das Lexikon einer anderen Sprache, in diesem Falle Englisch, zu kopieren. Normalerweise ist es vernünftig, mit einer Sprache zu beginnen, die der zu implementierenden Sprache möglichst nahe steht.[2]. Allerdings wurde dieser Schritt bereits von Aarne Ranta dadurch begonnen die englischen lexikalischen Ressourcen zu kopieren und anzupassen. Anschließend werden zunächst alle Zeichenketten durch mögliche lateinische Entsprechungen ersetzt. Um eine möglichst exakte Übersetzung für die verschiedenen Lexikoneinträge zu finden, mussten teilweise verschiedene Vorgehensweisen bemüht werden. Hauptsächlich wurden hier, soweit möglich, gedruckte Wörterbücher für die Übersetzung verwendet, gelegentlich waren aber auch Onlineresourcen unumgänglich.

Des weiteren wird der Übersetzungsschritt von den englischen Bezeichnern zu deutschen Entsprechungen, die für die weitere Übersetzung in das Lateinische verwendet wurden, nicht genauer erläutert. Es sei nur so viel gesagt, dass die Bedeutung der meisten Bezeichner ohne weitere Hilfsmittel ersichtlich ist. In den Fällen, in denen Unklarheiten herrschten, wurde ein bekanntes Onlinewörterbuch[3] zur Klärung verwendet.

3.1.1 Wörterbücher

Um eine passende lateinische Übersetzung für die Lexikoneinträge zu finden, wurde primär der deutsch-lateinische Teil eines handelsüblichen Schulwörterbuchs ([PL81]) verwendet, soweit ein entsprechender Eintrag in diesem Wörterbuch zu finden war.

[1] vgl. **Lexicon.gf** und **Structural.gf** im Ordner **lib/src/abstract**
[2] vgl. [Ran11] S. 224f.
[3] `http://dict.leo.org/`

Allerdings gibt es bereits an diesem Punkt diverse Herausforderungen. Denn eine Art von Wörtern, die allgemein zu Problemen bei der Übersetzung, und somit auch bei der Erstellung dieses Lexikons, führten, sind Wörter mit ambiger Bedeutung, oder auch homonyme Begriffe, wie das häufig als Beispiel angeführte Wort „Bank“ bzw. „bank“, das in vielen Sprachen mehrere verschiedene Bedeutungen haben kann, z.B. im Deutschen als Sitzgelegenheit und als Geldinstitut oder im Englischen als Geldinstitut oder als Ufer eines Flusses.[4] Für diesen und ähnliche Begriffe wurde willkürlich eine der plausiblen Bedeutungen gewählt, da keine Hinweise zur gewünschten Bedeutung in der Grammar Library gefunden werden konnten. Die Entscheidung eine einzige Bedeutung zu wählen, und nicht verschiedene Bedeutungen als Varianten des Wortes zu implementieren, muss getroffen werden um die Anzahl der möglichen Übersetzungen eines Ausdrucks möglichst gering zu halten. Für den Umgang mit ambigen Wörtern in einem Lexikon für das Grammatical Framework gibt es keine klaren Regeln, die angebrachteste Methode scheint aber zu sein, für jede Bedeutung einen eigenen Bezeichner zu wählen. So wäre möglicherweise in einem Lexikon `bank1_N` die Sitzgelegenheit und würde im englischen mit „bench“ übersetzt und `bank2_N` das Geldinstitut, das mit „bank“ übersetzt würde.

Ein weiteres Problem bei einer so alten Sprache wie Latein ist, dass bei vielen, meist moderneren Begriffen, nicht immer entsprechende Wörterbucheinträge gefunden werden können. Zwar gibt es auch andere Wörterbücher, wie das Schulwörterbuch von PONS ([DFV12]), das einen umfangreicheren deutsch-lateinischen Teil enthält. Es deckt mehr moderne Begriffe ab, allerdings gibt es auch dort Begriffe, für die kein Eintrag zu finden ist. Für diesen Fall müssen neben den bewährten gedruckten Wörterbüchern auch andere Quellen, vor allem Onlinequellen zu Rate gezogen werden. Einige davon werden im Folgenden kurz gezeigt.

3.1.2 Onlinequellen

Als mögliche Lösung bei der Suche nach Übersetzungen, die im Wörterbuch nicht zu finden sind, bietet sich die Nutzung von, meist kollaborativen, Internetquellen an. Eine der interessantesten Quellen für moderne Begriffe aus dem Bereich der Substantive ist wohl die lateinische Wikipedia[5]. Obwohl Latein als tote Sprache gilt, existieren dort über 90000 lateinische Artikel[6], die von einer recht lebendigen Gemeinschaft gepflegt werden. Natürlich muss man immer bedenken, dass es keine

[4] vgl. [Glu04] Homonymie: S. 3927

[5] `http://la.wikipedia.org/wiki/Pagina_prima`

[6] `http://la.wikipedia.org/wiki/Specialis:Census`; Stand: 30.7.2013

Garantie für die Qualität von kollaborativen Onlinequellen gibt. Allerdings hat sich das Prinzip der Wikipedia ja auch in anderen Sprachen bewährt, wenn auch die Qualitätssicherung durch manuelle Korrekturen, und damit auch die Qualität der einzelnen Artikel, direkt mit der Größe der an dem Projekt arbeitenden Community zusammenhängt. Neben der Wikipedia, die vom Konzept her eigentlich eine allgemeine Enzyklopädie ist, und nur im Nebeneffekt linguistische Ressourcen zur Verfügung stellt, gibt es noch weitere Internetquellen, die bei der Erstellung eines Lexikons helfen können. So gibt es das deutsche Lateinportal Auxilium-online.net[7], das englischsprachige Wiktionary[8] und die Lateinressourcen bei der Perseus Digital Library[9].

Auxilium-online.net bezeichnet sich selbst als das größte deutschsprachige Lateinportal im Internet und bietet ein kostenloses Onlinewörterbuch, sowohl in der Richtung Lateinisch-Deutsch als auch in umgekehrter Richtung, das von registrierten Benutzern erweitert und korrigiert werden kann. Allerdings liegt bei diesem Wörterbuch der Schwerpunkt auch eher auf dem klassischen Vokabular.

Das englischsprachige Wiktionary hilft zwar nicht direkt bei der Suche nach einer Übersetzung aus einer anderen Sprache, es bietet aber für ein umfangreiches Vokabular sowohl eine morphologische Analyse für viele Wortformen als auch detaillierte Informationen über Verwendung und Formenbildung für lateinische Vokabeln.

Die Perseus Digital Library und vor allem die darin enthaltenen Wörterbücher fallen eher in die Kategorie klassischer, gedruckter Wörterbücher, was primär daher rührt, dass diese Wörterbücher Digitalisate seit Jahrzehnten bewährter Wörterbücher sind.[10] Jedoch bietet Perseus die Möglichkeit einer erweiterten Suchfunktion sowie einer Angabe zur Wortfrequenz im verfügbaren Korpus.

Eine der Onlinequellen für moderne lateinische Begriffe, die offizielle Liste des Vatikans zur Übersetzung moderner Alltagsbegriffe[11], wurde nicht verwendet, da sie nur zwischen Latein und Italienisch übersetzt. Dies würde aus verschiedenen Gründen zu Problemen führen.

---

[7] `http://www.auxilium-online.net/`

[8] `http://en.wiktionary.org/`

[9] `http://www.perseus.tufts.edu/hopper/`

[10] *A Latin Dictionary. Founded on Andrews' edition of Freund's Latin dictionary. revised, enlarged, and in great part rewritten by. Charlton T. Lewis, Ph.D. and. Charles Short, LL.D. Oxford. Clarendon Press. 1879.* (`http://www.perseus.tufts.edu/hopper/text?doc=Perseus%3atext%3a1999.04.0059`) und *Lewis, Charlton, T. An Elementary Latin Dictionary. New York, Cincinnati, and Chicago. American Book Company. 1890.* (`http://www.perseus.tufts.edu/hopper/text?doc=Perseus%3atext%3a1999.04.0060`)

[11] `http://www.vatican.va/roman_curia/institutions_connected/latinitas/documents/rc_latinitas_20040601_lexicon_it.html`

3.1.3 Geschlossene Kategorien

```
1  param
2    PronReflForm = PronRefl | PronNonRefl ;
3    PronDropForm = PronDrop | PronNonDrop;
4    Agr = Ag Gender Number Case ;
5  oper
6    Determiner : Type =
7      { s : Gender => Case => Str ; n : Number } ;
8    Preposition : Type = {s : Str ; c : Case} ;
9    NounPhrase : Type =
10     { s : Case => Str ;
11       g : Gender ; n : Number ; p : Person } ;
12   Pronoun : Type = {
13     pers : PronDropForm => PronReflForm => Case => Str ;
14     poss : PronReflForm => Agr => Str ;
15     g : Gender ; n : Number ; p : Person ;
16     } ;
17 lincat
18 ---- Structural
19   Conj = {s1,s2 : Str ; n : Number} ;
20   Prep = Preposition ;
21 ---- Question
22   IDet = Determiner ; --{s : Str ; n : Number} ;
23   IP = {s : Case => Str ; n : Number} ;
24   IQuant = {s : Agr => Str} ;
25 ---- Noun
26   NP = NounPhrase ;
27   Det = Determiner ;
28   Predet = {s : Str} ;
29   Pron = Pronoun ;
30   Quant = Quantifier ;
31 ---- Common
32   CAdv = {s,p : Str} ;
```

Listing 3.1: Für **StructuralLat.gf** nötige `lincat`-Definitionen für geschlossene Kategorien

Das Lexikon einer Ressource Grammar ist unterteilt in zwei Dateien. Die erste Datei, **StructuralLat.gf**, enthält die Einträge für die geschlossenen Kategorien, so wie einige weitere Einträge die eher eine strukturelle als eine lexikalische Bedeutung haben. Die meisten Wortarten in diesem Teil des Lexikons gehören zu den so genannten Partikeln, die nicht flektiert werden. Dazu gehören vor allem Adverbien, Präpositionen und Konjunktionen.[12]. Die zweite Datei **LexiconLat.gf** wird

[12] vgl. [BL94] S.12

im nächsten Abschnitt näher beschrieben und enthält einige Vokabeln der offenen Kategorien.

Adverbien gehören eigentlich nicht zu den geschlossenen Kategorien, jedoch gibt es eine gewisse Anzahl von Adverbien und adverbial benutzten Wörtern, die den meisten Sprachen gemein sind, weswegen sie als strukturelle Bestandteile aufgefasst werden können. Meist werden Adverbien aus Adjektiven gebildet, weswegen man sie zu den offenen Kategorien rechnen sollte. Allerdings gibt es im Bereich der lokalen Adverbien (auf die Fragen wo?, wohin?, woher?) sowie vergleichende Adverbien gibt es nur ein eingeschränktes Vokabular, das zu Recht zu den geschlossenen Kategorien gerechnet werden kann.[13] in **Structural.gf** konkret als `Adv`[14] gekennzeichnet, sind im Falle der Ressource Grammar Library die Bezeichner `everywhere_Adv`, `here_Adv`, `here7to_Adv`, `here7from_Adv`, `somewhere_Adv`, `there_Adv`, `there7to_Adv` und `there7from_Adv`. Betrachtet man die Übersetzung dieser Bezeichner, so stellt sich heraus, dass die lateinischen Wörter *ubique*, *hic*, *huc*, *hinc*, *usquam*, *ibi*, *eo* und *inde* in der Lateingrammatik nicht als Adverbien, sondern als Pronominaladverbien, aufgeführt werden, also eher zur geschlossenen Kategorie der Pronomen, allerdings mit adverbialer Verwendung, gehören.

Zur selben grammatischen Kategorie gehören die meisten der im Grammatical Framework als `IAdv`[15] bezeichneten Vokabeln `how_IAdv` (lat. *qui*), `when_IAdv` (lat. *quando*) und `where_IAdv` (lat. *ubi*). Das Wort `how8much_IAdv` (lat. *quantum*) wird als korrellatives Pronomen bezeichnet, lediglich das Fragewort `why_IAdv` (lat. *cur*) ist in der gegebenen Grammatik nicht explizit eingeordnet, hat aber offensichtlich eine verwandtschaftliche Beziehung zu (Interrogativ-)Pronomen[16]. Man kann also sagen, dass hier alle als eine Form von Adverbien markierte Einträge Pronominaladverbien sind.

Die Einträge für die eben genannten Kategorien sind allerdings recht einfach, da sie meist nur die Zeichenkette mit einer einzigen Form enthalten. Anders verhält es sich bei den Interogativpronomen (`IP`) und Interrogativquantifikatoren (`IQuant`), denn diese bilden im Fall der Interrogativpronomen kasusabhängige Formen, im Falle der Quantifikatoren sind die Formen zusätzlich von Genus und Numerus abhängig. Da es jedoch nur wenige Einträge dieser Kategorien gibt, ist es nicht rentabel für sie eine zentrale, morphologische Funktion zu definieren. Deshalb müssen alle Formen direkt im Lexikon gelistet werden.

[13] vgl. [BL94] S.44
[14] verb-phrase-modifying adverb vgl. [Ran11] S. 298
[15] interrogative adverb vgl. [Ran11] S. 298
[16] vgl. wer? - lat. *quis*, Dat. *cur*

Eine weitere geschlossene Kategorie bilden die vergleichenden Adverbien. Dies sind Ausdrücke bestehend aus zwei Adverbien, die zusammen ein Verhältnis zwischen zwei Objekten ausdrücken, zwischen welche sie gesetzt werden. So drückt `less_CAdv` (lat. *minus ... quam*) aus, dass etwas kleiner oder geringer ist als etwas anderes, und `more_CAdv` (lat. *magis ... quam*) drückt aus, dass etwas größer ist. Dagegen drückt `as_CAdv` (lat. *ita ... ut*) die Gleichheit zweier Dinge aus. Objekte dieser Kategorie werden mit der Funktion `mkCAdv` aus den beiden Zeichenketten erstellt.

Eine weitere recht interessante Kategorie ist die Kategorie der Determinantia. Diese werden meist auf Basis von Adjektiven gebildet. Dadurch ist es möglich auf die Wortformenbildung von Adjektiven zurückzugreifen, um mit aus einzigen Wortform alle nötigen Formen bilden zu können.

Die letzte hier zu erwähnende einfache Kategorie von Wörtern sind Präpositionen. Präpositionen werden gemeinhin verwendet um die Funktion verschiedener Nomen-Objekte genauer zu spezifizieren. So geben Präposition auch einen Kasus an, mit dem sie verwendet werden.[17] Allerdings haben die Kasus im Lateinischen bereits eine relativ feste Funktion, die in anderen Sprachen durch Präpositionen zusammen mit einem entsprechenden Kasus ausgedrückt werden. Deshalb gibt es in dieser Lateingrammatik auch einige „leere“ Präpositionen, die also keine Zeichenkette produzieren, aber die Verwendung eines bestimmten Falles erzwingen. Zu diesen Präpositionen gehören unter anderem `part_Prep` und `posses_Prep`, deren Bedeutung schon allein durch einen Genitiv ausgedrückt wird. Andere, recht häufige, Präpositionen wie *in* können dagegen auch mit mehreren Fällen benutzt werden. Dies wird im Grammatical Framework durch so genannte freie Variationen ermöglicht (Listing 3.2). So kann *in* entweder zusammen mit Akkusativ oder Ablativ gebraucht werden. Kurz zu erwähnen sind auch Konjunktionen. Denn sie bestehen nicht nur aus einer

```
1 in_Prep = mkPrep "in" ( variants { Abl ; Acc } ) ; \
2   -- (Langenscheidts)
```

Listing 3.2: Beispiel für freie Variation (vgl. **StructuralLat.gf**)

einzelnen Zeichenkette, sondern aus einem Verbund aus zwei Zeichenketten, denn es gibt Konjunktionen wie z.B im Deutschen “sowohl ... als auch ...”, die aus zwei teilen Bestehen, von denen der erste vor die erste zu verbindende Phrase gesetzt wird und der zweite zwischen die Teile. Dieser Verbundtyp wird durch eine Hilfsfunktion namens `sd2` erzeugt, die allgemein in der Datei **Prelude.gf**[18] definiert ist.

[17] vgl. [BL94] S. 160f.
[18] im Ordner **lib/src/prelude**

```
oper
  regNP : (_,_,_,_,_,_ : Str)
    -> Gender -> Number -> NounPhrase =
    \nom,acc,gen,dat,abl,voc,g,n ->
    {
      s = table Case [ nom ; acc ; gen ; dat ; abl ; voc ] ;
      g = g ;
      n = n ;
      p = P3
    } ;
lin
  everything_NP = regNP "omnia" "omnia" "omnium" "omnis"
    "omnis" "omnia" Neutr Pl ;
```

Listing 3.3: Erzeugung des `NP`-Objekts für `everything_NP` (vgl. **ResLat.gf** und **StructuralLat.gf**)

Des weiteren sind hier auch einige komplette Nominalphrasen enthalten. Viele davon werden wieder durch Pronomenformen ausgedrückt, wie z.B. `everybody_NP`, `somebody_NP`, `something_NP`, `nobody_NP` und `nothing_NP`. Diese werden allgemein durch Formen von Indefinitpronomina gebildet. Allerdings müssen diese `NP`-Objekte unter Berücksichtigung aller Kasus-Formen, dem Genus und dem Numerus mit Hilfe der Funktion `regNP` (Listing 3.3 erzeugt werden. Die Tabellenschreibweise im `s`-Feld ist eine weitere Kurzform für Tabellen.[19] Lediglich `everything_NP` wird passender durch das Nomen *omnis* im Plural ausgedrückt. Es gibt in diesem Teil des Lexikons noch einige weitere einfache Kategorien wie satzbeginnende Konjunktionen, deren Einträge allerdings selbsterklärend sind, so dass sie hier nicht gesondert aufgeführt werden müssen.

3.1.4 Offene Kategorien

Das Lexikon der offenen Kategorien, **LexiconLat.gf**, enthält eine kleine Anzahl aus Wörtern aus den offenen Kategorien, vornehmlich Nomen, Verben und Adjektive. Der Umfang der Einträge ist dabei abhängig von der Menge an Informationen, die nötig ist um das gesamte Paradigma des Wortes zu generieren. Wovon dies abhängig ist, wird im Kapitel über die Morphologie genau beschrieben. Im allgemeinen ist, bei regelmäßiger Deklination[20] und Konjugation[21], eine einzelne Wortform ausreichen,

[19] Allgemeine Form `table` Typ_1 `[`$v_1;\ldots;v_n$`]` `;` mit $Typ_1 = V_1|\ldots|V_n$ ist gleichbedeutend mit `table {`$V_1 => v_1;\ldots;V_n => v_n$`}`

[20] Nomenflexion

[21] Verbflexion

```
1  param
2    VActForm  = VAct VAnter VTense Number Person ;
3  -- No anteriority because perfect forms are built
4  -- using participle
5    VPassForm = VPass VTense Number Person ;
6    VInfForm  = VInfActPres | VInfActPerf Gender |
7      VInfActFut Gender | VInfPassPres | VInfPassPerf Gender |
8      VinfPassFut ;
9    VImpForm  = VImp1 Number | VImp2 Number Person ;
10   VGerund   = VGenAcc | VGenGen |VGenDat | VGenAbl ;
11   VSupine   = VSupAcc | VSupAbl ;
12   VPartForm = VActPres | VActFut | VPassPerf ;
13 oper
14   Noun : Type = {s : Number => Case => Str ; g : Gender} ;
15   VV : Type = Verb ** { isAux : Bool } ;
16   Verb : Type = {
17     act   : VActForm => Str ;
18     pass  : VPassForm => Str ;
19     inf   : VInfForm => Str ;
20     imp   : VImpForm => Str ;
21     ger   : VGerund => Str ;
22     geriv : Agr => Str ;
23     sup   : VSupine => Str ;
24     part  : VPartForm =>Agr => Str ;
25     } ;
26     Adjective : Type = {
27       s : Degree => Agr => Str ;
28       comp_adv : Str ;
29       super_adv : Str
30       } ;
31 lincat
32 ---- Open lexical classes, e.g. Lexicon
33   V, VS, VQ, VA = Verb ;
34   V2, V2A, V2Q, V2S = Verb ** {c : Prep } ;
35   V3 = Verb ** {c2, c3 : Prep} ;
36   VV = VV ;
37   V2V = Verb ** {c2 : Str ; isAux : Bool} ;
38   A = Adjective ;
39   N = Noun ;
40   N2 = Noun ** { c : Prep } ;
41   N3 = Noun ** { c : Prep ; c2 : Prep } ;
42   PN = Noun ;
43   A2 = Adjective ** { c : Prep} ;
```

Listing 3.4: Für **LexiconLat.gf** nötige `lincat`-Definitionen für offene Kategorien

um daraus das gesamte Paradigma, also die Menge aller Wortformen, abhängig von den variablen Merkmalen, zu erzeugen.

Deshalb ist es bei den Nomen der ersten, zweiten, vierten und fünften Deklination, was man unter diesen Deklinationsklassen versteht, wird im Abschnitt über die Morphologie genauer erläutert, meist nicht nötig weitere Informationen anzugeben als die Nominativ-Singular-Form. Allerdings gibt es Ausnahmen, z.B. wenn bei einem Wort das Genus vom üblichen Geschlecht abweicht, das normalerweise mit der entsprechenden Endung kodiert wird. Also ist sowohl für Nomen der dritten Deklination, so wie für Nomen der anderen Deklinationsklassen nötig, statt einer Wortform zwei Wortformen, den Nominativ und Genitiv Singular, und das Geschlecht anzugeben.

Bei einigen Nomen, wie z.B. Bezeichnungen für Tiere, kann das entsprechende Nomen bei gleicher Wortform beide Genera annehmen. Deshalb wird in diesem Falle die Funktion zum Erzeugen des Paradigmas mit freier Variation über die möglichen Geschlechter, meist Femininum und Maskulinum, versehen (vgl. Listing 3.2). Andere Nomen haben dagegen sowohl eine männliche als auch eine weibliche Form, wobei diese meist sehr klar den üblicherweise entsprechenden Deklinationsklassen entsprechen. So gibt es im englischen nur ein geschlechtsunspezifisches Nomen für Cousin und Cousine. Deshalb heißt das entsprechende Symbol `cousin_N`. Die lateinische Übersetzung ist aber *consobrinus* für den männlichen Cousin und *consobrina* für das weibliche Pendant. In diesem Falle wird über die Zeichenkette variiert, aus der das Nomen-Objekt erzeugt wird.

Ein besonderer Nomeneintrag ist auch der Eintrag für `camera_N`, denn die Übersetzung dieses Begriffs im Lateinischen kann nicht durch einen einzelnen Begriff ausgedrückt werden. Stattdessen wird es mit *camera photographica* paraphrasiert. Deshalb muss für diesen Ausdruck zunächst einmal die Phrase aus ihren Bestandteilen konstruiert werden bevor sie in die Form eines einfachen Nomens gebracht wird. Dazu werden sowohl Syntaxfunktionen verwendet, die im Abschnitt 3.3 beschrieben werden, um die Phrase zu erzeugen, als auch eine Hilfsfunktion `useCNasN`[22], die die Phrase in die Form eines einfachen Nomens bringt. Da der Typ `N` nur eine Untermenge der Felder des Typs `CN` besitzt, ist die Umwandlung trivial, da lediglich alle im `N`-Typ vorhandene Felder übernommen werden.

Eine weitere Form speziellerer Nomen sind Nomen, die nur im Plural vorkommen können. Dies wird im Lexikoneintrag dadurch kodiert, dass das Nomen durch eine weitere Funktion namens `pluralN` gefiltert wird. Die Bedeutung dieser Funktion wird im Laufe der Morphologie genauer geschildert. Ein solches Nomen ist

[22] vgl. **ResLat.gf**

`science_N` das mit *literae*, der Pluralform von *litera* (Buchstabe), übersetzt wird.

Zusätzlich zu den einfachen Nomen gibt es Relationalnomen, die eine Beziehung zwischen Objekten ausdrücken. Ein Beispiel hierfür ist das Wort „Vater“, das neben seiner einfachen Verwendung, auch die Verwendung im Sinne “Vater von ...” haben kann. Deshalb benötigen diese Nomen neben ihrer einfachen Wortform auch die Information, wie diese Beziehung zu anderen Objekten ausgedrückt werden kann. Dies wird allgemein durch Präpositionen kodiert, das heißt diese Nomen haben in ihrem Lexikoneintrag zusätzlich zu den Wortformen, die nötig sind das Paradigma zu generieren, auch die Informationen zur Verwendung in Form der Angabe eines oder mehrerer Präpositionen.

Nun bleiben noch einige der moderneren Begriffe aus dem Bereich der Nomen zu nennen, nämlich `airplane_N`, `bank_N`, `bike_N`, `car_N`, `computer_N`, `fridge_N`, `paper_N`, `planet_N`, `plastic_N`, `radio_N`, `train_N` und noch einige mehr. Die Übersetzungen dieser Begriffe wurden mit den in den Abschnitten 3.1.1 und 3.1.2 genannten Quellen erfolgreich bewältigt. Abgesehen vom Auffinden einer möglichen Übersetzung ist es relativ unproblematisch gewesen, die Lexikoneinträge zu erstellen.

Die Einträge für Adjektive in diesem Lexikon sind alles in allem unproblematisch. Adjektive der ersten und zweiten Deklination können wieder aus einer einzigen Wortform, der maskulinen Nominativ-Singular-Form, erstellt werden. Lediglich bei Adjektiven der dritten Deklination wird, wie bei den Nomen, zusätzlich die Genitiv-Singular-Form benötigt. Es gibt auch im Bereich des Testlexikons **LexiconLat.gf** kein Adjektiv, bei dem das Erstellen des Eintrags in irgendeiner Form Probleme bereitet hat.

Bei den regelmäßigen Verben der ersten, zweiten und vierten Konjugation ist die einzige benötigte Information im Lexikon die Infinitiv-Präsens-Form. Dafür werden bei unregelmäßigen Verben und Verben der dritten Konjugation, wenn vorhanden, vier Verbformen benötigt. Dazu gehören neben dem Infinitiv die 1.-Person-Präsens-Indikativ-Aktiv, die 1.-Person-Perfekt-Indikativ-Aktiv und das Partizip-Perfekt-Passiv.

Die einzigen Verben, deren Übersetzung problematisch war, sind `switch8off_V2` und `switch8on_V2`. Da hier auch nicht die Wikipedia von Hilfe sein konnte, wurden zwei nahe liegende lateinische Begriffe gewählt, deren Bedeutung näherungsweise passend erschienen, nämlich *exstinguere* (löschen) und *accendere* (entzünden). Obwohl es keine direkten Übersetzungen sind, ist die Verwendung insofern gerechtfertigt, dass das entsprechende italienische Wort für “einschalten” auch *accendere* sein

kann und *exstinguere* das Gegenteil davon ist.

Hiermit sind alle problematischen oder interessanten Aspekte des Lexikons erläutert. Wie die Paradigmen aus diesen Lexikoneinträgen erzeugt werden können, wird im kommenden Abschnitt über die lateinische Morphologie ausführlich erläutert.

3.2 Morphologie

Eine der großen Herausforderung bei der Implementierung einer Lateingrammatik ist die Morphologie, da Latein eine flektierende Sprache ist, und deshalb viele grammatische Merkmale in der Wortform kodiert. Dies führt zu einer großen Menge an Wortformen für jeden Lexikoneintrag (vgl. Listing 3.5). Alles in allem haben Verben in Latein bis zu 260 Wortformen, die zur Bildung des Paradigmas komplett aufgelistet werden müssten.

```
1  ...
2  act (VAct VSim (VPres VInd) Sg P1) : dormio
3  act (VAct VSim (VPres VInd) Sg P2) : dormis
4  act (VAct VSim (VPres VInd) Sg P3) : dormit
5  act (VAct VSim (VPres VInd) Pl P1) : dormimus
6  act (VAct VSim (VPres VInd) Pl P2) : dormitis
7  act (VAct VSim (VPres VInd) Pl P3) : dormiunt
8  act (VAct VSim (VPres VConj) Sg P1) : dormiam
9  act (VAct VSim (VPres VConj) Sg P2) : dormias
10 act (VAct VSim (VPres VConj) Sg P3) : dormiat
11 act (VAct VSim (VPres VConj) Pl P1) : dormiamus
12 act (VAct VSim (VPres VConj) Pl P2) : dormiatis
13 act (VAct VSim (VPres VConj) Pl P3) : dormiant
14 ...
```

Listing 3.5: Auszug aus dem Paradigma des Verbs `sleep_V`

Um so wichtiger ist es, möglichst viele dieser Formen mit möglichst wenig Informationen zu generieren. Deshalb ist es ratsam, das Konzept der „Smart Paradigms“ zu implementieren. Dabei wird versucht Mit Hilfe von Stringanalysen und Pattern Matching möglichst viele Informationen zur Wortbildung aus den gegebenen Wortformen zu extrahieren. Im Falle der lateinischen Sprache werden dabei Wortsuffixe zu Rate gezogen. Wie so eine Stringanalyse mit entsprechender Fallunterscheidung für die Wortformenbildung aussehen kann, ist in Listing 3.6 zu sehen.

Die Implementierung der Morphologie ist hauptsächlich in der Quelltextdatei **MorphoLat.gf** zu finden, wobei die Konstruktion der konkreten Datenstrukturen, und damit auch ein Teil der Morphologie, in der Datei **ResLat.gf** zu finden ist. Allerdings sind die morphologischen Funktionen so gekapselt, dass üblicherweise zur Erzeugung eines Objektes eines Typs *Typ* die Funktion `mk`*Typ* verwendet wird. Diese Funktionen für die verschiedenen Typen sind in der Datei **ParadigmsLat.gf** definiert, rufen aber größtenteils lediglich die im Folgenden erklärten Funktionen auf.

```
1  oper
2    noun : Str -> Noun = \verbum ->
3    case verbum of {
4      _ + "a"  => noun1 verbum ;
5      _ + "us" => noun2us verbum ;
6      _ + "um" => noun2um verbum ;
7      _ + ( "er" | "ir" )
8        => noun2er verbum
9             ( (Predef.tk 2 verbum) + "ri" ) ;
10     _ + "u"  => noun4u verbum ;
11     _ + "es" => noun5 verbum ;
12     _
13       => Predef.error
14            ("3rd␣declinsion␣cannot␣be␣applied␣" ++
15            "to␣just␣one␣noun␣form␣" ++ verbum)
16   } ;
```

Listing 3.6: Beispiel für ein Smart Paradigm mit Hilfe von Pattern Matching und Fallunterscheidung (vgl. **MorphoLat.gf**)

3.2.1 Nomenflexion

Allgemeines

In der lateinischen Sprache gibt es fünf Deklinationsklassen für Nomen. Sie werden entweder durchnummeriert oder aber durch ihren Kennlaut bestimmt. Demnach unterscheidet man die erste bis fünfte Deklination bzw. die ā-, ŏ-, ĭ-, ŭ- und ē-Deklination. Zur Identifikation kann man den Kennlaut am leichtesten nach Abtrennung der Endung *-um* im Genitiv Plural erkennen. [23]. Um eine Endung, wie die soeben genannte, zu entfernen, stellt das Grammatical Framework die Funktion `tk` im Modul `Predef.gf`[24] zur Verfügung. Mit ihrer Hilfe kann eine Anzahl von Zeichen am Ende einer Zeichenkette entfernt werden.

Allerdings kann man meist die Deklinationsklasse auch an der Endung der Nominativ-Singular-Form erkennen. So haben z.B. alle Nomen der ā-Deklination den Ausgang *-ă* und das Genus Femininum. Es gibt so gut wie keine Ausnahmen, so können lediglich Flußnamen und männliche Personennamen männliches Geschlecht haben. Deshalb ist es bei fast allen Nomen dieser Deklinationsklasse nicht nötig, mehr als die Nominativ-Singular-Form anzugeben. Diese Überlegungen führen zu der Zeile 3 in Listing 3.6.

[23] vgl. [BL94] S. 21
[24] im Verzeichnis **/lib/src/prelude**

Bei der zweiten Deklinationsklasse gibt es eine größere Anzahl möglicher Wortausgänge, nämlich *-us*, *-um* und *-er* bzw. *-ir*. Grundsätzlich sind Nomen mit dem Ausgang *-um* Neutra, Nomen mit den Endungen *-us* und *-r* Maskulina. Diese Fälle sind in den Zeilen 4 bis 7 zu finden.

Die dritte oder auch ĭ-Deklination wird auch als Mischdeklination bezeichnet, da sie in zwei Unterklassen unterteilt werden kann, in Nomen mit konsonantischem oder vokalischem, also auf *-ĭ* auslautendem, Stamm. Die dritte Deklination ist deshalb eine größere und etwas schwerer zu handhabenden Flexionsklassen. In Folge dessen reicht auch nicht eine einzige Wortform für die Generierung des Paradigmas aus. Statt dessen werden die Nominativ- und Genitiv-Singular-Formen und das Genus für die Erzeugung des Paradigmas verwendet. Deshalb kommt die dritte Deklination auch nicht in Listing 3.6 vor, denn die dort gezeigte Funktion behandelt nur einzelne Stammformen. Statt dessen wird die Funktion `noun_ngg` verwendet.

Die vierte Deklinationsklasse hingegen ist wieder unkomplizierter. Sie hat im Nominativ Singular die Endungen *-ū* oder *-us* und die Nomen sind, wenn sie auf *-us* enden, maskulin und, wenn sie auf *-ū* enden, Neutra. Da die Nominativ-Singular-Form bei den Nomen auf *-us* nicht von Nomen der zweiten Deklination mit der gleichen Endung zu unterscheiden sind, kann das Smart Paradigm für nur eine Wortform nur bei den Nomen auf *-ū* angewandt werden, da die Endung *-us* schon die zweite Deklination identifiziert. In diesem Falle wird das Paradigma mit Hilfe der zwei Formen Nominativ Singular, Genitiv Singular und dem Geschlecht bestimmt. Endet die Nominativ-Singular-Form jedoch auf *-ū* endet, kann das Paradigma aus der einzelnen Stammform gebildet werden, wie in Zeile 8 von Listing 3.6 zu sehen ist.

Bei der fünften Deklination ist die Nominativ Singular-Endung an sich wieder eindeutig, sie enden alle auf *-es*. Jedoch können, wie oben bereits beschrieben, auch Nomen der dritten Deklination im Nominativ Singular auf *-es* enden. Man kann die unterschiedlichen Deklinationen aber klar an der Genitiv Singular-Form unterscheiden. Deshalb ist die sicherste Möglichkeit Fehler zu vermeiden, auch diese Genitivform im Lexikon anzugeben. Dies ist jedoch nicht nötig, da wenn nur die Nominativform angegeben ist und diese auf *-es* endet, das Smart Paradigm so definiert ist, dass ein Paradigma der fünften Deklination generiert wird, wie in Zeile 9 von Listing 3.6 zu sehen ist.

Für alle Deklinationen gilt, dass wenn entweder die Formenbildung nicht allein durch die Nominativ-Singular-Form bestimmt ist oder das Genus von dem zu erwartenden abweicht, müssen die mehrfach genannten zwei Wortformen und das Genus zur Formenbildung herangezogen werden. Dazu wird statt der Funktion `noun` in Lis-

ting 3.6 die Funktion `noun_ngg` herangezogen. Das `ngg` im Funktionsnamen steht für die Bedeutung der Parameter, nämlich Nominativ, Genitiv und Genus. Die Regeln für das Pattern Matching sind ähnlich zu denen in der `noun`-Funktion, differenzieren allerdings etwas genauer. Die Formenbildung erfolgt genau wie bei der obigen Funktion und schließlich wird mit Hilfe der Funktion `nounWithGen` das Genus, falls nötig, korrigiert. Die Bildung der Wortformen für Nomen ist relativ einfach. Der Wortstamm (vgl. Tabelle 3.1) wird meist dadurch gefunden, dass man, wenn nötig, die Nominativ-Singular- bzw. Genitiv-Singular-Endung abtrennt. Anschließend werden alle zwölf Wortformen, für die zwei Numeri und die sechs Kasus, durch anfügen der passenden Endung, die sehr regelmäßig sind, gebildet.

Wortstamm	Endung
terr	a | e
Wortstock	Wortausgang

Tabelle 3.1: Bestandteile eines lateinischen Nomens im Genitiv Singular (Vgl. [BL94] S. 21)

Erste Deklination

```
1  oper
2    -- a-Declension
3    noun1 : Str -> Noun = \mensa ->
4      let
5        mensae = mensa + "e" ;
6        mensis = init mensa + "is" ;
7      in
8      mkNoun
9        mensa (mensa +"m") mensae mensae mensa mensa
10       mensae (mensa + "s") (mensa + "rum") mensis
11       Fem ;
```

Listing 3.7: Deklinationsfunktion für die erste Deklination (vgl. **MorphoLat.gf**)

Bei der ersten Deklination ist der Wortstamm angenehmerweise gleich der Nominativ Singular-Form. Ebenfalls identisch zum Wortstamm sind die Ablativ und Vokativ Singular-Formen. Die Endungen für die restlichen Kasus sind *-m* für den Akkusativ Singular, *-e* für den Genitiv und Dativ Singular so wie Nominativ und Vokativ Plural, *-rum* für den Genitiv Plural, und *-s* für Akkusativ Plural. Etwas anders verhält es sich bei Dativ und Ablativ Plural. Bei diesen zwei Fällen wird die

Endung *-is* nicht an den Wortstamm, sondern an den Wortstock, also den Wortstamm, ohne den Kennvokal, angefügt.[25] Um den Wortstock zu erhalten, muss der Kennvokal vom Wortstamm entfernt werden. Dies geschieht mit der Hilfsfunktion `init`, dir vom Grammatical Framework zur Verfügung gestellt wird. Sie entfernt das letzte Zeichen aus einer Zeichenkette.

In der in Listing 3.7 gezeigten Funktion, in der das soeben genannte Vorgehen implementiert ist, werden zunächst für die Wortformen die öfter im Paradigma vorkommen, also die Formen mit der Endung *-ae* und *-is*, temporäre Variablen definiert, die in der Konstruktion des Paradigmas verwendet werden können. Anschließend wird mit der Funktion `mkNoun`[26] das Nomen-Objekt mit den Wortformen und dem Genus erzeugt. Die Reihenfolge der Nomenformen ist dabei Nominativ, Akkusativ, Genitiv, Dativ, Ablativ und Vokativ im Singular und Nominativ/Vokativ, Akkusativ, Genitiv, und Dativ/Ablativ.

Zweite Deklination

Bei der zweiten Nomendeklination ist das Vorgehen ganz ähnlich zur ersten Deklination, zumindest bei den Nomen auf *-us* und *-um*. Diesmal muss von der Nominativ-Singular-Form die Endung abgespalten werden um, in diesem Fall den Wortstock[27], zu erhalten. An diesen werden nun die kasusabhängigen Ausgänge angehängt. Diese sind in den meisten Fällen für alle Nomen dieser Deklinationsklasse gleich, in manchen Kasus unterscheiden sie sich aber je nach Nominativ-Singular-Endung.

Der Nominativ Singular hat offensichtlich die unterschiedlichen Endungen *-us*, *-um* oder *-r*. Bei den Nomen auf *-r* wird meist im Nominativ und Vokativ ein *-e-* eingefügt um die Aussprache zu erleichtern. Allerdings gibt es einige Nomen, die auf *-r* enden, bei denen ein *-e-* zum Wortstamm gehört, weswegen es in keinem Fall entfallen kann.[28]. Die selben Endungen haben all diese Nomen im Genitiv, Dativ, Akkusativ und Ablativ Singular (*-i*, *-o*, *-um*, *-o*) sowie im Genitiv, Dativ und Ablativ Plural (*-orum*, *-is* und *-is*). Unterschiede gibt es in den verbleibenden Fällen Vokativ Singular so wie Nominativ, Akkusativ und Vokativ Plural. Der Vokativ Singular stimmt bei Nomen auf *-um* und *-r* mit der Nominativ Singular-Form überein, Nomen auf *-us* bilden dagegen die eigenständige Form auf *-e*.

Im Plural bilden die Nomen auf *-us* und *-r* die selben Formen, im Nominativ und Vokativ mit den Endung *-i* und im Akkusativ mit *-os*. Die Neutra auf *-um* bilden in

[25] vgl. [BL94] S.21f.
[26] vgl. **ResLat.gf**
[27] vgl. Tabelle 3.1)
[28] vgl. [BL94] S. 24

allen drei Fällen Formen mit der Endung *-a*.[29] Zwar bietet sich hier möglicherweise eine Wiederverwendung gleicher Programmteile zur Implementierung an, jedoch wurde der Übersicht halber für jede Unterklasse der zweiten Deklination eine eigene Funktion, `noun2us`, `noun2um` und `noun2er`, implementiert. Außerdem ist der Mehraufwand, der nötig ist, um die Gemeinsamkeiten und Unterschiede herauszuarbeiten, in diesem Bereich kaum zu rechtfertigen, da keine großen Verbesserungen zu erwarten sind.

Dritte Deklination

Die dritte Deklination ist wohl die komplexeste Deklinationsklasse. Sie wird anhand der Wortstämme in zwei Klassen unterteilt, in die Nomen mit einem Wortstamm, der auf einen Konsonanten endet, und die Nomen, deren Wortstamm auf ein kurzes *-ĭ* endet.

Die Unterscheidung ist eine große Herausforderung, wenn man, wie bei dieser Arbeit, darauf verzichten möchte Vokalqualitäten zu unterscheiden. Denn die eigentlichen Flexionsregeln sind sowohl von Silbenzahlen als auch von Lautgesetzen abhängig. Und die Bestimmung von Silbengrenzen so wie die Anwendung von Lautgesetzen verlangt nach der Markierung von Vokallängen. Dies würde jedoch die Anwendung der Grammatik behindern. Die Markierung im Lexikon wäre nur mit einem etwas größeren Arbeitsaufwand verbunden aber noch relativ leicht machbar, da es ein einmaliger Mehraufwand wäre. Allerdings müssten auch bei jeder Eingabe für das Parsen und Übersetzen die Vokallängen berücksichtigt werden, was zu einem erheblich größeren Verarbeitungsaufwand führt.

Deshalb wurde versucht, diese Problematik zu umgehen, was zu einigen Regeln führte, die in zumindest im beschränkten Rahmen dieser Grammatik funktionieren. Sie wurden allerdings ad hoc entworfen, um die eigentlichen Regeln anzunähern und verlangen nicht nach Allgemeingültigkeit. So ist zunächst das Wort *bos* so unregelmäßig, dass es in diesem Falle einfacher ist, alle Formen einfach aufzulisten, anstatt Regeln für die Generierung zu entwerfen (Zeile 7ff. in Listing 3.8). Für einige andere Nomen wird direkt festgelegt, nach welchem Deklinationsschema die Formen gebildet werden sollen. So wird *nix* wie ein Nomen des *ĭ*-Stammes (Zeile 11f. in Listing 3.8) und *sedes*, *canis*, *iuvenis*, *mensis* und *sal* wie Nomen der Konsonantenstämmen (Zeile 13ff. in Listing 3.8) dekliniert, obwohl dies nach den nachfolgenden Regeln nicht so wäre. Diese Wörter werden aber auch in Grammatikbüchern als

[29] vgl. [BL94] S. 23f.

```
oper
  noun3 : Str -> Str -> Gender -> Noun = \rex,regis,g ->
    let
      reg : Str = Predef.tk 2 regis ;
    in
    case <rex,reg> of {
      -- Bos has to many exceptions to be handled correctly
      < "bos" , "bov" >
        => mkNoun "bos" "bovem" "bovis" "bovi" "bove"
                  "bos" "boves" "boves" "boum" "bobus" g;
      -- Some exceptions with no fitting rules
      < "nix" , _ > => noun3i rex regis g; -- Langenscheidts
      -- Bayer-Lindauer 31 3 and Exercitia Latina 32 b),
      -- sal must be handled here because otherwise it
      -- will be handled wrongly by the next rule
      < ( "sedes" | "canis" | "iuvenis" |
        "mensis" | "sal" ) ,
        _ >
          => noun3c rex regis g ;
      < _ + ( "e" | "al" | "ar" ) , _ >
        => noun3i rex regis g ; -- Bayer-Lindauer 32 2.3
      -- might not be completely right but seems fitting
      -- for Bayer-Lindauer 31 2.2
      ( < _ + "ter" , _ + "tr" >
      | < _ + "en"  , _ + "in" >
      | < _ + "s"   , _ + "r" >
        )
        => noun3c rex regis g ;
        -- Bayer-Lindauer 32 2.2
      < _ , _ + #consonant + #consonant > =>
        noun3i rex regis g ;
      < _ + ( "is" | "es" ) , _ > =>
        if_then_else
          Noun
          -- assumption based on Bayer-Lindauer 32 2.1
          ( pbool2bool ( Predef.eqInt ( Predef.length rex )
                                      ( Predef.length regis )
          ) )
          ( noun3i rex regis g )
          ( noun3c rex regis g ) ;
      _ => noun3c rex regis g
    } ;
```

Listing 3.8: Funktion zur Zuordnung von Nomen zu den Stämmen der dritten Deklination (vgl. **MorphoLat.gf**)

Ausnahmen gelistet.[30]

Die nachfolgenden Muster versuchen, die nicht umsetzbaren Entscheidungsregeln, die in der Literatur zu finden sind, mit den gegebenen Informationen zu imitieren. So gehören Nomen der dritten Deklination, die im Nominativ Singular auf *-e*, *-al* und *-ar* enden, zu den *ĭ*-Stämmen (Zeile 17f. in Listing 3.8). Dagegen ist die nächste Regel über die Zugehörigkeit zu den Konsonantenstämmen komplizierter. Die ursprüngliche Regel besagt, dass Nomen zu den Konsonantenstämmen gehören, wenn die Nominativ-Singular-Form gegenüber dem Wortstamm verändert ist. Es folgen eine Liste von Lautgesetzen, die hier Anwendung finden. Diese werden mit einer Folge von Mustern versucht anzunähern. So kann sich bei einer Ablautung des letzten Vokals bei einer Nominativ-Singular-Form auf *-ter* die Endung so verändern, dass der Wortstamm nur noch auf *-tr* endet. Es kann auch zu einer Klangfarbenänderung des letzten Vokals kommen, so dass aus der Nominativ-Singular-Form auf *-en* der Wortstamm *-in* wird. Des weiteren gibt es noch die Veränderung von *-s* zu *-r* zwischen Nominativ Singular und Wortstamm (Zeilen 19f. in Listing 3.8). Lediglich die Regel, dass sich auch nur die Vokallänge ändern kann, konnte nicht in dieser Form verwirklicht werden. Relativ problemlos dagegen ist auch die Regel, dass Nomen, deren Wortstock auf zwei oder mehr Konsonanten endet, zu den *ĭ*-Stämmen gehören (Zeile 26f. in Listing 3.8). Und die letzte Bedingung für die Zuordnung zu den Stämmen ist wieder abhängig von der Silbenzahl und kann deshalb nur angenähert werden. Statt der Silbenzahl wird bei Nomen auf *-es* und *-is* anhand der Buchstabenanzahl entschieden, zu welcher der beiden Kategorien ein Wort gehört. Haben Nominativ Singular und Genitiv Singular dieselbe Länge, so gehört das Wort zu den *ĭ*-Stämmen, andernfalls gehört es zu den Konsonantenstämmen (Zeile 28ff. in Listing 3.8).[31] Über die Allgemeingültigkeit dieser Regeln kann keine endgültige aussage getroffen werden. Allerdings liefern sie für alle Wörter im Lexikon die gewünschten Ergebnisse.

Alle Wörter, auf die keine der bisherigen Regeln zutrifft, werden einfach als den Konsonantenstämmen zugehörig angesehen.

Hat man die Nomen der dritten Deklination in *ĭ*- und Konsonantenstämme unterteilt, so kann man relativ problemlos die kompletten Paradigmen erzeugen. Bei der dritten Deklination hat man zum Erstellen des Paradigmas die Nominativ- und Genitiv-Singular-Form, so wie das Geschlecht, gegeben. Zunächst trennt man bei der Genitiv-Singular-Form die Endung *-is* ab, um den Wortstamm zu erhalten. Da Nomen der dritten Deklination allen drei Geschlechtern angehören können, und

[30] vgl. [BL94] S. 28
[31] vgl. [BL94] S. 26ff.

die Akkusativ-Singular-, Nominativ-Plural- und Akkusativ-Plural-Form geschlechtsabhängig sind, müssen diese Endungen abhängig vom Geschlecht des Wortes bestimmt werden. Dabei werden bei weiblichen und männlichen Nomen die Akkusativ-Singular-Form mit der Endung *-em* gebildet. Die beiden Fälle im Plural bilden Formen mit der Endung *-es*. Neutra bilden in allen drei Fällen die Endung *-a*. Dies gilt bei den *ĭ*-Stämmen jedoch nur für Nomen, deren Stamm auf zwei Konsonanten endet. Ist dies nicht der Fall, so sind die Endungen jeweils *-ia*.

Bei den Konsonantenstämmen wird also ein Paradigma mit den folgenden Singularformen gebildet: der gegebenen Nominativ-Form, der gegebenen Genitiv-Form, im Dativ dem Wortstamm mit der Endung *-i*, der vorher aus dem Geschlecht bestimmten Akkusativ-Form, der Ablativform mit der Endung *-e*, und im Vokativ erneut die Nominativform. Im Plural sind es die ebenfalls bereits bestimmten Nominativ Plural-Form, im Genitiv die Endung *-um*, im Dativ und Ablativ die selbe Endung *-ibus* und im Akkusativ wieder die selbe Form wie im Nominativ.

```
1  oper
2    -- i-declension
3    noun3i : Str -> Str -> Gender -> Noun = \ars,artis,g ->
4      let
5        art : Str = Predef.tk 2 artis ;
6        artemes : Str * Str = case g of {
7          Masc | Fem => < art + "em" , art + "es" > ;
8          Neutr => case art of {
9            -- seems to be working
10           _ + #consonant + #consonant => < ars , art + "a" > ;
11           _ => < ars , art + "ia" > -- Bayer-Lindauer 32 4
12           }
13         } ;
14       arte : Str = case ars of {
15         _ + ( "e" | "al" | "ar" ) => art + "i" ;
16         _ => art + "e"
17         };
18     in
19     mkNoun
20       ars artemes.p1 artis ( art + "i" ) arte ars
21       artemes.p2 artemes.p2 ( art + "ium" ) ( art + "ibus" )
22       g ;
```

Listing 3.9: Die Deklinationsfunktionen für die Nomen der dritten Deklination der *ĭ*-Stämme (vgl. **MorphoLat.gf**)

Bei den *ĭ*-Stämmen werden im Singular die Formen nach dem selben Schema gebildet, abgesehen von der Ablativform. Denn bei Nomen der *ĭ*-Stämme, die im

Nominativ Singular auf *-e*, *-al* und *-ar* enden, wird die Ablativ-Form mit der Endung *-i* gebildet, bei allen anderen, wie die Konsonantenstämmen, mit *-e*. Im Plural weicht die Genitivform ab, denn die Endung lautet hier *-ium* statt *-um*.[32]

Vierte Deklination

Nomen der vierten Deklination können in der Nominativ-Singular-Form auf *-u* oder *-us* enden.

Die Nomen auf *-us* im Nominativ Singular haben diese Endung auch im Genitiv und Vokativ Singular so wie im Nominativ, Akkusativ und Vokativ Plural. Im Dativ Singular enden die Formen auf *-ui* und im Akkusativ Singular auf *-um*. Die verbleibenden Endungen im Plural sind *-uum* im Genitiv und *-ibus* sowohl im Dativ als auch im Ablativ. Da die meisten Endungen mit einem *-u-* beginnen, wird dieser Buchstabe in der Implementierung direkt bei der Abspaltung der Nominativ Singular-Endung am Wortstamm belassen und nur für Dativ und Ablativ Plural explizit entfernt. Es werden also für die Erzeugung des Paradigmas zwei temporäre Zeichenketten verwendet, zum einen der Wortstamm und zum anderen der Wortstamm mit der Endung *-u*.

Bei den Nomen auf *-u*, die größtenteils Neutra sind, enden fast alle Formen des Singular auf *-u*. Lediglich im Genitiv enden die Formen auf *-us*. Um Plural hat man die für Neutra relativ üblichen Endungen auf *-a* bzw. hier auf *-ua* im Nominativ, Akkusativ und Vokativ. Die verbleibenden Pluralendungen sind identisch zu denen der Nomen auf *-us*. Zur Generierung des Paradigmas werden bei den Nomen auf *-u* drei Zeichenketten verwendet. Zum einen die Nominativ Singular-Form, die fünf mal im Paradigma ohne Endung und zwei mal mit verschiedenen Endungen vorkommt. Des weiteren der Wortstamm, der zwei mal mit Endungen vorkommt. Und schließlich der Wortstamm mit der Endung *-ua*, der immerhin drei mal im Paradigma vertreten ist.[33]

Fünfte Deklination

Die letzte Deklinationsklasse für Nomen, die fünfte oder *e*-Deklination, besteht aus den Nomen, die im Nominativ Singular auf *-es* enden. Die Vokativ-Form im Singular und Plural ist auch hier, wie fast immer, gleich der Nominativ-Form. Zusätzlich hat auch der Akkusativ Plural die gleiche Form. Genitiv und Dativ Singular enden beide auf *-ei*, und Akkusativ Singular auf *-em*. Der Ablativ Singular hat nur den Ausgang

[32] vgl. [BL94] S. 28
[33] vgl. [BL94] S. 33

-e, also keine Endung am Wortstamm. Im Plural endet die Genitiv-Form auf *-erum* und Dativ so wie Ablativ auf *-ebus*. Um das Paradigma zu erzeugen wird neben der Nominativ-Singular-Form, der davon abgeleitete Wortstamm so wie die Form, die aus dem Wortstamm mit dem Ausgang *-i* besteht, verwendet.[34]

Betrachtet man die Nomenendungen im Gesamtüberblick, so kann man auch oberhalb der Deklinationsklassen Muster erkennen, die man versuchen könnte zu formalisieren. Allerdings wäre der Nutzen davon wohl eher gering und würde zu größeren Problemen in den Details führen. Außerdem war ein Aspekt bei dieser Arbeit die Nähe zu einer gegebenen gedruckten Schulgrammatik. Deshalb wurde auch bei der Implementierung die etablierte Einteilung in die Deklinationsklassen gewahrt.

Sonderfälle

Die häufigste Sonderform von Nomen dürften die bereits im Lexikon-Kapitel erwähnten Nomen sein, die in der gewünschten Bedeutung nur Pluralformen bilden. Um das Paradigma für diese Nomen zu bilden wird lediglich das vollständige Nomenparadigma gebildet und anschließend durch die Hilfsfunktion `pluralN` alle Singularformen durch die Fehlerzeichenkette *######* ersetzt, die signalisiert, dass diese Formen nicht existieren. In einer möglichen zukünftigen Fassung der Lateingrammatik kann diese improvisierte Lösung durch die neue Methode ersetzt werden, den `Maybe`-Typ zu verwenden.

3.2.2 Adjektivflexion

Im Lateinischen müssen Adjektive mit dem Nomen in Genus, Numerus und Kasus übereinstimmen. Zusätzlich gibt es drei Steigerungsstufen, Positiv, Komparativ und Superlativ. Deshalb werden sie in diesen Merkmalen flektiert. Auf diese Art und Weise enthält das Paradigma ca. 108 Wortformen (3 Genera x 2 Numeri x 6 Kasus x 3 Steigerungsformen). Diese zu generieren ist aber relativ einfach, nachdem man bereits eine funktionierende Nomenflexion hat. Denn die Adjektive bilden, durch ihre Kongruenz mit Nomen, meist die gleichen Formen wie diese durch sie attribuierten Nomen.

Viele Adjektive gehören zur ersten und zweiten Deklination. Adjektive dieser Klasse bilden für Feminina die selben Endungen wie Nomen der ersten Deklination und verhalten sich für Maskulina und Neutra jeweils analog zu Nomen der zweiten Deklination auf *-us* (Maskulina) und *-um* (Neutra). Alle weiteren Adjektive werden zur

[34] vgl. [BL94] S. 34

dritten Adjektivdeklinationsklasse gezählt. Diese haben ebenfalls einige Gemeinsamkeiten. So haben all diese Adjektive die Endung *-e* bzw. *-i* im Ablativ Singular, *-um* bzw. *-ium* im Genitiv Plural und *-a* bzw. *-ia* im Nominativ, Vokativ und Akkusativ Plural des Neutrums, je nach Zugehörigkeit zu den Konsonanten- oder *ĭ*-Stämmen. Denn diese werden auch hier wieder unterschieden.[35]

Bei Adjektiven der ersten und zweiten Deklination kann wieder das ganze Paradigma aus einer einzigen Zeichenkette erzeugt werden. Ebenfalls ist dies bei Adjektiven auf *-is* und *-x* möglich, die zur dritten Deklination gehören. Sollte eine Zeichenkette nicht genügen, wie bei den meisten Adjektiven aus der dritten Deklinationsklasse, so kann das Paradigma aus zwei gegebenen Formen, Nominativ Singular und Genitiv Singular der maskulinen Form, generieren werden. Für sehr seltene Adjektive, für die auch diese Möglichkeit nicht ausreichend ist, kann aus drei Wortformen, nämlich den drei Nominativ Singular-Formen, alle weiteren Wortformen generiert werden. Diese Option wird jedoch im bisherigen Lexikon nicht benötigt.[36]

Die Deklinationsklasse der Adjektive wird, wenn nur eine Form, die Nominativ Singular-Form bei Maskulinum, gegeben ist, wieder anhand der Endung bestimmt. Ist diese *-us*, so ist das Adjektiv drei-endig[37] und gehört zur ersten und zweiten Deklination. Hat es eine andere Endung, so gehört es zur dritten Deklination. Sind zwei Wortformen vorhanden, so wird zusätzlich die gegebene Genitiv Singular-Form bei Maskulina betrachtet. Ist die gegebene Nominativ-Endung *-us* und die Genitiv-Endung *-i*, so ist, wie bereits gesagt, das Adjektiv drei-endig. Ist dagegen die Genitiv-Endung *-is*, so ist das Adjektiv Teil der dritten Deklination. Zur dritten Deklination gehören auch alle anderen Adjektive, die im Genitiv bei Maskulina auf *-is* enden so wie alle Adjektive die im gegebenen Nominativ auf *-is* und im entsprechenden Genitiv auf *-e* enden. Alle anderen Adjektive mit zwei Wortformen führen zu einem Fehler. Sollte dieser Fehler für ein Adjektiv im Lexikon auftreten, so muss man zur Erzeugung des Paradigmas die Funktion verwenden, die die drei Nominativ-Singular-Formen verwendet.[38]

Erste und zweite Deklination

Das Paradigma für Adjektive der ersten und zweiten Deklination wird folgendermaßen gebildet. Zunächst wird der Wortstamm bestimmt (Zeile 4-12 in Listing 3.10).

[35] vgl. [BL94] S. 38

[36] vgl. **ParadigmsLat.gf** und **MorphoLat.gf**

[37] Unter drei-endig versteht ‚man, wenn ein Adjektiv in jedem Genus eine andere Endung hat, unter zwei-endig, wenn das Adjektiv bei Femininum und Maskulinum die selbe Endung hat, diese sich jedoch von der Neutrum-Endung unterscheidet

[38] vgl. [BL94] S. 36 u. S. 38

```
oper
  adj12 : Str -> Adjective = \bonus ->
    let
    bon : Str = case bonus of {
      -- Exceptions Bayer-Lindauer 41 3.2
      ("asper" | "liber" | "miser" | "tener" | "frugifer")
        => bonus ;
      -- Usual cases
      pulch + "er" => pulch + "r" ;
      bon + "us" => bon ;
      _ => Predef.error ("adj12 does not apply to" ++ bonus)
        } ;
    nbonus = (noun12 bonus) ;
    compsup : ( Agr => Str ) * ( Agr => Str ) =
        -- Bayer-Lindauer 50 4
      case bonus of { (_ + #vowel + "us" ) |
                      (_ + "r" + "us" ) =>
           < table { Ag g n c => table Gender
                [ ( noun12 bonus ).s ! n ! c ;
                  ( noun12 ( bon + "a" ) ).s ! n ! c ;
                  ( noun12 ( bon + "um" ) ).s ! n ! c
                ] ! g
              } ,
              table { Ag g n c => table Gender
                  [ ( noun12 bonus ).s ! n ! c ;
                    ( noun12 ( bon + "a" ) ).s ! n ! c ;
                    ( noun12 ( bon + "um" ) ).s ! n ! c
                  ] ! g
              }
           > ;
          _ => comp_super nbonus
        };
      advs : Str * Str =
        case bonus of {
          -- Bayer-Lindauer 50 4
          idon + ( #vowel | "r" ) + "us" =>
            < "magis" , "maxime" > ;
          _ => < "" , "" >
        }
    in
    mkAdjective
    nbonus (noun1 (bon + "a")) (noun2um (bon + "um"))
    < compsup.p1 , advs.p1 > < compsup.p2 , advs.p2 > ;
```

Listing 3.10: Deklinationsfunktion für drei-endige Adjektive der ersten und zweiten Deklination (vgl. **MorphoLat.gf**)

Normalerweise entspricht der Wortstamm der Nominativ Singular-Form ohne die geschlechtsspezifische Endung. Also in diesem Fall, bei Maskulina, *-us*. Endet das Adjektiv allerdings auf *-er* statt auf *-us*, so ist der Wortstamm diese Nominativ-Form ohne das *-e-*. In einigen wenigen Ausnahmefällen entspricht er allerdings der Nominativ-Singular-Maskulin-Form. Zu diesen Ausnahmen gehören unter anderem *asper*, *liber*, *miser*, etc. Bei diesen Adjektiven auf *-er* bleibt also das *-e-* auch in allen anderen Formen erhalten. Als nächstes wird aus der maskulinen Nominativ-Singular-Form ein Nomenparadigma generiert, als ob es sich bei der Adjektivform um die Grundform eines Nomens handelt, und für die spätere Verwendung zwischengespeichert. Dazu wird die im Nomen-Abschnitt dieses Kapitel beschriebene Funktion verwendet, um ein Nomenparadigma der zweiten Deklination auf *-us* zu bilden (Zeile 13 in Listing 3.10. Dieses Nomen-Objekt wird unter anderem für die Erzeugung der Steigerungsformen benötigt, weswegen es zunächst in einer temporären Variable abgelegt wird, bevor es zur Erzeugung des Adjektiv-Objekts in Zeile 40 verwendet wird. Die Nomen-Objekte für die beiden verbleibenden Genera werden direkt am Ort ihrer Verwendung (Zeile 41f. in Listing 3.10) erzeugt.

Komparation

Die „Berechnungen“ in den Zeilen 14 bis 37 in Listing 3.10 werden verwendet um die Wortformen der Steigerungsstufen des Adjektivs zu erzeugen. Normalerweise wird die Steigerung von Adjektiven durch Flexion ausgedrückt. Es müssen also eigene Wortformen für jede Steigerungsstufe generiert werden. Bei manchen Adjektiven wird dies jedoch statt dessen mit der Positivform und entsprechenden Adverbien umschrieben. Adjektive, die so eine Umschreibung benötigen, enden allgemein auf *-us*, wobei aber der Wortstamm selbst wieder entweder auf einen Vokal oder *-r-* endet. Nach dieser Regel wird z.B. bei *arduus* und *mirus* nicht, wie später beschrieben, die Steigerung morphologisch kodiert, sondern mit den Adverbien *magis* (Komparativ) und *maxime* (Superlativ) umschrieben. Deshalb muss für diese Wörter nur die Positivform generiert werden. Bei allen anderen Adjektiven der ersten und zweiten Deklination werden für die Steigerung neue Wortstämme gebildet, an die wiederum die für die erste und zweite Deklination üblichen Endungen angehängt werden.[39]

Die Bildung des neuen Wortstammes ist nicht ganz trivial (vgl. Listing 3.11). Zunächst einmal gibt es in der lateinischen Sprache Adjektive, deren Komparativ- und Superlativstamm kaum Gemeinsamkeiten mit der Grundform haben. Dazu zählen z.B *bonus* (komp. *melior*, sup. *optimus*), *malus* (komp. *peior*, sup. *pessimus*), *magnus*

[39] vgl. [BL94] S. 36f

```
oper
  comp_super : Noun -> ( Agr => Str ) * ( Agr => Str ) =
    \bonus ->
    case bonus.s!Sg!Gen of {
      -- Exception Bayer-Lindauer 50 1
      "boni" => < comp "meli" ,
                  table { Ag g n c =>
                    table Gender
                      [ (noun2us "optimus").s ! n ! c ;
                        (noun1 "optima").s ! n ! c ;
                        (noun2um "optimum").s ! n ! c
                      ] ! g
                  }
                > ;
      "mali" => < comp "pei" , super "pessus" > ;
      "magni" => < comp "mai" ,
                   table { Ag g n c =>
                     table Gender
                       [ (noun2us "maximus").s ! n ! c ;
                         (noun1 "maxima").s ! n ! c ;
                         (noun2um "maximum").s ! n ! c
                       ] ! g
                     }
                  > ;
      "parvi" => < comp "mini" ,
                   table { Ag g n c =>
                     table Gender
                       [ (noun2us "minimus").s ! n ! c ;
                         (noun1 "minima").s ! n ! c ;
                         (noun2um "minimum").s ! n ! c
                       ] ! g
                     }
                  >;
      --Exception Bayer-Lindauer 50.3
      "novi" => < comp "recenti" , super "recens" > ;
      "feri" => < comp "feroci" , super "ferox" > ;
      "sacris" => < comp "sancti" , super "sanctus" >;
      "frugiferi" => < comp "fertilis" , super "fertilis" > ;
      "veti" => < comp "vetusti" , super "vetustus" >;
      "inopis" => < comp "egentis" , super "egens" >;
      -- Default Case use Singular Genetive to determine
      -- comparative
      sggen => < comp sggen , super (bonus.s!Sg!Nom) >
    } ;
```

Listing 3.11: Funktion zur Bestimmung der Komparativ- und Superlativformen eines Adjektivs (vgl. **MorphoLat.gf**)

(komp. *maior*, sup. *maximus*), *parvus* (komp. *minor*, sup. *minimus*), etc. Für jedes dieser Wörter muss eine eigene Regel existieren, wie der Wortstamm im Komparativ und Superlativ aussieht. Teilweise sind es wirklich nur Abbildungen auf eine neue Genitiv- und eine neue Nominativ-Form, die wie bei den regelmäßigen Adjektiven für die Bildung der Steigerungsformen verwendet werden können. Teilweise wird aber auch sogleich das entsprechende Nomenparadigma für den Superlativ gebildet. Dies hängt davon ab, ob die Superlativform eine der üblichen Superlativendungen hat oder nicht. Zur Zuordnung sowie zur Generierung der Komparativformen wird als kennzeichnende Form die Genitiv-Singular-Form des bereits erstellten Nomenparadigmas verwendet. Diese Wortform hat den Vorteil, dass sie bei allen Adjektiven den wirklichen Wortstamm enthält, was im Nominativ wie schon ausgeführt, nicht immer der Fall ist. Der Superlativ dagegen wird aus der Nominativ-Form gebildet.[40]

Der Komparativ wird üblicherweise durch das Nominativ-Suffix *-ior* für Feminina und Maskulina und *-ium* für Neutra ausgedrückt. Adjektive sind also im Komparativ zwei- statt drei-endig. Die Endungen im Singular sind *-ior*, *-ioris*, *-iori*, *-iorem*, *-iore* im Nominativ/Vokativ, Genitiv, Dativ, Akkusativ und Ablativ. Im Plural sind es entsprechend *-iores*, *-iorum*, *-ioribus*, *-iores* und *-ioribus*. Die Neutrumformen unterscheiden sich nur im Nominativ, Vokativ und Akkusativ von den Femininum-/Maskulinumformen, nämlich *-ius* im Singular und *-ia* im Plural. Diese Endungen werden an den Wortstamm, also die Genitiv-Form ohne die Genitivendung *-i* bzw. *-is*, angehängt.[41]

Der Superlativ ist hingegen wieder drei-endig und bildet die Formen nach der ersten und zweiten Deklination. Dafür ist es für den Superlativ nicht so leicht, das passende Suffix zu bilden, das abhängig von der Nominativ-Singular-Maskulin-Form des Adjektivs ist. Die Bestimmung dieses Suffixes ist in der Funktion `super` (vgl. Listing 3.12) zu sehen. Endet die Nominativ-Singular-Maskulin-Form auf *-er* so werden die Suffixe *-rimus,-a,-um* verwendet, dagegen verwendet man bei Wörtern auf *-lis* die Suffixe *-limus,-a,-um*, in allen anderen Fällen die Suffixe *-issimus,-a,-um*. Hinzu kommt allerdings noch eine mögliche Lautveränderung am Ende des Wortstocks. So wird beim Anhängen von *-issimus,-a,-um* aus einem *-x* ein *-c-* und endet die Grundform des Wortes auf *-ns*, so wird es im Wortinneren zu einem *-nt-*. Das komplette Superlativ-Paradigma wird wieder nach dem Schema für Nomen der ersten und zweiten Deklination erzeugt.[42]

Für die bereits genannten Adjektive, die ihre Komparation durch Adverbien und

[40] vgl. [BL94] S. 40ff.
[41] vgl. [BL94] S. 40f.
[42] vgl. [BL94] S. 42

```
oper
  super : Str -> ( Agr => Str ) = \bonus ->
    let
      prefix : Str = case bonus of {
        -- Bayer-Lindauer 48 2
        ac + "er" => bonus ;
        -- Bayer-Lindauer 48 3
        faci + "lis" => faci + "l" ;
        -- Bayer-Lindauer 48 1
        feli + "x" => feli + "c" ;
        -- Bayer-Lindauer 48 1
        ege + "ns" => ege + "nt" ;
        -- Bayer-Lindauer 48 1
        bon + ( "us" | "is") => bon
        };
      suffix : Str = case bonus of {
        ac + "er" => "rim" ; -- Bayer-Lindauer 48 2
        faci + "lis" => "lim" ; -- Bayer-Lindauer 48 3
        _ => "issim" -- Bayer-Lindauer 48 1
        };
    in
    table {
      Ag Fem n c =>
        (noun1 ( prefix + suffix + "a" )).s ! n ! c ;
      Ag Masc n c =>
        (noun2us ( prefix + suffix + "us" )).s ! n ! c;
      Ag Neutr n c =>
        (noun2um ( prefix + suffix + "um" )).s ! n ! c
    } ;
```

Listing 3.12: Erzeugung der Superlativ-Formen eines Adjektivs (vgl. **MorphoLat.gf**)

nicht durch Morphologie kodieren, wird zusätzlich zur entsprechenden Steigerungsstufe das passende Adverb gespeichert oder, wenn keines benötigt wird, die leeren Zeichenketten. Zum Schluss werden alle Einzelbestandteile, also die Positiv-, Komparativ- und Superlativformen so wie die möglicherweise nötigen Adverbien zu einem Adjektiv-Objekt verbunden, das die Form hat, wie sie in Zeile 22-26 in Listing 3.4, zu sehen ist.

Dritte Deklination

Für die Adjektive der dritten Deklination stimmt das Vorgehen größtenteils mit dem gerade beschriebenen Vorgehen für die erste und zweite Deklination überein. Allerdings sind zwei Wortformen für die Generierung des Paradigmas nötig, die Nominativ- und die Genitiv-Form des Maskulinums im Singular. Zunächst wird die Endung von dieser gegebenen Nominativ-Form abgetrennt und die Nominativformen für alle drei Genera gebildet. Dabei können Adjektive der dritten Deklination entweder drei-endig (m. *acer*, f. *acris*, n. *acre*), wenn sie als Nominativ Maskulin-Form auf *-er* enden, zwei-endig (m./f. *fortis*, n. *forte*), wenn die Nominativform auf *-is* endet, oder sonst auch ein-endig (m./f./n. *felix*) sein.

In diesem Falle kann nicht so klar auf die Nomenflexion zurückgegriffen werden. Statt dessen wird das Paradigma direkt aus zwei gegebenen Formen und dem gewünschten Geschlecht hergeleitet. Dazu wird zunächst der Wortstamm durch das Abtrennen der Genitivendung von der entsprechenden Form gebildet. Anschließend wird von Geschlecht abhängig die Akkusativ Singular- (Endung: m./f. *-em*, n. keine Endung) und Nominativ/Vokativ/Akkusativ-Plural-Form (Endung: m./f. *-es*, n. *-ia*) gebildet, ebenso wie die geschlechtsunabhängige Dativ/Ablativ-Singular-Form mit der Endung *-i*. Zusammen mit der feststehenden Genitiv-Singular- (*-is*), Genitiv-Plural- (*-ium*) und Dativ/Ablativ-Plural-Endung (*-ibus*) können alle Formen des Paradigmas gebildet werden (vgl. Listing 3.13).

Darauf folgt die Bildung der Komparativ- und Superlativformen genauso wie bei den Adjektiven der vorherigen Klasse. Abschließend wird wieder das Adjektiv-Objekt zusammengesetzt. Nur werden bei Adjektiven dieser Deklination die Steigerungsformen immer durch Flexion kodiert. Deshalb bleiben die Felder für die Steigerungsadverbien leer.

3.2.3 Verbflexion

Verben bilden im Lateinischen von allen Wortarten die meisten Formen. Bei finiten Verben werden folgende Merkmale unterschieden: Person, Numerus, Tempus,

```
oper
  noun3adj : Str -> Str -> Gender -> Noun =
    \audax,audacis,g ->
    let
      audac   = Predef.tk 2 audacis ;
      audacem = case g of {
        Neutr => audax ;
        _ => audac + "em"} ;
      audaces = case g of {
        Neutr => audac +"ia" ;
        _ => audac + "es"} ;
      audaci  = audac + "i" ;
    in
    mkNoun
      audax audacem (audac + "is") audaci audaci
      audax audaces audaces (audac + "ium")
      (audac + "ibus")
      g ;
```

Listing 3.13: Deklinationsfunktion für die „Nomenformen“ der Adjektive der dritten Deklination (vgl. **ResLat.gf**)

Modus, und Diathese, jeweils mit unterschiedlichen Wertebereichen. So gibt es drei Personen, zwei Numeri (Singular und Plural), sechs Zeitformen (Präsens, Imperfekt, Perfekt, Plusquamperfekt, Futur I und Futur II), drei Modi (Indikativ, Konjunktiv, Imperativ I und Imperativ II) und zwei Diathesen (Aktiv und Passiv). Hinzukommen infinitivische Verbformen wie der Infinitiv im Präsens, Perfekt und Futur, das Gerundium, das Supin, Partizipien im Präsens, Perfekt und Futur so wie das Gerundiv. Dabei zählen die Infinitive, das Gerundium und das Supin nach ihrer Verwendung und Formenbildung zu den substantivischen Formen und die Partizipien und das Gerundiv zu den adjektivischen Verbformen.[43] Für jede dieser Formen ist im Datentyp für lateinische Verben im Grammatical Framework (vgl. Listing 3.4) ein Feld vorhanden. In den Typen der Felder ist auch jeweils kodiert, in welchen Merkmalen diese Form flektiert wird. So werden Aktiv-Formen eines Verbes nach Zeitstufe und Tempus, die zusammen die üblichen Tempora bilden, Numerus und Person flektiert.

Die wenigsten der lateinischen Verben bilden jedoch tatsächlich alle diese Verbformen. So kann ein komplettes Passiv nur von transitiven Verben gebildet werden[44]. Dagegen gibt es im Lateinischen so genannte Deponentia, die zwar aktivisch verwendet werden, allerdings nur passive Formen bilden[45]. Ganz zu schweigen von

[43] vgl. [BL94] S. 66
[44] vgl. [BL94] S. 67
[45] vgl. [BL94] S. 83

all den Besonderheiten der unregelmäßigen Verben[46]. Um fehlende Verbformen zu markieren wurde eine Zeichenkette gewählt, die in Eingaben mit an Sicherheit grenzender Wahrscheinlichkeit nicht vorkommt. Die gewählte Zeichenkette ist wieder ###### und sollte ebenfalls in Zukunft auf die neue `Maybe`-Methode geändert werden. Sie sollte entsprechend an den nötigen Stellen behandelt werden. Allerdings ist die Fehlerbehandlung im Grammatical Framework momentan noch eher rudimentär. Die Behandlungen von fehlenden Werten soll aber in Zukunft durch die Einführung eines in Haskell und anderen funktionalen Programmiersprachen vorhandenen `Maybe`- oder auch Option-Datentyps ermöglicht werden. Dieser polymorphe Datentyp hat für einen konkreten Datentyp zwei mögliche Werte, entweder `Just` a mit einem konkreten Wert $a$eines anderen Datentyps, wenn solch ein Wert vorhanden ist, oder `Nothing`, wenn kein Wert vorhanden ist.

Konjugationsklassen

In der lateinischen Sprache gibt es für die Konjugation[47] der Verben, ähnlich wie die Deklinationsklassen der Nomen, vier Klassen. Diese Konjugationsklassen verhalten sich teilweise auch ganz analog zu den Deklinationsklassen der Nomen und Adjektiven. Die Konjugationsklassen werden wieder anhand von Kennlauten unterschieden. Die erste Konjugation hat, wie die erste Deklination, als Kennlaut den Vokal -$\bar{a}$-, die zweite den Kennlaut -$\bar{e}$- und die vierte den Kennlaut -$\bar{i}$-. Die dritte Konjugation ist wieder unterteilt, zum einen in die konsonantische Konjugation und zum anderen in die kurzvokalische Konjugation. Wie der Name schon sagt, ist der Kennlaut der konsonantischen Konjugation ein Konsonant, und bei der kurzvokalischen Konjugation entweder der Kurzvokal -$\check{u}$- oder -$\check{i}$-.[48]

Diese Klassen gelten sowohl für die Konjugation der regulären Verben als auch für die Deponentia. Denn für die Bildung des Verbparadigmas werden primär diese beiden Arten von Verben unterschieden, denn die meisten Verben gehören einer dieser beiden Verbklassen an. Das Paradigma für die meisten Verben der ersten, zweiten und vierten Konjugation, sowohl bei den normalen Verben als auch bei den Deponentia, kann von einer einzigen Verbform, der Infinitiv-Präsens-Aktiv-Form, gebildet werden. Für Verben der dritten Konjugation so wie unregelmäßigere Verben der anderen Konjugationsklassen werden vier Verbformen verwendet, neben der Infinitiv-Präsens-Aktiv-Form die 1.-Person-Singular-Präsens-Indikativ-Aktiv-Form, die 1.-Person-Singular-Perfekt-Indikativ-Aktiv-Form, sowie die Parti-

[46] vgl. [BL94] S. 105ff.
[47] Verbflexion
[48] vgl. [BL94] S. 68f.

zip-Perfekt-Passiv-Form.

Für die Verben der ersten, zweiten und vierten Konjugation kann die Zugehörigkeit zur entsprechenden Konjugationsklasse am Wortausgang[49] der Infinitiv-Präsens-Aktiv-Form abgelesen werden. Endet die Verbform auf *-a-*, *-e-* oder *-i-* mit der Endung *-ri*, so handelt es sich um ein Deponens der ersten, zweiten oder vierten Konjugation. Endet die Verbform dagegen auf *-a-*, *-e-* oder *-i-* mit der Endung *-re*, so handelt es sich entsprechend um ein reguläres Verb der ersten, zweiten oder vierten Konjugation.

```
oper
  verb_ippp : (iacere,iacio,ieci,iactus : Str)
    -> Verb =
    \iacere,iacio,ieci,iactus ->
    case iacere of {
      _ + "ari" => deponent1 iacere ;
      _ + "eri" => deponent2 iacere ;
      _ + "iri" => deponent4 iacere ;
      _ + "i" => case iacio of {
        _ + "ior" => deponent3i iacere iacio iactus ;
        _ => deponent3c iacere iacio iactus
        } ;
      _ + "are" => verb1 iacere ;
      _ + "ire" => verb4 iacere ; -- ieci iactus ;
      _ + "ere" => case iacio of {
        -- Bayer-Lindauer 74 1
        _ + #consonant + "o" => verb3c iacere ieci iactus ;
        _ + "eo" => verb2 iacere ;
        -- Bayer-Lindauer 74 1
        _ + ( "i" | "u" ) + "o" => verb3i iacere ieci iactus ;
        _ => verb3c iacere ieci iactus
        } ;
      _ => Predef.error
             ("verb_ippp: illegal infinitive form" ++ iacere)
    } ;
```

Listing 3.14: Smart Paradigm für vier Verbformen (vgl. **MorphoLat.gf**)

Zur Einordnung der Verben der dritten Konjugation sind mindestens zwei Wortformen, neben der Infinitiv-Präsens-Aktiv- auch noch die 1.-Person-Singular-Präsens-Indikativ-Aktiv-Form, nötig. üblicherweise werden allerdings gleich vier Verbformen für die Formenbildung verwendet, nämlich auch noch 1.-Person-Perfekt-Indikativ-Aktiv und Partizip-Perfekt-Passiv (vgl. Listing 3.14). Endet die 1.-Person-Sin-

[49] Kennlaut zusammen mit der Endung vgl. Tabelle 3.1

gular-Präsens-Aktiv-Form nämlich nur auf *-i*, so gehört sie zu den Deponentia der dritten Konjugation. Der Kennlaut zur Unterscheidung in Konsonantenstämme oder Kurzvokalstämme erscheint erst bei den finiten Präsensformen, weshalb eine von diesen, nämlich die erwähnte 1.-Person-Singular-Präsens-Indikativ-Aktiv-Form, zu Rate gezogen wird. Ist in diesem Falle der Wortausgang *-ior*, so gehört das Wort zu den Kurzvokalstämmen der dritten Konjugation, sonst gehört es zu den Konsonantenstämmen. Endet die Infinitiv-Form dagegen auf *-ere*, muss folgendermaßen unterschieden werden: Endet die 1.-Person-Singular-Präsens-Indikativ-Aktiv-Form auf einen Konsonanten gefolgt von *-o*, so gehört das Verb zu den Konsonantenstämmen der dritten Konjugation, endet diese Form auf *-eo*, so gehört sie statt zur dritten zur zweiten Konjugation, was jedoch nicht allein anhand der Infinitivform zu unterscheiden ist. Endet die Wortform auf einen der beiden Kennvokale der kurzvokalischen dritten Deklination gefolgt von einem *-o*, so ist das Wort offensichtlich Teil der Kurzvokalstämme, in allen anderen Fällen ist es teil der Konsonantenstämme der dritten Konjugation.[50]

Verbstämme

Für jede Konjugationsklasse werden nun einige Wortformen und -stämme gebildet, aus denen das gesamte Paradigma erstellt werden kann. Einige der dafür erstellten Formen mögen in einer der Konjugationsklassen redundant sein, das heißt für mehrere Merkmale wird die selbe Zeichenkette gebildet, dies liegt jedoch an der einheitlichen Form der Behandlung aller Verben.

In Lateingrammatiken findet man meist drei Arten von Wortstämmen eines Verbes. Zu diesen gehört zunächst der Präsensstamm, von dem allgemein die Präsens-, Imperfekt- und Futur-I-Formen im Aktiv und Passiv, das Gerundium, das Gerundiv und das Partizip so wie der Infinitiv-Präsens gebildet werden. Der Perfektstamm dagegen wird verwendet, um die Perfekt-, Plusquamperfekt- und Futur-II-Formen im Aktiv so wie den Infinitiv-Perfekt im Aktiv zu bilden. Zuletzt gibt es noch den Partizipialstamm, von dem die Perfekt-, Plusquamperfekt- und Futur-II-Formen im Passiv, zusammen mit dem Hilfsverb esse, so wie das Partizip- und der Infinitiv-Futur im Aktiv gebildet werden.[51]

Zur zusätzlichen Erleichterung bei der Formenbildung werden neben den drei Verbstämmen auch für jede Zeitform und jeden Modus der entsprechende Wortstock und der Infinitiv-Präsens-Aktiv verwendet. Alles in allem werden für das gesamte

[50] vgl. [BL94] S. 68f.
[51] vgl. [BL94] S. 66

Verbparadigma also 15, bei Deponentia lediglich 9, wegen des unvollständigen Paradigmas, Grundformen verwendet. An diese verschiedenen Zeichenketten werden die Endungen angehängt, die die Merkmale wie Person, Numerus, Diathese, etc. kodieren.

In der lateinischen Sprache werden grammatische Merkmale bei Verben an verschiedenen Stellen kodiert. Einerseits gibt es Infixe, die Tempus, im Bereich von Präsens, Imperfekt und Futur, und Modus, im Bereich von Indikativ und Konjunktiv, kodieren. Zum anderen werden in der Endung die Diathese, also Aktiv oder Passiv, der Modus des Imperativs, vor allem jedoch Numerus und Person, kodiert.[52]

```
1  oper
2  -- 1./a-conjugation
3    verb1 : Str -> Verb = \laudare ->
4      let
5        lauda = Predef.tk 2 laudare ;
6        laud = init lauda ;
7        laudav = lauda + "v" ;
8        pres_stem = lauda ;
9        pres_ind_base = lauda ;
10       pres_conj_base = laud + "e" ;
11       impf_ind_base = lauda + "ba" ;
12       impf_conj_base = lauda + "re" ;
13       fut_I_base = lauda + "bi" ;
14       imp_base = lauda ;
15       perf_stem = laudav ;
16       perf_ind_base = laudav ;
17       perf_conj_base = laudav + "eri" ;
18       pqperf_ind_base = laudav + "era" ;
19       pqperf_conj_base = laudav + "isse" ;
20       fut_II_base = laudav + "eri" ;
21       part_stem = lauda + "t" ;
22     in
23     mkVerb
24       laudare pres_stem pres_ind_base pres_conj_base
25       impf_ind_base impf_conj_base fut_I_base imp_base
26       perf_stem perf_ind_base perf_conj_base pqperf_ind_base
27       pqperf_conj_base fut_II_base part_stem ;
```

Listing 3.15: Bildung der Wortstämme und -stöcke für Verben der ersten Konjugation (vgl. **MorphoLat.gf**)

Diese 15 bzw. 9 Formen werden für jede Konjugationsklasse unabhängig gebildet. Für reguläre Verben und Deponentia der ersten, zweiten und dritten Konjugation

[52] vgl. [BL94] S. 71f.

werden alle Zeichenketten alleinig aus der Infinitiv-Präsens-Aktiv-Form gebildet. Der erste Schritt auf dem Weg zum vollen Paradigma ist es, die Infinitiv-Präsens-Endung, *-re* bei regulären Verben und *-ri* bei Deponentia, abzutrennen um den Präsensstamm zu erhalten. Der Wortstock im Präsens Indikativ ist zu diesem identisch, womit schon die ersten zwei der benötigten Formen vorhanden sind. Die dritte, der Stamm bei Präsens Konjunktiv, wird in der ersten Konjugation durch Ersetzen des Kennlauts gebildet. Der Kennlaut *-ā-* wird durch ein *-e-* ersetzt. Bei der zweiten und vierten Konjugation wird statt dessen an den Präsensstamm ein *-a-* angehängt. Für das Imperfekt wird an den Präsensstamm ein Suffix angefügt, das sowohl diese Zeitstufe als auch den Modus ausdrückt. Dieses Suffix ist für den Indikativ *-ba-* und für den Konjunktiv *-re-*. Lediglich bei der vierten Konjugation wird zwischen Stamm und Suffix noch ein *-e-* eingeschoben. Die sechste Form, der Wortstock des Futur Indikativ, wird analog zum Imperfekt, durch ein Suffix ausgedrückt, in diesem Falle *-bi-* bei Verben der ersten und zweiten und *-e-* bei Verben der vierten Konjugation. Damit sind alle Wortstöcke, die auf dem Präsensstamm basieren, gebildet.

Als nächstes folgt der Perfektstamm. Dieser, so wie alle auf ihm basierenden Wortstöcke, existieren nur bei den regulären Verben und nicht bei den Deponentia, denn diese bilden alle vorzeitigen Zeitformen, Perfekt, Plusquamperfekt und Futur II im Gegensatz zu Präsens, Imperfekt und Futur, mit Hilfe des Partizips zusammen mit einer Form des Hilfsverbs *esse*. Bei allen anderen Verben wird die Vorzeitigkeit durch das Suffix *-v-* bzw. *-u-* ausgedrückt. Deshalb wird bei diesen Verben der Perfektstamm im Grunde aus dem Präsensstamm durch Anhängen des passenden Suffixes gebildet. Bei Verben der ersten und vierten Konjugation geschieht dies wirklich nur durch Anhängen des Suffixes *-v-*. Bei der zweiten Konjugation jedoch entfällt der Kennvokal *-ē-*und statt des Suffixes *-v-* wird das Suffix *-u-* angehängt. Der Perfekt-Indikativ-Stamm ist wieder identisch zum Perfektstamm. Im Konjunktiv wird dagegen ein weiteres Suffix *-eri-* benötigt. Das selbe betrifft die Plusquamperfekt-Stämme mit den Suffixen *-era-* im Indikativ und *-isse-* so wie den Futur-II-Stamm mit dem Suffix *-eri-*.

Die letzte fehlende Zeichenkette, um das Paradigma generieren zu können, ist der Partizipialstamm, der auch wieder für die Deponentia gebildet wird. Dazu wird an den Präsensstamm einfach das Suffix *-t-* angehängt. Lediglich bei Verben und Deponentia der zweiten Konjugation wird zusätzlich der Kennvokal *-ē-* zu einem *-i-*. Zusammen mit dem als Ausgangsbasis gewählten Infinitiv sind damit alle 15 bzw. 9 Formen bzw. Formenbestandteile gebildet, die verwendet werden, um das

Paradigma zu generieren.[53]

```
oper
-- 3./Consonant conjugation
  deponent3c : ( sequi,sequor,secutus : Str ) -> Verb =
  \sequi,sequor,secutus ->
    let
      sequ = Predef.tk 2 sequor ;
      secu = Predef.tk 3 secutus ;
      pres_stem = sequ ;
      pres_ind_base = sequ ;
      pres_conj_base = sequ + "a" ;
      impf_ind_base = sequ + "eba" ;
      impf_conj_base = sequ + "ere" ;
      fut_I_base = sequ + "e" ;
      imp_base = sequi ;
      part_stem = secu + "t" ;
    in
    mkDeponent
      sequi pres_stem pres_ind_base pres_conj_base
      impf_ind_base impf_conj_base fut_I_base imp_base
      part_stem ;
```

Listing 3.16: Bildung der Wortstämme und -stöcke für Deponentia der dritten Konjugation mit konsonantischem Stamm (vgl. **MorphoLat.gf**)

Die Bildung der soeben genannten Formen ist für die Verben und Deponentia der dritten Konjugation etwas anders, vor allem weil die Gesamtheit der Wortformen nicht nur von einer einzelnen Wortform, wie bei den anderen Konjugationsklassen, sondern von drei Wortformen gebildet wird. Die zusätzlichen Wortformen sind 1.-Person-Singular-Perfekt-Indikativ-Aktiv und das Partizip-Perfekt-Passiv. Der Präsensstamm wird bei Konsonantenstämmen und kurzvokalischen Stämmen unterschiedlich gebildet. Zunächst wird bei beiden die Infinitiv-Endung, diesmal *-ere*, abgetrennt. Bei den kurzvokalischen Stämmen wird stattdessen ein Suffix *-i-* angefügt. Bei den Deponentia wird bei den Konsonantenstämmen die Endung *-or* von dem 1.-Person-Singular-Präsens-Indikativ-Aktiv abgetrennt, bei den Kurzvokalstämmen fällt der Präsensstamm mit dem 1.-Person-Singular-Präsens-Indikativ-Aktiv zusammen. Die restlichen Präsensformen werden genauso gebildet, wie bei der vierten Konjugation, ebenso der Stamm bei Imperfekt-Indikativ. Beim Imperfekt-Konjunktiv-Stamm wird lediglich zusätzlich vor dem Suffix ein *-e-* eingefügt. Der Futur-I-Stamm fällt bei den Konsonantenstämmen mit dem Präsensstamm zusam-

[53] vgl. [BL94] S. 68ff.

men, bei den kurzvokalischen Stämmen wird wie bei der vierten Konjugation das Suffix *-e-* angehängt. Der Perfektstamm, wo vorhanden, wird von der gegebenen Perfektform gebildet, indem die Endung *-i* abgetrennt wird. Die restlichen Perfekt-, Plusquamperfekt- und Futur-II-Formen werden analog zur vierten Konjugation aus dem Perfektstamm gebildet. Der Partizipstamm, als letzte nötige Zeichenkette, wird aus der gegebenen Partizip-Form durch Abtrennen der Endung *-us* gebildet.[54]

Verbformenbildung

```
oper
  actPresEnding : Number -> Person -> Str =
    useEndingTable
      <"m", "s", "t", "mus", "tis", "nt"> ;

  actPerfEnding : Number -> Person -> Str =
    useEndingTable
      <"i", "isti", "it", "imus", "istis", "erunt"> ;

  passPresEnding : Number -> Person -> Str =
    useEndingTable
      <"r", "ris", "tur", "mur", "mini", "ntur"> ;

  passFutEnding : Str -> Number -> Person -> Str =
    \lauda,n,p ->
    let endings : Str * Str * Str * Str * Str * Str =
      case lauda of {
        ( _ + "a" ) |
        ( _ + "e" ) =>
          < "bo" , "be" , "bi" , "bi" , "bi" , "bu" > ;
        _           =>
          < "a"  , "e"  , "e"  , "e"  , "e"  , "e"  >
      }
    in
    (useEndingTable endings n p) + passPresEnding n p ;
```

Listing 3.17: Funktionen um Verbendungen zu bilden und bereitzustellen (vgl. **ResLat.gf**)

Hat man all diese Wortstämme und -stöcke, so kann man einen großen Teil des Paradigmas allein durch das Anhängen der entsprechenden Endung bilden. Die Endungen sind zunächst einmal anhand des Modus in zwei Gruppen unterteilt, die Endungen für Indikativ und Konjunktiv sowie die Imperativ-Endungen. Erstere sind

[54] vgl. [BL94] S. 68ff.

wieder unterteilt in Aktivendungen (*-m* - 1. Person Singular, *-s* - 2. Person Singular, *-t* - 3. Person Singular, *-mus* - 1. Person Plural, *-tis* - 2. Person Plural, *-nt* - 3. Person Plural), Aktivendungen bei Vorzeitigkeit[55] (*-i*, *-is-ti*, *-it*, *-imus*, *-is-tis*, *-er-unt-*) und Passivendungen (*-r*, *-ris*, *-tur*,*-mur*, *-mini*, *-ntur*). Zweitere sind unterteilt in Aktivendungen (*-e* oder keine Endung - 2. Person Singular Imperativ I, *-to* - 2./3. Person Imperativ Singular II, *-te* - 2. Person Plural Imperativ I, *-to-te* - 2. Person Plural Imperativ, *-nto* - 3. Person Plural Imperativ II) und Imperativendungen bei Deponentia (*-re* - 2. Person Singular Imperativ I, *-tor* - 2./3. Person Singular Imperativ II, *-mini* - 2. Person Plural Imperativ I, *-ntor* - 3. Person Plural Imperativ II).[56] Allerdings kommen dabei immer wieder Lautgesetze zum tragen, weswegen es immer wieder Besonderheiten bei der Formenbildung gibt. Wenn der entsprechende Wortstamm auf einen gewissen Laut endet, wird bei einigen Formen zwischen Stamm und Endung noch ein zusätzlicher Vokal eingefügt. Deshalb werden in den folgenden Abschnitten die Details bei der Formenbildung genau ausgeführt.

Reguläre Formenbildung

```
1  oper
2    mkVerb :
3      (regere,reg,regi,rega,regeba,regere,rege,regi,
4      rex,rex,rexeri,rexera,rexisse,rexeri,rect : Str)
5      -> Verb =
6      \inf_act_pres,pres_stem,pres_ind_base,pres_conj_base,
7      impf_ind_base,impf_conj_base,fut_I_base,imp_base,
8      perf_stem,perf_ind_base,perf_conj_base,pqperf_ind_base,
9      pqperf_conj_base,fut_II_base,part_stem ->
10     let
11       fill : Str * Str * Str = case pres_stem of {
12         _ + ("a" | "e") => <"","","">;
13         _ + #consonant => <"e","u","i">;
14         _ => <"e","u","">
15         };
16     in
17     {
18 ...
19     }
```

Listing 3.18: Kopf der Funktion um reguläre Verbformen zu bilden (vgl. **ResLat.gf**)

Zunächst einmal soll die Paradigmenbildung der regulären Verben geschildert wer-

[55] auch Perfekt-Aktiv-Endungen
[56] vgl. [BL94] S. 72

den. Die entsprechende, sehr umfangreiche Funktion `mkVerb` ist in der Datei **ResLat.gf** komplett zu finden.

Beginnt man bei den Präsens-Indikativ-Aktiv-Formen, so ist man schon bei der 1.-Person-Singular mit einer recht unregelmäßigen Form konfrontiert. Denn es ist neben der 1.-Person-Singular-Futur-I-Indikativ-Aktiv-Form die einzige, die nicht die übliche 1.-Person-Singular-Endung *-m*, sondern die Endung *-o* hat. Des weiteren wird, wenn der Stamm auf ein *-a* endet, dieses entfernt, bevor die Endung angefügt wird. Bei den restlichen Formen wird lediglich unter Umständen ein zusätzlicher Vokal, abhängig vom Ende des Präsens-Stammes, eingefügt. Endet der Stamm auf *-a* oder *-e*, so wird kein zusätzlicher Vokal eingefügt. Endet der Stamm auf einen Konsonanten, so wird bei der 3. Person Plural der Vokal *-u-* und bei jeder anderen Form der Vokal *-i* eingefügt, bei jedem anderen Vokal wird nur bei der 3. Person Plural der Vokal *-u* eingefügt, bei anderen Formen allerdings nichts. Im Konjunktiv dagegen wird einfach die zu Person und Numerus passende Endung an den Präsens-Konjunktiv-Aktiv-Stamm angehängt, um alle Formen zu bilden. Analoges erfolgt bei den beiden Modi des Imperfekt. Lediglich bei den Futur-I-Aktiv-Formen muss der 1. Person Singular und der 3. Person Plural besondere Beachtung geschenkt werden. Endet der Futur-I-Wortstamm auf *-bi* so wird bei der 1. Person Singular das *-i* durch ein *-o* ersetzt und keine weitere Endung angefügt. Andernfalls wird der letzte Buchstabe des Stammes durch ein *-a* ersetzt und die 1.-Person-Singular-Präsens-Indikativ-Aktiv-Endung angefügt. Bei der 3. Person Plural wird, falls der Stamm auf *-bi* endet, das *-i* durch ein *-u* ersetzt, bevor die Endung angehängt wird. Endet der Stamm nicht auf *-bi* so wird nur die entsprechende Endung angehängt. Bei den Perfekt-Indikativ-Formen muss lediglich die passende Perfekt-Aktiv-Endung an den der Zeitstufe entsprechenden Wortstamm angehängt werden. Bei den Plusquamperfekt-Formen und Futur-II-Formen wird dagegen die Präsens-Aktiv-Endung an den der Zeitform entsprechenden Wortstamm angehängt. Dabei tritt bei der 1. Person Singular aber ein Sonderfall ein. Statt eine Endung anzuhängen wird der letzte Buchstabe des Wortstamms entfernt und durch ein *-o* ersetzt. Damit sind schon einmal alle Aktiv-Formen des Paradigmas gebildet.[57]

Als nächstes soll die Bildung der Passivformen beschrieben werden. Im Passiv müssen nur die Präsens-, Imperfekt- und Futur-I-Formen gebildet werden, denn alle vorzeitigen Verbformen werden wieder durch das Partizip-Perfekt-Passiv zusammen mit einer passenden Form des Hilfsverbs *esse* gebildet. Wie bei den Aktiv-Formen folgt die Mehrzahl der Formen dem regelmäßigen Grundschema, einige Formen aber

[57] vgl. [BL94] S. 74f., S. 78f. u. S. 84f.

```
act =
  table {
    VAct VSim (VPres VInd)  Sg P1 => -- Present Indicative
    ( case pres_ind_base of {
      _ + "a" =>  ( init pres_ind_base ) ;
      _ => pres_ind_base
    }
    ) + "o" ; --actPresEnding Sg P1 ;
    VAct VSim (VPres VInd)  Pl P3 => -- Present Indicative
      pres_ind_base + fill.p2 + actPresEnding Pl P3 ;
    VAct VSim (VPres VInd)  n  p  => -- Present Indicative
      pres_ind_base + fill.p3 + actPresEnding n p ;
    VAct VSim (VPres VConj) n  p  => -- Present Conjunctive
      pres_conj_base + actPresEnding n p ;
    VAct VSim (VImpf VInd)  n  p  => -- Imperfect Indicative
      impf_ind_base + actPresEnding n p ;
    VAct VSim (VImpf VConj) n  p  => -- Imperfect Conjunctive
      impf_conj_base + actPresEnding n p ;
    VAct VSim VFut          Sg P1 => -- Future I
      case fut_I_base of {
        _ + "bi" => ( init fut_I_base ) + "o" ;
        _  => ( init fut_I_base ) + "a" + actPresEnding Sg P1
      } ;
    VAct VSim VFut          Pl P3 => -- Future I
      ( case fut_I_base of {
        _ + "bi" => ( init fut_I_base ) + "u";
        _ => fut_I_base
      }
      ) + actPresEnding Pl P3 ;
    VAct VSim VFut          n  p  => -- Future I
      fut_I_base + actPresEnding n p ;
    VAct VAnt (VPres VInd)  n  p  => -- Prefect Indicative
      perf_ind_base + actPerfEnding n p ;
    VAct VAnt (VPres VConj) n  p  => -- Prefect Conjunctive
      perf_conj_base + actPresEnding n p ;
    VAct VAnt (VImpf VInd)  n  p  => -- Plusperfect Indicative
      pqperf_ind_base + actPresEnding n p ;
    VAct VAnt (VImpf VConj) n  p  => -- Plusperfect Conjunctive
      pqperf_conj_base + actPresEnding n p ;
    VAct VAnt VFut          Sg P1 => -- Future II
      ( init fut_II_base ) + "o" ;
    VAct VAnt VFut          n  p  => -- Future II
      fut_II_base + actPresEnding n p
  } ;
```

Listing 3.19: Ausschnitt aus der Funktion `mkVerb` um aktive Verbformen zu bilden (vgl. **ResLat.gf**)

```
pass =
  table {
    VPass (VPres VInd)  Sg P1 => -- Present Indicative
      ( case pres_ind_base of {
          _ + "a" => (init pres_ind_base ) ;
          _ => pres_ind_base
        }
      ) + "o" + passPresEnding Sg P1 ;
    VPass (VPres VInd)  Sg P2 => -- Present Indicative
      ( case imp_base of {
        _ + #consonant =>
          ( case pres_ind_base of {
              _ + "i" => ( init pres_ind_base ) ;
              _ => pres_ind_base
            } ) + "e" ;
        _ => pres_ind_base
        }
      ) + passPresEnding Sg P2 ;
    VPass (VPres VInd)  Pl P3 => -- Present Indicative
      pres_ind_base + fill.p2 + passPresEnding Pl P3 ;
    VPass (VPres VInd)  n  p  => -- Present Indicative
      pres_ind_base + fill.p3 + passPresEnding n p ;
    VPass (VPres VConj) n  p  => -- Present Conjunctive
      pres_conj_base + passPresEnding n p ;
    VPass (VImpf VInd)  n  p  => -- Imperfect Indicative
      impf_ind_base + passPresEnding n p ;
    VPass (VImpf VConj) n  p  => -- Imperfect Conjunctive
      impf_conj_base + passPresEnding n p ;
    VPass VFut          Sg P1 => -- Future I
      ( case fut_I_base of {
          _ + "bi" => ( init fut_I_base ) + "o" ;
          _ => ( init fut_I_base ) + "a"
        }
      ) + passPresEnding Sg P1 ;
    VPass VFut          Sg P2 => -- Future I
      ( init fut_I_base ) + "e" + passPresEnding Sg P2 ;
    VPass VFut          Pl P3 => -- Future I
      ( case fut_I_base of {
          _ + "bi" => ( init fut_I_base ) + "u" ;
          _ => fut_I_base
        } ) + passPresEnding Pl P3 ;
    VPass VFut          n  p  => -- Future I
      fut_I_base + passPresEnding n p
  } ;
```

Listing 3.20: Ausschnitt aus der Funktion `mkVerb` um passive Verbformen zu bilden (vgl. **ResLat.gf**)

weichen mehr oder weniger stark davon ab. Dies beginnt bereits bei den Präsens-Indikativ-Formen. Schon bei der Form für die 1. Person Singular muss, wenn der Präsens-Indikativ-Stamm auf ein *-a* endet, dieses abgetrennt werden, bevor die passende Endung angefügt werden kann. Bei der 2. Person Singular muss, wenn der Imperativ-Stamm auf einen Konsonanten und der Präsens-Indikativ-Stamm auf ein *-i* endet, dieses entfernt und durch ein *-e* ersetzt werden. Endet der Präsens-Indikativ-Stamm jedoch nicht auf ein *-i* wird nur ein *-e* angehängt bevor die passende Endung angefügt wird. Trifft die Bedingung bei dem Imperativ-Stamm nicht zu, wird die Endung einfach an den Präsens-Stamm angehängt. Die restlichen Formen des Präsens-Indikativ verhalten sich wie die entsprechenden Aktiv-Formen. Es werden lediglich die Passiv-Endungen statt der Aktiv-Endungen angehängt. Ebenso bei Präsens-Konjunktiv-, Imperfekt- und den meisten der Futur-I-Formen. Lediglich die 2. Person Singular verhält sich anders als im Aktiv. Denn in diesem Falle wird zwischen Stamm und Endung noch ein *-e-* eingefügt. Da, wie bereits gesagt, im Passiv alle Perfekt-, Plusquamperfekt- und Futur-II-Formen durch Partizipien umschrieben werden, sind damit auch alle Passiv-Formen beschrieben.[58]

Damit sind die häufigsten Formen des Verbparadigmas bereits bekannt. Als nächstes folgen die Infinitiv-Formen. Infinitive gibt es in Latein für Präsens, Perfekt und Futur jeweils als Aktiv- und Passiv-Form. Die Infinitiv-Präsens-Aktiv-Form ist schon von Grund auf bekannt, da es die Verbform ist, die in jedem Verbeintrag im Lexikon vorkommen muss. Die Infinitiv-Perfekt-Aktiv-Form dagegen muss aus dem Perfekt-Stamm mit der Endung *-isse* gebildet werden. Im Infinitiv-Futur-Aktiv dagegen basiert die Form auf dem Partizip-Futur-Aktiv und ist deshalb geschlechtsabhängig. An den Partizip-Stamm wird für Maskulina und Neutra *-urum* und für Feminina *-uram* angehängt. Des weiteren basiert dieser Infinitiv auf der Infinitiv-Präsens-Aktiv-Form des Hilfsverbs *esse*, welche aber als Zeichenkette im Paradigma, hier und auch zukünftig, nicht explizit enthalten sein wird. Im Passiv ist der Infinitiv-Präsens wie der Infinitiv-Aktiv-Präsens, jedoch mit der Endung *-ri* statt der Endung *-re*. Der Infinitiv-Perfekt-Passiv bildet wie der Infinitiv-Futur-Aktiv genusabhängige Formen zusammen mit dem Hilfsverb *esse*. Die Formen sind der Partizipialstamm mit den Endungen *-um*, *-am* und *-um*. Der Infinitiv-Futur-Passiv wird aus dem Partizipialstamm mit der Endung *-um* und der Verbform *iri*[59] gebildet.[60]

Es folgen die Imperativ-Formen, von denen es üblicherweise sechs Stück gibt. Zum einen den Imperativ I, der direkte Aufforderungen ausdrückt, in einer Singular- und

[58] vgl. [BL94] S. 76f u. S. 80f.
[59] Infinitiv-Präsens-Passiv des Verbs *ire*
[60] vgl. [BL94] S. 82

```
1  inf =
2    table {
3      VInfActPres       => -- Infinitive Active Present
4        inf_act_pres ;
5      VInfActPerf _     => -- Infinitive Active Perfect
6        perf_stem + "isse" ;
7      VInfActFut Masc   => -- Infinitive Active Future
8        part_stem + "urum" ;
9      VInfActFut Fem    => -- Infinitive Active Future
10       part_stem + "uram" ;
11     VInfActFut Neutr  => -- Infinitive Active Future
12       part_stem + "urum" ;
13     VInfPassPres      => -- Infinitive Present Passive
14       ( init inf_act_pres ) + "i" ;
15     VInfPassPerf Masc => -- Infinitive Perfect Passive
16       part_stem + "um" ;
17     VInfPassPerf Fem  => -- Infinitive Perfect Passive
18       part_stem + "am" ;
19     VInfPassPerf Neutr => -- Infinitive Perfect Passive
20       part_stem + "um" ;
21     VInfPassFut       => -- Infinitive Future Passive
22       part_stem + "um"
23   } ;
```

Listing 3.21: Ausschnitt aus der Funktion `mkVerb` um Infinitiv-Verbformen zu bilden (vgl. **ResLat.gf**)

```
imp =
  let
    imp_fill : Str * Str =
      case imp_base of {
        _ + #consonant => < "e" , "i" > ;
        _ => < "" , "" >
      };
  in
  table {
    VImp1 Sg          => -- Imperative I
      imp_base + imp_fill.p1 ;
    VImp1 Pl          => -- Imperative I
      imp_base + imp_fill.p2 + "te" ;
    VImp2 Sg ( P2 | P3 ) => -- Imperative II
      imp_base + imp_fill.p2 + "to" ;
    VImp2 Pl P2       => -- Imperative II
      imp_base + fill.p3 + "tote" ;
    VImp2 Pl P3       => -- Imperative II
      pres_stem + fill.p2 + "nto" ;
    _ => "######" -- No imperative form
  } ;
```

Listing 3.22: Ausschnitt aus der Funktion `mkVerb` um Infinitiv-Verbformen zu bilden (vgl. **ResLat.gf**)

einer Plural-Form. Und den Imperativ II, der eher in die fernere Zukunft gerichtet ist oder allgemeiner verwendet wird. Von ihm gibt es zwei Singular- und zwei Plural-Formen, jeweils für die 2. und 3. Person. Dabei fallen allerdings die beiden Singular-Formen zusammen, bilden also die gleiche Zeichenkette. Der Imperativ-I-Singular besteht aus dem Imperativ-Stamm an den keine Endung angehängt wird, außer der Vokal *-e*, wenn der Stamm auf einen Konsonanten endet. Dann wird angehängt. Im Plural wird an den Stamm die Endung *-to* angehängt und, wenn der Stamm auf einen Konsonanten endet ein Vokal *-i-* eingeschoben. Die beiden Imperativ-II-Singular-Formen werden genau so wie der Imperativ-I-Plural gebildet, jedoch mit der Endung *-to* statt der Endung *-te*. In Plural dagegen wird bei der 2. Person die Endung *-tote* angehängt. Davor wird allerdings, wenn der Stamm auf einen Konsonant endet, ein *-i-* eingefügt. Und bei der 3. Person wird, wenn der Stamm nicht auf ein *-a* oder *-u* endet, ein *-u-* eingefügt, bevor die Endung *-nto* angefügt wird.[61]

```
1  ger =
2    table {
3      VGenAcc => -- Gerund
4        pres_stem + fill.p1 + "ndum" ;
5      VGenGen => -- Gerund
6        pres_stem + fill.p1 + "ndi" ;
7      VGenDat => -- Gerund
8        pres_stem + fill.p1 + "ndo" ;
9      VGenAbl => -- Gerund
10       pres_stem + fill.p1 + "ndo"
11   } ;
```

Listing 3.23: Ausschnitt aus der Funktion `mkVerb` um Gerundiv-Verbformen zu bilden (vgl. **ResLat.gf**)

Die Gerund-Formen zählen zwar zu den substantivischen Nominalformen des Verbs, bilden allerdings nur vier Kasusformen, Genitiv, Dativ, Akkusativ und Ablativ. Diese Formen werden im Grunde analog zu den Formen eines Nomens der zweiten Deklination gebildet. Gebildet werden sie, indem an den Präsens-Stamm die Endungen *-ndi* (Genitiv), *-ndo* (Dativ und Ablativ) und *-ndum* (Akkusativ) angehängt werden. Wieder wird, wenn der Stamm nicht auf ein *-a* oder *-e* endet, ein *-e-* eingeschoben. [62]

Das Gerundiv hingegen wird adjektivisch verwendet und verhält sich so wie ein drei-endiges Adjektiv der ersten und zweiten Deklination. Die Stammformen stam-

[61] vgl. [BL94] S. 82
[62] vgl. [BL94] S. 82

men vom Gerund ab und bestehen aus dem Präsens-Stamm, unter Umständen gefolgt von einem *-e-* und den Endungen *-ndus*, *-nda* und *-ndum*. Die restlichen Formen werden wie von den Adjektiven gewohnt gebildet.[63]

Ebenfalls wie Adjektive gebildet und verwendet werden die Partizipien. Es gibt drei Partizipien im Lateinischen. Das Partizip-Präsens-Aktiv, das Partizip-Futur-Aktiv und das Partizip-Perfekt-Passiv. Das Partizip-Präsens-Aktiv wird wie ein einendiges Adjektiv gebildet. Die Grundform besteht, wie bei Gerund und Gerundiv, aus dem Präsens-Stamm, gefolgt von einem *-e-*, wenn der Stamm nicht auf ein *-e* oder *-a* endet, und der Endung *-ns*. Der Genitiv wird analog mit der Endung *-ntis* gebildet. Das Partizip-Futur-Aktiv so wie das Partizip-Perfekt-Passiv werden wieder wie drei-endige Adjektive gebildet. Die Nominativ-Formen werden mit Hilfe des Partizip-Stammes gebildet. Beim Partizip-Futur-Aktiv lauten die drei Nominativ-Endungen *-urus*, *-urum* und *-urum*, beim Partizip-Perfekt-Passiv *-us*, *-a* und *-um*.[64]

```
1 sup =
2   table {
3     VSupAcc => -- Supin
4       part_stem + "um" ;
5     VSupAbl => -- Supin
6       part_stem + "u"
7   } ;
```

Listing 3.24: Ausschnitt aus der Funktion `mkVerb` um Supin-Verbformen zu bilden (vgl. **ResLat.gf**)

Die beiden letzten Verbformen sind die Supin-Formen. Diese Formen werden wie die Maskulin-Akkusativ- und Maskulin-Ablativ-Singular-Form des Partizip-Perfekt-Passivs gebildet.[65]

Auf die Verwendung einiger dieser Formen wird später im Syntaxteil noch genauer eingegangen. Viele der in diesem Abschnitt beschriebenen Verbformen werden in speziellen Konstruktionen verwendet, deren Implementierung noch für die Zukunft Möglichkeiten der Beschäftigung bietet. Sie wurden aber trotzdem in die Wortbildung einbezogen, um diesen Abschnitt möglichst vollständig abzuschließen und so eine solide Grundlage für mögliche spätere Arbeiten zu bieten.

An dieser Stelle kann man sich auch kurz überlegen, ob es sinnvoll ist all diese Verbformen an einer einzelnen Stelle zu bilden. Denn lediglich die Aktiv- und

[63] vgl. [BL94] S. 82
[64] vgl. [BL94] S. 82
[65] vgl. [BL94] S. 82

Passiv-Formen werden in der Syntax verwendet, wie man es für Verben gewohnt ist. Deshalb kann man möglicherweise nur die Bildung dieser Formen als Flexionsmorphologie ansehen, die an dieser Stelle dargestellt werden sollte. Alle anderen Formen, die hier gebildet werden, kann man dagegen als Derivationsmorphologie auffassen, die als eigenständige Funktion andernorts umgesetzt werden sollte um bei Bedarf aus gegebenen Verben andere Wortarten formen zu können. Jedoch findet man die Verbflexion in der hier besprochenen Form in gängigen Schulgrammatiken vor. Deshalb wurde diese Form auch für diese Arbeit gewählt.

Deponentia

```
1  oper
2    mkDeponent : ( sequi,sequ,sequi,sequa,sequeba,sequere,
3      seque,sequi,secut : Str ) -> Verb =
4      \inf_pres,pres_stem,pres_ind_base,pres_conj_base,
5      impf_ind_base,impf_conj_base,fut_I_base,imp_base,
6      part_stem ->
7    let fill : Str * Str =
8    case pres_ind_base of {
9      _ + ( "a" | "e" ) => < "" , "" >;
10     _ => < "u" , "e" >
11   }
12   in
13   {
14 ...
15   }
```

Listing 3.25: Kopf der Funktion um Deponentia-Formen zu bilden (vgl. **ResLat.gf**)

Die zweitgrößte Gruppe der lateinischen Verben nach den regulären Verben sind die so genannten Deponentia. Deren Name kommt vom Verb *deponere* (ablegen), da man sagen kann, dass diese Verben ihre aktiven Formen bzw. ihre passive Bedeutung abgelegt haben.[66]

Aus diesem Grunde ist das Paradigma, im Vergleich zu den regulären Verben, nicht vollständig. Es müssen also auch weniger Formen gebildet werden. Deshalb werden für die Bildung des ganzen Paradigmas auch weniger Wortstämme benötigt. Wie diese gebildet werden, ist bereits in einem vorangegangenen Kapitel, beschrieben worden. Die Gesamtheit der Wortformen wird mit Hilfe der Funktion `mkDeponent` gebildet, die der Funktion `mkVerb` stark ähnelt.

[66] vgl. [BL94] S. 83

```
act =  table {
    VAct VSim (VPres VInd)  Sg P1 => -- Present Indicative
      ( case pres_ind_base of {
        _ + "a" =>  ( init pres_ind_base ) ;
          _ => pres_ind_base } ) + "o" + passPresEnding Sg P1 ;
    VAct VSim (VPres VInd)  Sg P2 => -- Present Indicative
      ( case inf_pres of {
          _ + "ri" => pres_ind_base  ;
          _ => ( case pres_ind_base of {
            _ + "i" => init pres_ind_base ;
            _ => pres_ind_base } ) + "e"
        } ) + passPresEnding Sg P2 ;
    VAct VSim (VPres VInd)  Pl P3 => -- Present Indicative
      pres_ind_base + fill.p1 + passPresEnding Pl P3 ;
    VAct VSim (VPres VInd)  n  p  => -- Present Indicative
      pres_ind_base +
      ( case pres_ind_base of {_ + #consonant => "i" ; _ => ""}
      ) + passPresEnding n p ;
    VAct VSim (VPres VConj) n  p  => -- Present Conjunctive
      pres_conj_base + passPresEnding n p ;
    VAct VSim (VImpf VInd)  n  p  => -- Imperfect Indicative
      impf_ind_base + passPresEnding n p ;
    VAct VSim (VImpf VConj) n  p  => -- Imperfect Conjunctive
      impf_conj_base + passPresEnding n p ;
    VAct VSim VFut          Sg P1 => -- Future I
      (init fut_I_base ) +
      ( case fut_I_base of { _ + "bi" => "o" ; _ => "a" }
      ) + passPresEnding Sg P1 ;
    VAct VSim VFut          Sg P2 => -- Future I
      ( case fut_I_base of {
          _ + "bi" => ( init fut_I_base ) + "e" ;
          _ => fut_I_base } ) + passPresEnding Sg P2 ;
    VAct VSim VFut          Pl P3 => -- Future I
      (init fut_I_base ) +
      (case fut_I_base of { _ + "bi" => "u" ; _ => "e" }) +
      passPresEnding Pl P3 ;
    VAct VSim VFut          n  p  => -- Future I
      fut_I_base + passPresEnding n p ;
    VAct VAnt (VPres _)  n  p | -- Prefect
    VAct VAnt (VImpf _)  n  p | -- Plusperfect
    VAct VAnt VFut          n  p  => -- Future II
      "######" -- Use participle
  } ;
```

Listing 3.26: Ausschnitt aus der Funktion `mkDeponent` um Aktiv-Verbformen zu bilden (vgl. **ResLat.gf**)

Den Anfang bilden wieder die Aktiv-Formen. Die Präsens-Formen sind geprägt von leichten Unterschieden zum Grundschema. So hängt die 1. Person Singular wieder von der Endung des Präsens-Stammes ab. Endet dieser auf *-a* so wird dieses entfernt bevor die Endung *-or* angefügt wird. Die Bildung der 2. Person ist dagegen von der Infinitiv-Form abhängig. Endet diese auf *-ri* bleibt der Präsens-Stamm unverändert. Andernfalls wird, wenn der Präsens-Stamm auf eine *-i* endet, dieses durch ein *-e* ersetzt, oder falls nicht, das *-e* einfach an den Präsens-Stamm angehängt. Schlussendlich folgt die 2.-Person-Singular-Passiv-Endung. Bei der 3. Person Singular wird die passende Endung entweder direkt an den Präsens-Stamm gehängt, außer dieser endet nicht auf ein *-a* oder *-e*, dann wird zwischen Stamm und Endung ein *-u-* eingefügt. Bei allen weiteren Präsens-Formen wird die Endung entweder direkt an den Stamm gefügt, oder es wird, wenn der Stamm auf einen Konsonanten endet, der Vokal *-i-* zwischen Stamm und Endung eingeschoben. Der Präsens-Konjunktiv und Imperfekt-Indikativ so wie Imperfekt-Konjunktiv haben wieder keine vom Grundschema, Stamm plus Endung, abweichenden Formen. Erst bei den Futur-I-Formen sind wieder Abweichungen zu finden. Bei der 1. Person Singular wird zunächst vom Futur-I-Stamm der letzte Buchstabe entfernt. War er Teil des Suffixes *-bi*, so wird er durch ein *-o-* ersetzt, sonst durch ein *-a-*. Zum Schluss wird die 1.-Person-Singular-Passiv-Endung angefügt. Bei der 2. Person Singular wird, wenn der Stamm auf das Suffix *-bi* endet, wieder das *-i* durch ein *-e-* ersetzt, bevor die entsprechende Endung eingesetzt wird. Und schließlich wird bei der 3. Person Plural der letzte Buchstabe des Stammes entweder durch ein *-u-* ersetzt, wenn der Stamm auf das Suffix *-bi* endet, bevor die Endung angefügt wird. Andernfalls wird er durch ein *-e-* ersetzt.[67]

Da alle weiteren Aktiv-Formen mit Hilfe des Partizips umschrieben werden, sind alle möglichen Aktiv-Formen beschrieben. Passiv-Formen kommen bei Deponentia naturgemäß nicht vor. Deshalb können diese Formen durch die Fehlerzeichenkette *######* ersetzt werden, um zu markieren, dass sie im Paradigma nicht vorhanden sind.

Die nächsten im Paradigma teilweise vorhandenen Formen sind die Infinitiv-Formen. Der Infinitiv-Präsens-Aktiv ist identisch zum gegebenen Infinitiv-Stamm. Die Infinitive für Perfekt-Aktiv und Futur-Aktiv müssen, wie auch bei den regulären Verben, wieder im Geschlecht mit dem Bezugswort übereinstimmen und bilden deshalb drei Formen. Diese basieren auf dem Partizip-Stamm an den jeweils die drei geschlechtsspezifischen Endungen angefügt werden. Bei dem Infinitiv-Perfekt-Aktiv sind das *-um*, *-am* und *-um*, bei dem Infinitiv-Futur-Aktiv sind es *-urum*, *-uram*

[67] vgl. [BL94] S. 84f.

```
1  inf =
2    table {
3      VInfActPres          => -- Infinitive Present Active
4        inf_pres ;
5      VInfActPerf Masc     => -- Infinitive Perfect Active
6        part_stem + "um" ;
7      VInfActPerf Fem      => -- Infinitive Perfect Active
8        part_stem + "am" ;
9      VInfActPerf Neutr    => -- Infinitive Perfect Active
10       part_stem + "um" ;
11     VInfActFut Masc      => -- Infinitive Future Active
12       part_stem + "urum" ;
13     VInfActFut Fem       => -- Infinitive Perfect Active
14       part_stem + "uram" ;
15     VInfActFut Neutr     => -- Infinitive Perfect Active
16       part_stem + "urum" ;
17     VInfPassPres         => -- Infinitive Present Passive
18       "######" ; -- no passive form
19     VInfPassPerf _       => -- Infinitive Perfect Passive
20       "######" ; -- no passive form
21     VInfPassFut          => -- Infinitive Future Passive
22       "######"  -- no passive form
23   } ;
```

Listing 3.27: Ausschnitt aus der Funktion `mkDeponent` um Infinitiv-Verbformen zu bilden (vgl. **ResLat.gf**)

und *-urum*. Die passiven Infinitiv-Formen entfallen wieder und werden durch die Fehlerzeichenkette ersetzt.[68]

```
imp =
  table {
    VImp1 Sg                => -- Imperative I
      ( case inf_pres of {
          _ + "ri" => imp_base ;
          _ => (init imp_base ) + "e"
        }
      ) + "re" ;
    VImp1 Pl                => -- Imperative I
      imp_base + "mini" ;
    VImp2 Sg ( P2 | P3 ) => -- Imperative II
      imp_base + "tor" ;
    VImp2 Pl P2             => -- Imperative II
      "######" ; -- really no such form?
    VImp2 Pl P3             => -- Imperative II
      pres_ind_base + fill.p1 + "ntor" ;
    _ => "######" -- No imperative form
  } ;
```

Listing 3.28: Ausschnitt aus der Funktion `mkDeponent` um Imperativ-Verbformen zu bilden (vgl. **ResLat.gf**)

Als nächstes folgen die Imperativ-Formen, von denen es den Imperativ I im Singular und Plural sowie den Imperativ II in der 2. und 3. Person Singular und der 3. Person Plural gibt. Die erste Form, der Imperativ I Singular entspricht, wenn der Infinitiv-Stamm auf *-ri* endet, einfach aus dem Imperativ-Stamm, besteht aber andernfalls aus dem Imperativ-Stamm bei dem der letzte Buchstabe durch ein *-e* ersetzt ist. Die Imperativ-I-Plural-Form besteht einfach aus dem Imperativ-Stamm mit der Endung *-mini* und die 2. und 3. Person des Imperativ II im Singular aus dem Stamm mit der Endung *-tor*. Die 3. Person des Imperativ II im Plural dagegen besteht aus dem Präsens-Indikativ-Stamm, dem Vokal *-u-*, wenn eben dieser Stamm nicht auf *-a* oder *-e* endet, und der Endung *-ntor*.[69]

Die Bildung des Gerunds ist relativ analog zur Bildung des Gerunds bei regulären Verben. Die vier Formen des Gerund werden aus dem Präsensstamm, dem Vokal *-e-*, wenn der Präsens-Indikativ-Stamm nicht auf ein *-a* oder *-e* endet, und den vier Endungen *-ndum*, *-ndi*, *-ndo* und *-ndo*.[70]

[68] vgl. [BL94] S. 86
[69] vgl. [BL94] S. 84f.
[70] vgl. [BL94] S. 86

```
1  ger =
2    table {
3      VGenAcc => -- Gerund
4        pres_stem + fill.p2 + "ndum" ;
5      VGenGen => -- Gerund
6        pres_stem + fill.p2 + "ndi" ;
7      VGenDat => -- Gerund
8        pres_stem + fill.p2 + "ndo" ;
9      VGenAbl => -- Gerund
10       pres_stem + fill.p2 + "ndo"
11   } ;
```

Listing 3.29: Ausschnitt aus der Funktion `mkDeponent` um Gerund-Verbformen zu bilden (vgl. **ResLat.gf**)

Das Gerundiv bildet wieder alle Formen eines drei-endigen Adjektivs der ersten und zweiten Deklination. Die drei Nominativ-Grundformen sind dabei, wie eben schon beim Gerund der Präsens-Stamm, unter den bereits genannten Bedingungen der Vokal *-e-* und den Endungen *-ndus*, *-nda* und *-ndum*.[71]

Als nächstes folgen die Partizipien. Obwohl bisher alle passiven Formen aus dem Paradigma gefallen sind, sind nun allerdings alle drei Partizipien vorhanden. Deshalb möchte ich an dieser Stelle nicht von aktiven und passiven Partizipien, sondern nur von Partizip Präsens, Partizip Perfekt und Partizip Futur sprechen. Diese werden allerdings genau so gebildet wie das Partizip-Präsens-Aktiv, das Partizip-Perfekt-Passiv und das Partizip-Futur-Aktiv bei den regulären Verben.[72]

Nun zur letzten Form des Deponens-Paradigmas, dem Supin. Dieses wird zum Abschluss auch wieder genau so gebildet, wie bei den regulären Verben. Damit ist auch die zweite größere Klasse lateinischer Verben komplett behandelt.[73]

Unregelmäßige Verben

Zusätzlich zur relativ großen Gruppe der Deponentia gibt es in der lateinischen Sprache noch einige andere unregelmäßige Verben. Diese bilden entweder stark vom simplen "Stamm plus Endung"-Schema abweichende Formen oder nur kleine Teile des Paradigmas, oder auch beides. Deshalb müssen sie etwas gesondert von den anderen Verben behandelt werden. Dies geschieht in einem eigenen Teil der Grammatik zur Behandlung unregelmäßiger Wortbildung, den Dateien **IrregLatAbs.gf** und **IrregLat.gf**. Bisher werden die Formen einiger sehr wichtiger Verben auf diese

[71] vgl. [BL94] S. 86
[72] vgl. [BL94] S. 86
[73] vgl. [BL94] S. 86

```
lin
  -- Bayer-Lindauer 93 1
  be_V =
    let
      pres_stem = "s" ; perf_stem = "fu" ; part_stem = "fut" ;
      pres_ind_base = "su" ; pres_conj_base = "si" ;
      impf_ind_base = "era" ; impf_conj_base = "esse" ;
      fut_I_base = "eri" ;
      imp_base = "es" ;
      perf_ind_base = "fu" ; perf_conj_base = "fueri" ;
      pqperf_ind_base = "fuera" ; pqperf_conj_base = "fuisse" ;
      fut_II_base = "fueri" ;
      verb = mkVerb "esse" pres_stem pres_ind_base pres_conj_base
               impf_ind_base impf_conj_base fut_I_base imp_base
               perf_stem perf_ind_base perf_conj_base
               pqperf_ind_base pqperf_conj_base fut_II_base
               part_stem ;
      in {
        act =
          table {
            VAct VSim (VPres VInd)  n  p  =>
              table Number
                [ table Person [ "sum" ; "es" ; "est" ] ;
                  table Person [ "sumus" ; "estis" ; "sunt" ]
                ] ! n ! p ;
            a => verb.act ! a
          };
        pass = \\_ => "######" ; -- no passive forms
        inf = verb.inf ;
        imp = table { VImp1 Sg => "es" ;
                      VImp1 Pl => "este" ;
                      VImp2 Pl P2 => "estote" ;
                      a => verb.imp ! a } ;
        sup = \\_ => "######" ; -- no supin forms
        ger = \\_ => "######" ; -- no gerund forms
        geriv = \\_ => "######" ; -- no gerundive forms
        part = table {
          VActFut => verb.part ! VActFut ;
          VActPres | VPassPerf =>
            \\_ => "######" -- no such participle
          }
      } ;
```

Listing 3.30: Regel um das Paradigma für unregelmäßige Verb *esse* zu erzeugen (vgl. **IrregLat.gf**)

Weise gebildet, so z.B. die Kopula *esse* (vgl. Listing 3.30). Das Vorgehen ist meist recht ähnlich. Zuerst werden die 9 oder 15 Wortstämme, je nachdem ob das Verb eher wie ein Deponens oder ein reguläres Verb gebildet wird, per Hand aufgelistet. Als nächstes wird entweder die Funktion `mkVerb` oder `mkDeponent` verwendet um aus den gegebenen Stämmen ein komplettes Paradigma zu erzeugen. Und als letzter Schritt werden in diesem kompletten Paradigma noch einmal Wortformen korrigiert, die falsch gebildet wurden und Wortformen entfernt, die im Zielparadigma nicht vorhanden sind. Auf diese Weise werden die Wörter *esse* (`be_V`), *posse* (`can_VV`), *ferre* (`bring_V`), *velle* (`want_V`), *ire* (`go_V`) und *fieri* (`become_V`) gebildet.[74]

Anders verhalten sich dagegen die so genannten *Verba impersonalia*. Sie kommen gewöhnlich nur in der 3. Person Singular oder im Infinitiv auf.[75] Deshalb kann man annehmen, dass das Verbparadigma für diese Verbart nur diese Formen enthält. Aus diesem Grunde ist es erheblich effizienter eben nur diese Formen aufzuzählen, statt ein komplettes Verbparadigma zu generieren und anschließend die nicht vorkommenden Formen zu eliminieren. Dieses Vorgehen wurde bei `rain_V0` (lat. *pluit*) angewandt.

3.2.4 Pronomenflexion

Ein Kapitel über Pronomenflexion im Sinne der vorhergehenden Kapitel über Nomen, Adjektive und Verben kann man aus mehreren Gründen als nicht unumstritten sehen. Zunächst einmal gehören Pronomen zu den geschlossenen Kategorien, also zu den Wortarten, zu denen nur wenig Wörter gehören. Schon deswegen stellt sich die Frage, ob der Aufwand gerechtfertigt ist, eine allgemeine Flexion für eine Klasse von Wörtern zu entwerfen, wenn nur recht wenige Wörter zu dieser Klasse gehören. Oder ob es einfacher wäre im Lexikon einfach alle Wortformen im entsprechenden Eintrag aufzulisten. Außerdem bilden die Pronomen keine wirklich homogene Wortart, sondern bestehen aus verschiedensten Unterklassen, die teilweise nach unterschiedlichen Merkmalen flektiert werden. So kann man die Pronomen untergliedern in Personalpronomen, Possesivpronomen, Demonstrativpronomen, Relativpronomen, Interrogativpronomen, Indefinitpronomen und ein paar mehr. Manche, wie die Personalpronomen, werden wie Nomen dekliniert, andere, wie die Possesivpronomen, werden wie Adjektive dekliniert.[76]

Es wurde deshalb ein Mittelweg gewählt. Denn für die Personal- und Possesivpro-

[74] vgl. [BL94] S. 105ff.
[75] vgl. [BL94] S. 111
[76] vgl. [BL94] S. 48ff.

nomen wurde eine möglichst allgemeine Flexionsfunktion implementiert. Für alle weiteren Klassen von Pronomen wurde dagegen die Lösung gewählt, alle Wortformen im Lexikon direkt aufzulisten. Dies wurde bereits im Abschnitt 3.1 erwähnt. Deshalb wird hier die Flexion von Personal- und Possesivpronomen erläutert.

Dafür wird ein Typ namens `Pronoun` definiert (vgl. Listing 3.4). Als feste Felde hat er zunächst Genus, Numerus und Person hat. Zusätzlich hat er zwei Tabellenfelder, eines für das diesem Genus, Numerus und Person entsprechende Personalpronomen und eines für das entsprechende Possesivpronomen. Die Form der Personalpronomen ist primär vom Kasus abhängig und Genus sowie Numerus sind fix. Possesivpronomen werden dagegen wie Adjektiv dekliniert, die Form ist also sowohl von Kasus, als auch von Genus und Numerus, abhängig. Deshalb sind für die Possesivpronomenform die festen Felder nicht von Bedeutung. Dafür hängen diese Pronomenarten von ein bis zwei weiteren Merkmalen ab. Zum einen, ob der Pro-Drop-Parameter gesetzt ist, also ob das Personalpronomen in der Subjektposition entfallen kann.[77] Und zum anderen bilden die Pronomen der 3. Person unterschiedliche Formen, abhängig davon, ob sie reflexiv verwendet werden oder nicht. Deshalb gibt es das Merkmal der Reflexivität. Hauptsächlich sind die Formen der Pronomen von Numerus und Person abhängig. Die Pro-Drop-Form der Personalpronomen ist immer der leere String, denn sie können alle in der Subjektposition entfallen, da die nötige Information bereits im Verb kodiert ist.

Die reguläre Personalformen des Pronomens in der ersten Person Singular sind die der Nomenflexion entsprechend die vom Kasus abhängigen Formen *ego* (Nominativ), *mei* (Genitiv), *mihi* (Dativ), *me* (Akkusativ) und *me* (Ablativ). Der Vokativ existiert bei Personalpronomen aus offensichtlichen Gründen nicht. Die possesiven Formen dagegen entsprechen den Formen eines Adjektivs der ersten und zweiten Deklination mit den Grundformen *meus, -a, -um*. Allerdings lauten die Vokativ-Formen *mi, mea, meum*. Bei Pronomen der 1. und 2. Person spielt die Reflexivität noch keine Rolle. In der 2. Person Singular lauten die regulären Personalformen entsprechend *tu*, *tui*, *tibi*, *te* und *te* und die possesiven Formen folgen wieder der ersten und zweiten Deklination bei den Grundformen *tuus, -a, -um*.

Bei der 1. Person Plural bildet das Personalpronomen die Formen *nos*, *nostri*, *nobis*, *nos* und *nobis* und die Formen des entsprechenden Possesivpronomens werden aus aus den Grundformen *noster, nostra, nostrum* gebildet. Die Formen der 2. Person Plural sind analog dazu *vos*, *vostri*, *vobis*, *vos* und *vobis* und beim Possesivpronomen werden sie aus den Grundformen *vester, vestra, vestrum* gebildet.

[77] vgl. [Glu04] S. 7585

```
<Sg,P1> =>
  <
  table {
    PronDrop    => \\_,_ => "" ;
    PronNonDrop => \\_ =>
      pronForms "ego" "me" "mei" "mihi" "me" "me"
  },
  \\_ => table {
    Ag Masc Sg c =>
      ( pronForms "meus" "meum" "mei" "meo" "meo" "mi" )
        ! c ;
    Ag Masc Pl c =>
      ( pronForms "mei" "meos" "meorum" "meis" "meis" "mei" )
        ! c ;
    Ag Fem Sg c =>
      ( pronForms "mea" "meam" "meae" "meae" "mea" "mea" )
        ! c ;
    Ag Fem Pl c =>
      ( pronForms "meae" "meas" "mearum" "meis" "meis" "meae" )
        ! c ;
    Ag Neutr Sg c =>
      ( pronForms "meum" "meum" "mei" "meo" "meo" "meum" )
        ! c ;
    Ag Neutr Pl c =>
      ( pronForms "mea" "mea" "meorum" "meis" "meis" "mea" )
        ! c
  }
  > ;
```

Listing 3.31: Ausschnitt aus der Funktion `mkPronoun` für 1. und 2. Person Singular (vgl. **ResLat.gf**)

In der 3. Person Singular und Plural gibt es nun mehr Formen. Zum einen werden unterschiedliche Formen bei reflexivem und irreflexivem Gebrauch verwendet. Zum anderen unterscheiden sich bei der 3. Person auch die irreflexiven Personalpronomen-Formen nach Geschlecht. Dafür entfallen eben diese irreflexiven Formen bei den Possesivpronomen ganz. Also ergibt sich für die 3. Person Singular der Personalpronomen folgendes Formenschema: Bei den irreflexiven, maskulinen Formen *is*, *eius*, *ei*, *eum* und *eo*, bei femininen *ea*, *eius*, *ei*, *eam* und *ea* und bei Neutra *id*, *eius*, *ei*, *id* und *eo*. Bei den reflexiven Formen fallen wieder alle drei Geschlechter zusammen. Zusätzlich fehlt eine Nominativ-Form. Die verbleibenden Formen sind *sui* im Genitiv, *sibi* im Dativ und *se* sowohl im Akkusativ als auch im Ablativ. Die irreflexiven Possessiv-Formen der 3. Person existieren eigentlich nicht, werden aber durch die Genitiv-Singular- bzw. Plural-Form von *is, ea, id* nämlich *eius* im Singular und *eorum, -a, -um* im Plural ersetzt. Dafür sind die reflexiven Formen dem Schema entsprechend, sowohl im Singular als auch im Plural, wieder analog zu Adjektivformen mit der Grundform *suus, -a, -um*.[78]

Nachdem hier die Formenbildung aller wichtigen Wortarten für das Lateinische beschrieben ist, sind alle Bestandteile vorhanden um aus den Wörtern und Wortformen der Lexikoneinträge größere Einheiten bilden zu können. Dies geschieht in Form von Syntaxregeln, deren Aufbau der letzte Teil der Grammatik gewidmet ist.

[78] vgl. [BL94] S. 48ff.

3.3 Syntax

Den letzten Teil dieser Arbeit bildet der Bereich der Syntax, um aus den einfachen Einheiten aus dem Lexikon komplexe Einheiten von einfacheren Phrasen bis hin zu kompletten Sätzen zu bilden. Zunächst wird erläutert, wie einfache Phrasen konstruiert werden. Und diese Phrasen wiederum werden weiter kombiniert, um größere Ausdrücke zu erzeugen, bis hin zur Satzebene, bzw. zu den Äußerungen (Utterances, `Utt`) in der Nomenklatur des Grammatical Frameworks.

3.3.1 Nominalphrasen

Als erstes sollen nun die verschiedenen Möglichkeiten beschrieben werden, eine Nominalphrase zu konstruieren. Dies geschieht mit Hilfe von Regeln die in der Datei **NounLat.gf** zu finden sind.

```
1  lin
2    DetCN det cn = -- Det -> CN -> NP
3      {
4        s = \\c =>
5          det.s ! cn.g ! c ++
6          cn.preap.s ! (Ag cn.g det.n c) ++
7          cn.s ! det.n ! c ++
8          cn.postap.s ! (Ag cn.g det.n c) ;
9        n = det.n ;
10       g = cn.g ;
11       p = P3 ;
12     } ;
```

Listing 3.32: Die Syntaxregel `DetCN`, um ein Determinans und ein Common Nouns zu einer `NP` zu verbinden (vgl. **NounLat.gf**

Die erste Regel namens `DetCN` gibt an, dass ein Determinans zusammen mit einer Phrase von Typ `CN` (Common Noun) zu einer Nominalphrase verbunden werden kann. Allerdings haben wir bisher noch nicht besprochen, wie Common Nouns aufgebaut sind und wie man sie erzeugt. Common Nouns sind, möglicherweise durch Adjektive attribuierte, Nomen. Deshalb sind sie im Grunde aufgebaut wie Nomen, haben also ein Feld für das inhärente Genus und ein `s`-Feld für die Wortformen. Zusätzlich hat ein Common Noun aber auch zwei Felder, in denen Adjektive bzw. genauer Adjektivphrasen, die das Nomen näher beschreiben, abgelegt werden können. Da für Adjektive in der lateinischen Sprache keine klaren Regeln herrschen, ob

sie vor oder nach dem Nomen stehen, auf das sie Bezug nehmen, gibt es für jede der beiden Positionen je ein Feld, in das Adjektive flexibel angefügt werden können.[79]

Adjektivphrasen und Adjektiv können in diesem Falle synonym verwendet werden, denn Adjektivphrasen unterscheiden sich aktuell nur soweit von Adjektiven, dass sie statt verschiedener Steigerungsformen nur in der Positiv-Form vorkommen. Das heißt, um aus einem Adjektiv eine Adjektivphrase zu erzeugen werden lediglich die Positiv-Formen des Adjektivs ausgewählt.[80] Adjektivphrasen können allerdings auch per Konjunktion verbunden werden um daraus wieder eine neue Adjektivphrase zu bilden. Dies geschieht mit Hilfe der Regel `ConjAP` in der Datei **ConjunctionLat.gf**. Sie erzeugt aus einer Konjunktion wie `and_Conj` oder `or_Cat` und einer Liste von Adjektivphrasen eine Adjektivphrase.

```
1 lincat
2   [AP] = {s1,s2 : Agr => Str } ;
3 lin
4   ConjAP conj ss =
5     conjunctDistrTable Agr conj ss ;
6   BaseAP x y =
7     lin A ( twoTable Agr x y ) ;
8   ConsAP xs x =
9     lin A ( consrTable Agr and_Conj.s2 xs x );
```

Listing 3.33: Der Listentyp Für Adjektivphrasen und die Funktionen zum Erstellen und Verwenden (vgl. **ConjunctionLat.gf**

Dazu muss kurz erklärt werden, wie Listen im Grammatical Framework erzeugt werden könne. Im Grammatical Framework werden Listentypen als Typen in eckigen Klammern geschrieben. So ist `[AP]` eine Liste von Objekten des Typs `AP`. Die Liste der Adjektivphrasen ist als ein Verbund von zwei Feldern definiert. Das zweite davon enthält das letzte Element der Liste und das erste alle davor befindlichen Elemente, die schon durch Konjunktionen verbunden sind. Diese Listen werden mit folgenden zwei Regeln aufgebaut. Die Regel `BaseAP` erzeugt eine zweielementige Liste aus zwei Adjektivphrasen. Dazu wird eine interne Funktion namens `twoTables`[81] verwendet, die den Verbund erstellt und den richtigen Typ festlegt. Die zweite Listenoperation ist die Funktion `ConsAP`, die an eine bestehende Liste ein Element anfügt. Dazu wird das ehemals letzte Element zunächst an den Rest angehängt. Zur Verbindung wird eine fixe Zeichenkette verwendet, in diesem Falle pauschal die

[79] vgl. [BL94] S. 118
[80] vgl. Regel `PositA` in **AdjectiveLat.gf**
[81] vgl. **Coordination.gf** in **lib/src/prelude**

Konjunktion `and_Conj`. Mit diesen Regeln können Adjektivphrasen erstellt werden, die aus mehreren einzelnen `AP`s bestehen, die mit Konjunktionen verbunden sind. Die beiden Funktionen `consrTable` und `conjunctDistrTable` sind wie `twoTable` Hilfsfunktionen, die direkt vom Grammatical Framework bereitgestellt werden um die Listenhandhabung zu ermöglichen.

```
lin
  UseN n = -- N -> CN
    lin CN ( n ** {preap, postap = {s = \\_ => "" } } ) ;
  UseN2 n2 = -- N2 -> CN
    lin CN ( n2 ** {preap, postap = {s = \\_ => "" } } ) ;
```

Listing 3.34: Die Syntaxregeln `UseN` und `UseN2` um ein Nomen als `CN` zu verwenden (vgl. **NounLat.gf**

```
param
  AdjPos = Pre | Post ;
lin
  AdjCN ap cn =  -- AP -> CN -> CN
    let pos = variants { Post ; Pre }
    in
    {
      s = cn.s ; g = cn.g
      postap = case pos of {
        Pre => cn.postap ;
        Post =>
          { s = \\a => ap.s ! a ++ cn.postap.s ! a } } ;
      preap = case pos of {
        Pre =>
          { s = \\a => ap.s ! a ++ cn.preap.s ! a } ;
        Post => cn.preap } ;
    } ;
```

Listing 3.35: Die Syntaxregeln `AdjCN` um ein Common Noun mit einer Adjektivphrase zu erweitern (vgl. **NounLat.gf**

Die einfachste Möglichkeit Common Nouns zu bilden ist es, einfach Nomen als `CN`s zu verwenden. Dazu gibt es die beiden Regeln `UseN` und `UseN2`. Sie erweitern die beiden Nomentypen `N` und `N2` lediglich um die beiden leeren `AP`-Felder um sie in Common Nouns zu verwandeln. Diese Common Nouns können dann durch Adjektive genauer bestimmt werden. Dazu wird in der Regel `AdjCN` über freie Variation ausgewählt, ob die modifizierende Adjektivphrase vor oder hinter das Common Noun

eingefügt wird. Entsprechend der so gewählten Position wird die `AP` in eines der beiden vorgesehenen Felder eingefügt. Das `s`-Feld für die Nomen-Formen so wie das Nomen-Genus werden dabei einfach übernommen.

```
lin
  NumSg = { s = \\_,_ => [] ; n = Sg } ;
  NumPl = { s = \\_,_ => [] ; n = Pl } ;
  DefArt = { s = \\_ => [] ; sp = \\_ => [] } ;
  IndefArt = { s = \\_ => [] ; sp = \\_ => [] } ;
  DetQuant quant num = {
    s  = \\g,c =>
      quant.s  ! Ag g num.n c ++ num.s ! g ! c ;
    sp = \\g,c =>
      quant.sp ! Ag g num.n c ++ num.s ! g ! c ;
    n  = num.n
  };
```

Listing 3.36: Die Syntaxregeln `DetQuant` um aus einem `Num`-Element und einem Quantifikator ein `Det` zu erzeugen (vgl. **NounLat.gf**

Als nächstes fehlen für die `DetCN`-Regel die Determinantia. Einige Wörter vom Typ `Det` sind im Lexikon definiert. Die Determinantia, die in den meisten Sprachen am häufigsten vorkommen, sind dort allerdings nicht zu finden, der bestimmte und der unbestimmte Artikel. Stattdessen werden sie direkt bei den Regeln für die Nominalphrasen definiert. Allerdings existieren diese in der lateinischen Sprache im üblichen Sinne überhaupt nicht. Deswegen sind sie Quantifikatoren mit leeren Zeichenketten. Quantifikatoren können mit Hilfe der Regel `DetQuant` zu einem Determinans umgewandelt werden. Dazu wird zusätzlich ein so genanntes „number determining element“, kurz `Num`, als Marker für den Numerus verwendet. Denn in verschiedenen Grammatiktheorien, unter anderem der Theorie der Transformationsgrammatik, gilt der Numerus als festes Merkmal des Determinans, das das Nomen in diesem Merkmal regiert[82]. So erhält man für die bestimmten und unbestimmten Artikel vier verschiedene Objekte der Kategorie `Det`, die alle die leere Zeichenkette erzeugen, und von denen zwei das Merkmal des Singular und zwei den Plural tragen. Bestimmte und unbestimmte Artikel sind dabei nicht wirklich zu unterscheiden.

Damit haben wir alle nötigen Bestandteile für die `DetCN`-Regel (Listing 3.32). Diese Regel funktioniert nun folgendermaßen. Im `s`-Feld ist die Zeichenkette enthalten, die durch die Phrase erzeugt werden kann. Da diese weiterhin vom Kasus abhängig ist, wird eine Tabelle erzeugt, wobei der später zur Auswahl genutzte Wert in der Va-

[82] vgl. [Glu04] S. 6706

riable `c` zur Verfügung steht. Diese Zeichenkette besteht aus der Determinans-Form, die von Genus und Kasus abhängt. Das Genus wird vom `g`-Feld des Common Noun bestimmt, der Kasus stammt aus der Variable `c`. Auf die so entstandene Form folgen die Adjektivphrasen, die vor dem Nomen platziert sind. Diese stammen aus dem Feld `preap` des Common Nouns und müssen mit dem Nomen in Genus, Numerus und Kasus übereinstimmen. Dazu wird, unter anderem um keine dreifach geschachtelte Tabelle zu benötigen, ein so genannter abhängiger Typ für die Kongruenz zwischen Nomen und Adjektiven verwendet. Der Typ heißt `Agr` für Agreement, also Übereinstimmung, und besteht aus einem Konstruktor, also einer Art Schlüsselwort wie hier `Ag` und einer Liste von Typen. In diesem Falle sind das Genus, Numerus und Kasus. Als nächstes folgt die Nomenform, die den Numerus des Determinans übernimmt, so wie den Kasus aus `c`. Als letztes folgen die Adjektivphrasen nach dem Nomen aus dem Feld `postap` analog zu `preap`. Durch die Verkettung dieser Bestandteile wird die Zeichenkette gebildet, die der Nominalphrase entspricht. Diese Nominalphrase behält den Numerus des Artikels so wie das Genus des Common Nouns als inhärente Merkmale. Zusätzlich hat sie als Person die 3. Person, die die Verbform beeinflusst. Dies ist die komplexeste der bisher implementierten Regeln um Nominalphrasen zu erzeugen.

```
1 lin
2   UsePN pn =
3     lin NP { s = pn.s ! Sg ; g = pn.g ; n = Sg ; p = P3 } ;
```

Listing 3.37: Die Syntaxregel `UsePN`, um einen Eigennamen als `NP` zu verwenden (vgl. **NounLat.gf**

Daneben gibt es noch zwei kurze Regeln um Nominalphrasen aus Eigennamen und Personalpronomen zu erzeugen. Die erste namens `UsePN` hat die selbe Form wie der verwendete Name im Singular. Entsprechend ist der Genus der `NP` gleich dem Genus des Eigennamens, der Numerus ist Singular und die Person wieder die 3. Person.

Und die letzte Regel `UsePron` verwandelt ein Pronomen in eine Nominalphrase. Dazu werden Genus, Numerus und Person aus dem Pronomen übernommen. Bei der Kasus-abhängigen Form kommt dafür das Pro-Drop-Phänomen zu tragen. Denn wenn die aus einem Pronomen gebildete Nominalphrase im Nominativ verwendet wird, entfällt sie, es wird also nur die leere Zeichenkette produziert. In allen anderen Kasus wird einfach die entsprechende Pronomenform gebildet. Soll allerdings auch in der Subjektposition die Pronomenform erzwungen werden, existiert im Extra-

```
lin
  UsePron p = -- Pron -> Np
  {
    g = p.g ; n = p.n ; p = p.p ;
    s = \\c => case c of {
      -- Drop pronoun in nominative case
      Nom => p.pers ! PronDrop ! PronRefl ;
      -- but don't drop it otherwise
      _ => p.pers ! PronNonDrop ! PronRefl
    } ! c ;
  } ;
```

Listing 3.38: Die Syntaxregel `UsePN` um ein Personalpronomen als `NP` zu verwenden (vgl. **NounLat.gf**)

Teil[83] der Grammatik eine Funktion namens `UsePronNonDrop`, die aus der Nicht-Pro-Drop-Form des Pronomens eine Nominalphrase erzeugt. Der Extra-Teil einer Grammatik kann im Grammatical Framework verwendet werden um für eine einzelne Sprache spezielle Konstrukte zu beschreiben. Diese können zwar beim Parsen verwendet werden, nicht jedoch beim Übersetzen, wenn die Zielsprache nicht auch die selbe erweiterte abstrakte Syntax hat. Allerdings können bei nahe verwandten Sprachen möglicherweise auch die Extra-Konstrukte aufeinander abgebildet werden, so dass auch zwischen diesen Sprachen übersetzt werden kann.

Diese drei Regeln bieten die grundlegenden Möglichkeiten um in einer Sprache Nominalphrasen zu bilden, wie sie in einfachen Sätzen Verwendung finden. In der abstrakten Syntax existieren noch fünf weitere Regeln, wovon eine einzige wirklich eine neue `NP` erzeugt. Die restlichen Regeln modifizieren lediglich die Nominalphrasen. Es verbleiben also diese Regeln für mögliche zukünftige Arbeiten.

3.3.2 Verbalphrasen

Die zweite Gruppe von wichtigen Phrasen in dieser Grammatik sind die Verbalphrasen. Denn die Sorte einfacher Sätze, die den Abschluss dieser Arbeit bilden, werden durch die Kombination von einer Nominalphrase mit einer Verbalphrase gebildet. Nachdem wir schon wissen, wie Nominalphrasen gebildet werden können, soll nun die Konstruktion von Verbalphrasen in dieser Grammatikimplementierung folgen.

Die einfachste Form einer Nominalphrase wird aus einem intransitiven Verb gebildet, also einem Verb, dass kein Objekt benötigt. Die entsprechende Regel heißt `UseV`

[83] **ExtraLatAbs.gf** und **ExtraLat.gf**

und benutzt lediglich eine Hilfsfunktion namens `predV`. Diese Funktion baut die Datenstruktur, die die für eine Verbalphrase nötigen Felder enthält (Listing 3.39). Dies

```
1 oper
2   VerbPhrase : Type = {
3     fin : VActForm => Str ;
4     inf : VInfForm => Str ;
5     obj : Str ;
6     adj : Agr => Str
7   } ;
```

Listing 3.39: Datenstruktur für Verbalphrasen (vgl. **ResLat.gf**)

sind finite Verbformen, infinite Verbformen, wenn vorhanden ein Objekt und möglicherweise auch ein modifizierendes Adjektiv. Die Typen `VActForm` und `VInfForm` haben dabei die selben möglichen Werte wie bei den Verben. Die Funktion `predV` füllt das `fin`-Feld mit den aktiven Verbformen und das `inf`-Feld mit den Infinitivformen des Verbs. Die Felder für das Objekt und das Adjektiv werden mit leeren Zeichenketten gefüllt. Damit ist die Verbalphrase auch schon erzeugt.

Für transitive Verben ist auf dem Weg zur Verbalphrase noch ein Zwischenschritt nötig. Denn aus einem transitiven Verb wird zunächst eine `VP` erzeugt, der eine Nominalphrase in der Objektposition fehlt. Diese Kategorie heißt analog zur Nomenklatur im GPSG-Grammatikformalismus VPSlash (in der GPSG-Notation VP/NP).[84] Solch eine `VPSlash` besteht lediglich aus einer, um ein zusätzliches Feld erweiterte, Verbalphrase. In diesem zusätzlichen Feld kann, wenn nötig eine Präposition gespeichert werden, die unter anderem bestimmt, in welchem Kasus das Objekt des Verbs stehen muss. Diese bei transitiven Verben nötige Präposition ist üblicherweise im Lexikoneintrag des Verbs zu finden oder ein Akkusativobjekt wird angenommen. Die Transformation eines transitiven Verbs in eine `VPSlash` erfolgt in der Regel `SlashV2a` analog zu den intransitiven Verben mit einer Funktion namens `predV2`, wobei der Typ `V2` der Typ für transitive Verben ist. Diese Funktion macht nichts anderes um mit Hilfe von `predV` eine `VP` zu erzeugen, um die Präposition des transitiven Verbs erweitern und den Typ auf `VPSlash` zu ändern.

Um nun aus solch einer `VPSlash` eine vollständige Verbalphrase zu erzeugen, kann die Regel `ComplSlash` verwendet werden. Sie kombiniert eine `VPSlash` zusammen mit einer Nominalphrase zu einer `VP`. Dazu werden zwei Hilfsfunktionen verwendet. Die erste, `appPrep` verbindet die in der `VPSlash` hinterlegte Präposition mit der Objekt-`NP` indem sie die Präposition vor die Form der Nominalphrase im richtigen

[84] vgl. [Ran11] S. 217

```
oper
  appPrep : Preposition -> (Case => Str) -> Str =
    \c,s -> c.s ++ s ! c.c ;
lin
  ComplSlash vp np = -- VPSlash -> NP -> VP
    insertObj (appPrep vp.c2 np.s) vp ;
```

Listing 3.40: Die Syntaxregel `ComplSlash` um eine `VPSlash` und eine `NP` zu einer Verbalphrase zu verbinden (vgl. **NounLat.gf** und **ResLat.gf**)

Kasus setzt. Die so entstandene Zeichenkette wird dann in der Funktion `insertObj` in das Objekt-Feld der Verbalphrase, vor möglichen weiteren Objekten, eingefügt.

3.3.3 Einfache Sätze

Nachdem wir uns in den vorangegangenen Abschnitten mit dem Aufbau einzelner Phrasen in der lateinischen Syntax beschäftigt haben, können wir uns nun letztendlich dem Aufbau ganzer Sätze widmen. Das die Wortstellung in der lateinischen Sprache nicht unproblematisch ist, wurde nun schon des öfteren angesprochen. Doch wie die Situation wirklich ist, ist z.B. bei Bamman und Crane ([BC06]) zu finden.

In diesem Paper, das sich primär mit der Erstellung einer Treebank für lateinische Texte beschäftigt, sind an Hand der Treebank einige empirische Werte für die Wortstellung in lateinischen Sätzen aufgeführt. Das dazu untersuchte Korpus enthält Textauszüge aus dem Werk vier verschiedener Autoren, nämlich Cicero (ca. 63 v. Chr.), Caesar (ca. 51 v. Chr.), Vergil (ca. 19 v. Chr.) und Hieronymus (ca. 405 n. Chr.). Damit werden sowohl verschiedenste Epochen und verschiedene Stilrichtigungen abgedeckt. So sind Caesar und Hieronymus Autoren relativ einfacher Prosa, Cicero ein bekannter Redner und Rhetoriker und Vergil ein kunstvoller Dichter. Insgesamt umfasst das Korpus 12098 Wörter wovon auf Cicero 1189, auf Caesar 1486, auf Vergil 2647 und auf Hieronymus 6776 Wörter entfallen.[85] Bei der lateinischen Wortstellung geht man davon aus, dass allgemein die Reihenfolge Subjekt-Objekt-Verb bevorzugt wird. Dies verändert sich jedoch hin zur Folge Subjekt-Verb-Objekt, wie sie häufig bei modernen romanischen Sprachen zu finden ist. Caesar verwendet vorwiegend die für das klassische Latein zu erwartende Wortstellung Subjekt-Objekt-Verb (64.7%), Cicero dagegen benutzt am häufigsten die Wortstellung Objekt-Subjekt-Verb (52.6%), die allgemein dazu dient, das Objekt zu betonen. Vergil als Lyriker verwendet alle möglichen Wortstellungen relativ häufig

[85] [BC06] S. 71f.

um unter anderem der gewünschten lyischen Form Rechnung zu tragen, am häufigsten jedoch wie bei Cicero die Wortstellung Objekt-Subjekt-Verb (25.0%). Und schließlich Hieronymus verwendet bereits vornehmlich die Wortfolge Subjekt-Verb-Objekt (68.5%). Betrachtet man zusätzlich diejenigen Sätze ohne Subjekt, also die Sätze bei denen ein Pronomen in der Subjektposition entfallen ist, so stellt man fest, dass die klassischeren Autoren Caesar, Cicero und Vergil klar die Wortstellung Objekt-Verb bevorzugten. Nur Hieronymus verwendet meistens die Wortstellung Verb-Objekt.[86]

Wie man an diesen Beispielen sehen kann, wurden selbst in einem recht kurzen Zeitraum viele verschiedene Wortstellungen in Sätzen verwendet, zumindest in literarischen Texten. Deshalb müssen im Grunde in der Grammatik sechs verschiedene Wortstellungen ermöglicht werden: Subjekt-Verb-Objekt, Subjekt-Objekt-Verb, Verb-Subjekt-Objekt, Verb-Objekt-Subjekt, Objekt-Subjekt-Verb und Objekt-Verb-Subjekt. Dies wurde so auch umgesetzt, wobei üblicherweise nur die Wortstellung Subjekt-Objekt-Verb erzeugt wird. Die anderen Wortstellungen können möglicherweise in einer zukünftigen Version der Grammatik über die Extra-Grammatik verwendet werden.

In den Ressource Grammars des Grammatical Frameworks gibt es drei verschiedene Kategorien für Sätze, bei denen unterschiedliche Merkmale fixiert oder variabel sind. Bei Clauses (`Cl`) sind noch alle Tempusmerkmale, die Polarität, also ob ein Satz verneint oder nicht geäußert wird, und im Lateinischen die Wortstellung variabel. Bei Sentences (`S`), also auf der Satzebene werden die Wortstellung, die Tempusmerkmale und die Polarität festgelegt. Dabei ist die Wortstellung bisher fix und die anderen Merkmale können über Parameter gesetzt werden. Die nächste Stufe der Utterances `Utt` verhält sich an sich wie Sentences kann aber mit einer Konjunktion und einem Vokativ zu einer Phrase (`Phr`) erweitert werden. Phrases sind die übliche Startkategorie für Sätze im Grammatical Framework. Dabei werden Konjunktion und Vokativ dadurch optional, dass Objekte der entsprechende Typen gegeben sind, die die leere Zeichenkette erzeugen.

```
1 lincat
2   Cl = { s : Tense => Anteriority \
3       => Polarity => Order => Str } ;
```

Listing 3.41: Typ von Clauses (vgl. **CalLat.gf**)

Der Typ eines Clauses ist eine `Str`-Tabelle, die von `Tense`, `Anteriority`, `Polarity`

[86] [BC06] S. 74

und `Order` abhängig ist. Dabei sind `Tense`, `Anteriority` und `Polarity` gemeinsame Parameter vieler Sprachen und deshalb auch unabhängig von diesen definiert.[87]. Der Parameter `Order` entspricht den oben genannten Wortstellungen (`Order = SVO | VSO | VOS | OSV | OVS | SOV` vgl. **ResLat.gf**).

Erzeugt werden Clauses durch die Regel `PredVP` (vgl. Listing 3.42), die eine `NP` und eine `VP` zu einer Clause kombiniert. Die Kombination der Bestandteile erfolgt logischerweise bei den verschiedenen Wortstellungen in unterschiedlicher Reihenfolge. Das Vorgehen wird allerdings der Einfachheit halber nur am Beispiel der Wortstellung Subjekt-Objekt-Verb gezeigt.

Zunächst wird eine, von allen nötigen Merkmalen abhängige Tabelle gebildet, die nach Auswahl aller Merkmale eine Zeichenkette liefert. Diese Zeichenkette ist folgendermaßen aufgebaut: Aus der Subjekt-`NP`, die unter dem Namen `np` zu finden ist, wird die Nominativ-Form verwendet und an diese wird die im `obj`-Feld der `VP` gespeicherte Objektform angehängt. Als nächstes wird, wenn der Satz negiert ist, der Negationspartikel *non* eingefügt. Dies geschieht mit Hilfe der Funktion `negation` aus **ResLat.gf**, die wenn der Wert für `pol` negativ ist, die entsprechende Zeichenkette, und wenn `pol` positiv ist, die leere Zeichenkette zurück gibt. Für den Fall, dass ein Adjektiv prädikativ gebraucht wird, ist dieses Adjektiv in der Verbalphrase im Feld `adj` abgelegt, und wird als nächstes in die Zeichenkette eingefügt. Dazu wird die Singular-Nominativ-Form des Adjektivs im Genus des Subjekts gewählt. Und zum Schluss folgt das Verb in der passenden Form. Dazu werden die Tabellenwerte für die Tempusmerkmale so wie der Numerus und das Genus des Subjekts verwendet. Da für die Verbformen eigene Typen für die Tempusmerkmale verwendet werden, die von denen in **ParamX.gf** unterschiedlich sind, müssen sie mit den Hilfsfunktionen `anteriorityToVAnter` und `tenseToVTens`[88] entsprechend umgewandelt werden. Das hat historische Gründe, denn die Verb-Merkmale wurden von den vorhergehenden Arbeiten an der Grammatik übernommen, die allerdings aus einem früheren Stand der Ressource Grammar Library zu stammen scheinen. Als auf Satzebene die Merkmale aus `ParamX` nötig waren, wurde der Einfachheit halber die beschriebene Abbildung zwischen den Merkmalen implementiert. In Zukunft sollten diese Merkmale möglichst vereinheitlicht werden. Für die anderen Wortstellungen erfolgt die Zusammensetzung der Zeichenkette analog, jedoch in anderer Reihenfolge der einzelnen Komponenten.[89]

Damit ist die erste Stufe auf dem Weg einer komplette Phrase abgeschlossen. Als

[87] vgl. **ParamX.gf** im Ordner **lib/src/common/**
[88] vgl. **ResLat.gf**
[89] vgl. **SentenceLat.gf**

```
lin
  PredVP np vp = -- NP -> VP -> Cl
  {
    s = \\tense,anter,pol,order =>
      case order of {
        SVO =>
          np.s ! Nom ++ negation pol ++
          vp.adj ! Ag np.g Sg Nom ++
          vp.fin ! VAct ( anteriorityToVAnter anter )
                        ( tenseToVTense tense )
                        np.n np.p ++
          vp.obj ;
        VSO => negation pol ++ vp.adj ! Ag np.g Sg Nom ++
          vp.fin ! VAct ( anteriorityToVAnter anter )
                        ( tenseToVTense tense )
                        np.n np.p ++
          np.s ! Nom ++ vp.obj ;
        VOS => negation pol ++ vp.adj ! Ag np.g Sg Nom ++
          vp.fin ! VAct ( anteriorityToVAnter anter )
                        ( tenseToVTense tense )
                        np.n np.p ++
          vp.obj ++ np.s ! Nom ;
        OSV => vp.obj ++ np.s ! Nom ++ negation pol ++
          vp.adj ! Ag np.g Sg Nom ++
          vp.fin ! VAct ( anteriorityToVAnter anter )
                        ( tenseToVTense tense )
                        np.n np.p ;
        OVS => vp.obj ++ negation pol ++
          vp.adj ! Ag np.g Sg Nom ++
          vp.fin ! VAct ( anteriorityToVAnter anter )
                        ( tenseToVTense tense )
                        np.n np.p ++
          np.s ! Nom ;
        SOV => np.s ! Nom ++ vp.obj ++ negation pol ++
          vp.adj ! Ag np.g Sg Nom ++
          vp.fin ! VAct ( anteriorityToVAnter anter )
                        ( tenseToVTense tense )
                        np.n np.p
      }
  } ;
```

Listing 3.42: Syntaxregel `PredVP` um eine `NP` und eine `VP` zu einer `Cl` zu kombinieren (vgl. **SentenceLat.gf**)

nächstes Folgt die Umwandlung einer Clause in einen Sentence. Diese Umwandlung wird in der Regel `UseCl` beschrieben. Zusätzlich zu einem `Cl`-Objekt werden ein Objekt vom Typ `Temp` und ein Objekt vom Typ `Pol` benötigt. `Temp` ist eine gemeinsame Datenstruktur, die `Tense` im `t`-Feld, `Anteriority` im `a`-Feld und eine mögliche Zeichenkette im `s`-Feld zusammenfasst. Ebenso enthält `Pol` die `Polarity` in einem `p`-Feld und eine Zeichenkette im `s`-Feld[90]. Die Funktion `UseCl` fasst diese Parameter nun so zusammen, dass sie das `s`-Feld des `Temp`-Parameter mit dem `s`-Feld des `Pol`-Parameters und dem `s`-Feld der Clause verbindet. Anzumerken ist, dass die beiden `s`-Felder üblicherweise leer sind. Also wird aus der Clause die passende Form ausgewählt, in dem `Tense` und `Anteriority`, also das `t`- und `a`-Feld aus der Variable `t`, die das `Temp`-Objekt enthält, verwendet werden. Anschließend wird anhand der Polarität aus dem `Pol`-Objekt ausgewählt und schließlich die Wortstellung `SOV` festgelegt.[91]

Nachdem auch dieser Schritt getan ist, sind nur noch zwei weitere Regeln nötig, bis Objekte vom Typ `Phr` erzeugt werden können. Zunächst muss es möglich sein Sentences in Utterances zu transformieren. Dies geschieht in der Regel `UttS` allein durch die explizite Änderung des Typs mit Hilfe des Schlüsselworts `lin`.

```
1 lin
2   -- PConj -> Utt -> Voc -> Phr ;
3   PhrUtt pconj utt voc =
4     {s = pconj.s ++ utt.s ++ voc.s} ;
5   NoPConj = {s = []} ;
6   PConjConj conj = {s = conj.s2} ; --
7   NoVoc = {s = []} ;
8   VocNP np = {s = np.s ! Voc} ;
```

Listing 3.43: Syntaxregeln um eine Utterance mit einer Konjunktion und einem Vokativ zu einer Phrase zu kombinieren (vgl. **PhraseLat.gf**)

Die Regel `PhrUtt`, um aus einer Utterance ein `Phr`-Objekt zu erzeugen, bedarf etwas größeren Aufwands. Denn sie erweitert unter Umständen die Utterance noch um eine Konjunktion am Satzbeginn und am Schluss eine Nominalphrase im Vokativ. Dazu benötigt sie zwei zusätzliche Parameter, der eine vom Typ `PConj` und der andere vom Typ `Voc`.

Zum einen sind dafür die parameterlosen Funktionen `NoPConj` und `NoVoc` definiert, die einen leeren Platzhalter für diese Typen erzeugen, also Objekte, die nur die leere Zeichenkette erzeugen. Des weiteren gibt es die Regel `PConjConj` die ein

[90] vgl. **CommonX.gf** im Ordner **lib/src/common**
[91] vgl. **SentenceLat.gf**

`PConj`-Objekt[92] aus einer normalen Konjunktion erzeugt, indem es die Zeichenkette aus dem `s2`-Feld der Konjunktion, die die Zeichenkette enthält, die bei normaler Verwendung zwischen die zwei zu verbindenden Phasen gesetzt wird. Und für das Anfügen des Vokativs gibt es die Regel `VocNP`, die eine `NP` in ein `Voc`-Objekt umwandelt, indem es aus der Nominalphrase die Vokativ-Form auswählt.

Nun kann aus einem, möglicherweise leeren, `PConf`-Objekt, einer Utterance und einem, ebenfalls optionalen, Vokativ eine Phrase gebildet werden, in dem die drei Zeichenketten in den jeweiligen `s`-Feldern in eben dieser Reihenfolge, in der sie aufgezählt wurden, verkettet werden.[93]

Mit diesen Syntaxregeln ist das Ziel erreicht, Sätze mit der Startkategorie `Phr` zu erzeugen und zu parsen. Somit lässt sich diese Lateingrammatik, in gewissem Umfang, zusammen mit anderen Sprachen der Ressource Grammar Library verwenden.

[92] phrase-beginning conjunction
[93] vgl. **PhraseLat.gf**

4 Fazit

4.1 Ergebnis

Alle nötigen Schritte, um eine Ressource-Grammatik im Grammatical Framework zu implementieren, sind hiermit abgeschlossen. Die Grammatik kann nun geladen und getestet werden. Um die Lateingrammatik zu laden, muss entweder die Datei **LangLat.gf** oder die Datei **AllLat.gf** geladen werden. Dies geschieht entweder durch das Aufrufen des Grammatical Frameworks mit der gewünschten Datei als Kommandozeilenparameter (`$ gf LangLat.gf` bzw. `$ gf AllLat.gf`) oder, wenn die interaktive Eingabeaufforderung des Grammatical Frameworks bereits gestartet ist, durch den Befehl `> import LangLat.gf` bzw. `> import AllLat.gf`. Die erste dieser Dateien lädt nur die Bestandteile, die alle Sprachen in der Ressource Grammar Library gemein haben, die zweite lädt zusätzlich die sprachspezifische Extra-Grammatik. Ist eine dieser Dateien geladen, kann man die Abhängigkeiten der einzelnen Module betrachten, um den vollen Umfang dieser Implementierung zu sehen. Wie dieser Abhängigkeitsgraph für die Datei **AllLat.gf** aussieht, ist in Abbildung 4.1 zu sehen. Der Graph für **LangLat.gf** ist ein Subgraph dieses Graphen, bei dem alle Knoten fehlen, die mit "All" und "Extra" beginnen.

Nun kann man zum Testen entweder Testsätze parsen oder Sätze anhand der Grammatik erzeugen, um sie auf Korrektheit zu überprüfen. Zum Parsen existiert im Grammatical Framework der Befehl `parse`, dem eine Zeichenkette als Parameter übergeben wird und als der Ergebnis eine Zeichenkette, welche die entsprechenden abstrakten Syntaxbäume repräsentiert, zurück gibt. So kann z.B. der Satz *feminae dormiunt* (deutsch: "Die Frauen schlafen" bzw. "Frauen schlafen") geparsed werden. Als Ergebnis erhält man dann die abstrakten Syntaxbäume in Listing 4.1. Mit dem Befehl `generate_random` lassen sich dagegen zufällige abstrakte Bäume erzeugen, die mit dem Befehl `linearize` in einen lateinischen Satz linearisiert werden können. So erzeugt die Befehlsfolge `> generate_random | linearize` einen zufälligen lateinischen Satz. Mit dem Befehl `generate_trees` können alle Bäume einer Kategorie, bis zu einer gewissen Tiefe, meist der Tiefe von vier, generiert werden. So

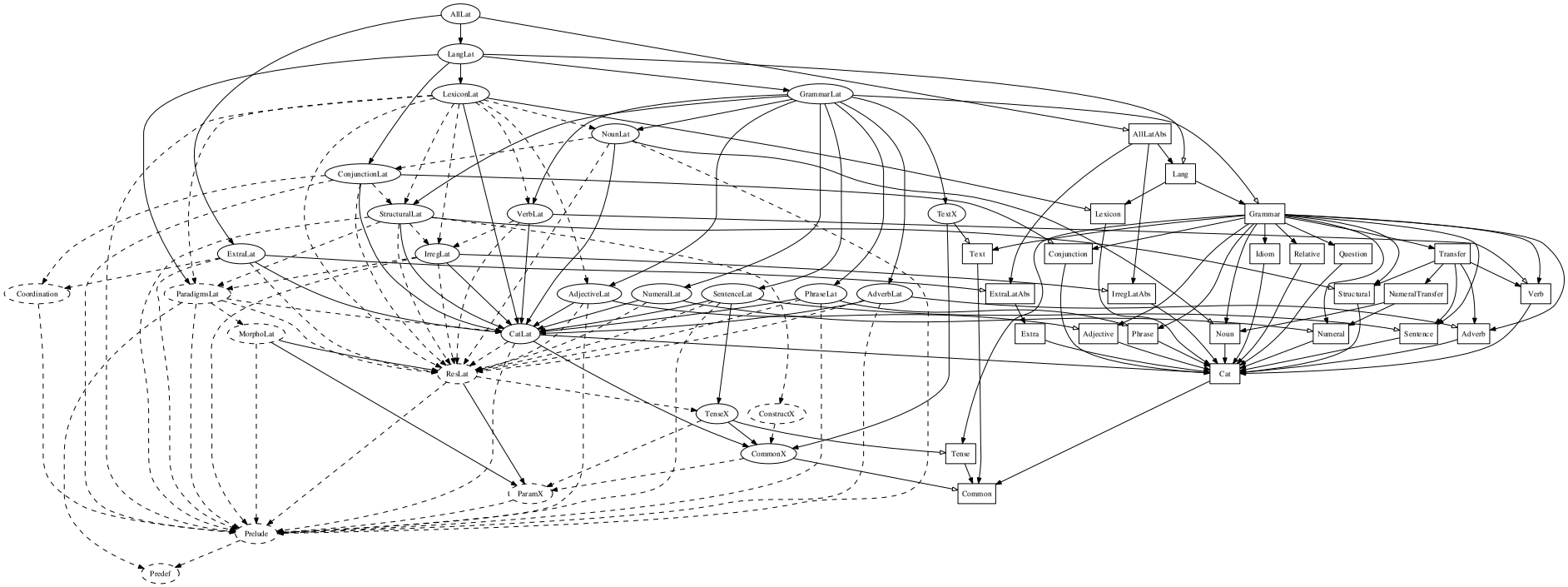

Abbildung 4.1: Die Abhängigkeiten zwischen den Modulen der Lateingrammatik

```
1 PhrUtt NoPConj (UttS (UseCl (TTAnt TPres ASimul) PPos \
2  (PredVP (DetCN (DetQuant DefArt NumPl) (UseN woman_N)) \
3  (UseV sleep_V)))) NoVoc
4 PhrUtt NoPConj (UttS (UseCl (TTAnt TPres ASimul) PPos \
5  (PredVP (DetCN (DetQuant IndefArt NumPl) (UseN woman_N)) \
6  (UseV sleep_V)))) NoVoc
```

Listing 4.1: Abstrakte Syntaxrepräsentationen des Satzes *feminae dormiunt*

lassen sich zumindest theoretisch alle in der Grammatik möglichen Sätze bilden. Allerdings sind dies bei dieser Lateingrammatik, selbst bei einer Tiefe von vier, schon fast 70 Millionen Sätze.

Offensichtlich ist diese Lateingrammatik also bereits zu einem gewissen Grade benutzbar, jedoch gibt es noch großen Spielraum für Erweiterungen. Die Bereiche des Lexikons und der Morphologie sind bereits komplett, entsprechend der abstrakten Syntax, die für eine Ressource Grammar vorgegeben ist, implementiert. Hier wäre es wohl am ehesten sinnvoll, die Ergebnisse, die bereits erzielt werden können zu bewerten, nach möglichen Fehlern zu suchen und diese zu korrigieren. So wären mögliche Fehler im Lexikon z.B. Übersetzungen, die im tatsächlichen Sprachgebrauch fast nie vorkommen oder viel seltener vorkommen als gleichbedeutende Wörter. In diesem Falle sollte die Übersetzung angepasst werden. Fehler in der Morphologie dagegen können dazu führen, dass entweder existierende Wortformen nicht analysiert werden können oder dass Wortformen falsch analysiert werden. Man kann nun unter anderem versuchen, unter Verwendung von Korpora, diese Fehler zu finden. Dies kann jedoch mit langwieriger Arbeit verbunden sein, die unter Umständen nur wenig Verbesserung bringt.

Dagegen ist im Bereich der Syntax noch einige Arbeit nötig, um den vollen Umfang zu erreichen. Viele interessante linguistische Phänomene warten noch darauf, umgesetzt zu werden, denn das Ziel dieser Arbeit war es, überhaupt in der Lage zu sein, lateinische Sätze bilden und analysieren zu können. So fehlen aktuell leider noch Relativsätze, Fragesätze, verschiedene Wortstellungen bei der `Phr`-Kategorie, Partizipialkonstruktionen, etc. um nur ein paar zu nennen. Dadurch, dass die Anzahl und Funktion der Regeln vorgegeben ist, sollte es aber im Anschluss an diese Arbeit relativ einfach sein, die Grammatik so zu vervollständigen, dass sie als vollständige Sprache in das Grammatical Framework aufgenommen werden kann.

4.2 Anwendung

Eine Frage, der man sich für die Zukunft ebenfalls widmen kann, ist die mögliche Anwendung dieser Lateingrammatik, vor allem, wenn sie in Zukunft noch in den gerade genannten Bereichen verbessert werden sollte.

Der Traum jedes Schülers und zugleich wohl der Alptraum der Lehrer wäre ein tatsächlich funktionierendes Lateinübersetzungsprogramm. Ob dieses Ziel, vor allem bei der Übersetzung literarischer und lyrischer Texte, erreicht werden kann, ist jedoch noch nicht abzusehen. Allerdings wäre es denkbar z.B. eine automatische Übersetzungshilfe für lateinische Inschriften für interessierte Touristen anzubieten. Dafür wäre die Integration in eine Anwendung für moderne Mobiltelefone denkbar. Dies ist insofern möglich, da das Grammatical Framework in der Lage ist, Grammatiken in ein portables Grammatikformat namens „PGF“ umzuwandeln, für das es, teils noch unvollständige, Bibliotheken für verschiedene Programmiersprachen gibt. So auch für Java, wodurch es möglich ist, eine PGF-Grammatik in einer Android-Anwendung zu verwenden. Ebenfalls wäre die Integration in ein Webangebot möglich, da auch hierfür Bibliotheken bereitgestellt werden.[1]

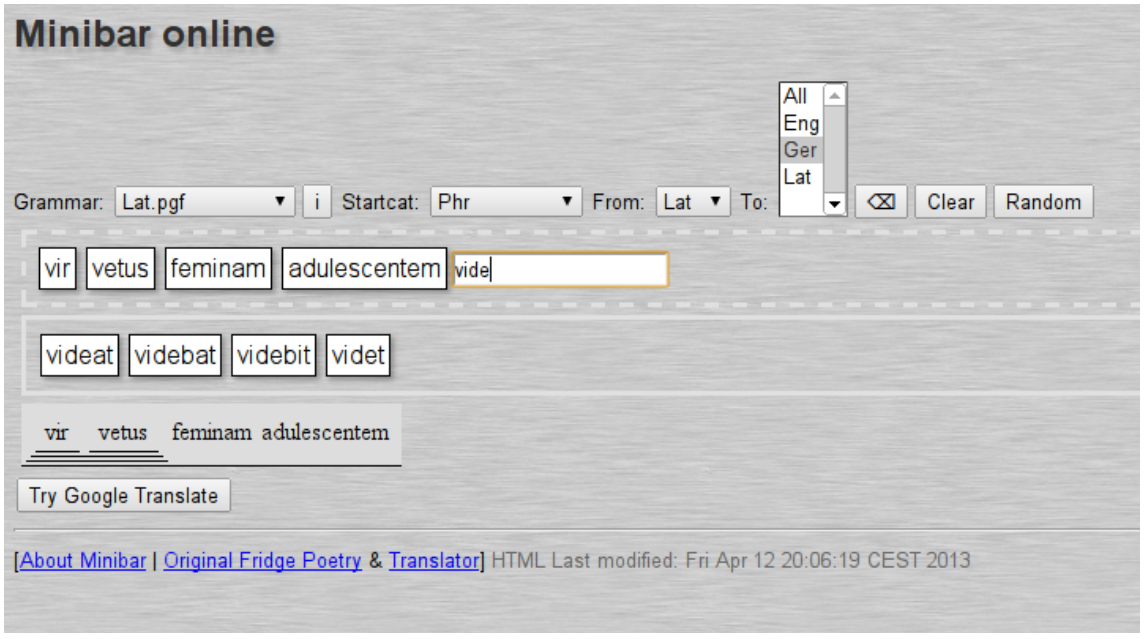

Abbildung 4.2: Beispiel für eine „fridge magnet“-Sitzung

Interessanter dürfte jedoch die Überlegung sein, inwieweit diese Lateingrammatik im Unterricht eingesetzt werden könnte. Denn die Verwendung des Computers in der Schule wird wohl auf Dauer zunehmen. Deshalb müssen für verschiedenste Schul-

[1] vgl. [Ran11] S. 166ff.

fächer Konzepte zur Verwendung neuer Medien erarbeitet werden, so auch für den Lateinunterricht, und eben in diesem Bereich sind meiner Meinung nach mögliche Anwendungen dieser Arbeit zu finden. Zum einen ist eine beliebte Beispielanwendung für die Mehrsprachigkeit des Grammatical Frameworks eine Webanwendung namens „fridge magnets poetry“, bei der man wie aus Magneten am Kühlschrank, auf die Wörter gedruckt sind, Sätze bilden kann. Anders als am Kühlschrank müssen die Sätze den Regeln einer Grammatik folgen. Die so gebildeten Sätze können dann in verschiedene Sprachen übersetzt werden.[2] Auf diese Weise ist ein spielerischer Umgang mit der lateinischen Sprache möglich, indem man z.B. deutsche Sätze baut und sich anschließend ansehen kann, wie eben dieser Satz auf Latein aussehen würde. Durch die Beschränkung, dass nur syntaktisch korrekte Sätze konstruiert werden können, kann man durch das Zusammensetzen lateinischer Sätze möglicherweise spielerisch ein Sprachgefühl entwickeln.

Des weiteren bietet das Grammatical Framework auch Möglichkeiten mit Hilfe einer Grammatik Übungen für eine Sprache zu bieten. Denn das Grammatical Framework bietet so genannte “Quizes” . Das erste dieser Fragespiele, das mit dem Befehl `morphology_quiz` gestartet wird, fragt nach einer spezifischen Wortform eines Wortes einer gegebenen Kategorie, in dem es einem die Grundform und die gewünschten morphologischen Merkmale nennt. Eine Beispielsitzung ist in Listing 4.2 zu sehen. Auf diese Weise lässt sich die lateinische Wortformenbildung üben. Um das Übersetzten auch abseits von Lehrbuchtexten zu üben, stellt das Grammatical Framework analog zum “Morphology Quiz” ein “Translation Quiz” bereit. Zur Verwendung müssen mindestens die Grammatiken zweier Sprachen geladen sein. Anschließend kann man unter Angabe der Quell- und Zielsprache, ähnlich wie bei dem “Morphology Quiz”, vom Programm vorgegebene Sätze übersetzen, wobei das Können wieder wie oben durch eine Punktzahl bewertet wird.

Diese zwei eingebauten Übungsmöglichkeiten sind jedoch nur bedingt für den wirklichen Einsatz im Unterricht geeignet. Denn viele, nicht zu sehr Technik-affine, Schüler und Schülerinnen werden möglicherweise von Konsolensitzungen abgeschreckt. Da das Grammatical Framework aber leicht in eigene Programme und Systeme integriert werden kann, bieten sich auch hier Möglichkeiten zur Verbesserung und Anpassung an die pädagogischen Bedürfnisse des Lateinunterrichts.

In dieser Arbeit wurden die am Anfang beschriebenen Ziele erreicht. Doch wie in diesem Abschnitt dargestellt wurde, bieten sich noch Möglichkeiten, um an diese Arbeit anzuknüpfen. Dabei kann man sich entscheiden, ob man sich bemüht die

[2] vgl. [Ran11] S. 166

```
> morphology_quiz -cat=N
Welcome to GF Morphology Quiz.
The quiz is over when you have done at least 10 examples
with at least 75 % success.

You can interrupt the quiz by entering a line consisting of a dot ('.').


navis s Pl Voc
> naves
Yes.
Score 1/1
plastica s Pl Nom
> plasticae
Yes.
Score 2/2
pulvis s Sg Voc
>
```

Listing 4.2: Eine Beispielsitzung des Grammatical Framework Morphology Quiz

Grammatik weiter zu verbessern oder sich lieber mit der konkreten Umsetzung einer oder mehrerer Verwendungsmöglichkeiten widmet. Für Interessierte bietet diese Arbeit aber auf jeden Fall Anregung und Grundlage für weiterer Beschäftigung in den verschiedensten Bereichen.

Literatur

[BC06] David Bamman und Gregory Crane. "The Design and Use of a Latin Dependency Treebank". In: *Proceedings of the Fifth International Treebanks and Linguistic Theories Conference*. Hrsg. von Jan Hajic und Joakim Nivre. Institute of Formal, Applied Linguistics, Faculty of Mathematics und Physics, Charles University. Prag, 2006, S. 67–78. URL: `http://hdl.handle.net/10427/42684`.

[BL94] Karl Bayer und Josef Lindauer, Hrsg. *Lateinische Grammatik*. 2. Auflage, auf der Grundlage der Lateinischen Schulgrammatik von Landgraf-Leitschuh neu bearbeitet. C.C. Buchners Verlag, J. Lindauer Verlag, R. Oldenburg Verlag, 1994.

[DFV12] Anette Dralle, Walther Federking und Gregor Vetter, Hrsg. *Schülerwörterbuch Latein. Latein-Deutsch und Deutsch-Latein*. 1. Auflage. PONS GmbH, 2012.

[Glu04] Helmut Glück, Hrsg. *Metzler Lexikon Sprache*. 2. Auflage. Digitale Bibliothek 34. Directmedia, 2004.

[Mul06] Johannes Müller-Lancé. *Latein für Romanisten. Ein Lehr- und Arbeitsbuch*. 1. Auflage. Gunter Narr Verlag, 2006.

[PL81] Dr. Erich Pertsch und Dr. Ernst Erwin Lange-Kowal, Hrsg. *Langenscheidts Schulwörterbuch Lateinisch. Lateinisch-Deutsch Deutsch-Latein*. 14. Auflage. Langenscheidt, 1981.

[Ran11] Aarne Ranta. *Grammatical Framework. Programming with Multilingual Grammars*. CSLL Studies in Computational Linguistics, 2011.

[Sch08] Uwe Schöning. *Theoretische Intormatik - kurz gefasst*. 5. Auflage. Spektrum Akademischer Verlag, 2008.

Tabellen- und Abbildungsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Hausarbeit selbständig und ohne fremde Hilfe angefertigt, alle benutzten Quellen und Hilfsmittel angegeben und Zitate als solche kenntlich gemacht habe.
Ich versichere ferner, dass ich die Arbeit weder für eine Prüfung an einer weiteren Hochschule noch für eine staatliche Prüfung eingereicht habe.

München, den _____

_____

Herbert Lange