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

Erstellen einer Lateingrammatik im Grammatical Framework

vorgelegt von Herbert Lange

Fach: Computerlinguistik
Referent: Prof. Dr. Klaus U. Schulz
München, den 30.9.2013

Zusammenfassung

In dieser Arbeit sollen an einem konkreten Beispiel die nötigen Schritte gezeigt werden, um eine computergestützte Grammatik für eine natürliche Sprache zu entwerfen. Am Beispiel der lateinischen Sprache wird gezeig, wie ein Lexikon, ein Morphologiesystem und eine Syntax implementiert werden kann, die sich in ein größeres, multilinguales Grammatiksystem einfügen lässt. Dadurch können zum einen in der implementierten Sprache Sätze einfach geparsed werden, aber auch in jede andere im System vorhandene Sprache übersetzt werden können. Gezeigt wird ein rein regelbasierter Ansatz der sich von den statistischen Methoden durch seine Striktheit und Beschränktheit, aber auch durch seine Zuverlässigkeit abhebt.

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 gute 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 die Fülle an Daten, die über das Internet verfügbar sind. Die Grundlagen regelbasierter Grammatiken sind in etwa so alt wie die Wissenschaft der Linguistik selbst und sollte auch weiterhin zum Wissen eines jeden Computerlinguisten gehören.

In dieser Arbeit sollen an einem konkreten Beispiel die nötigen Schritte gezeigt werden, um eine computergestützte Grammatik für eine natürliche Sprache zu entwerfen. Am Beispiel der lateinischen Sprache wird gezeig, wie ein Lexikon, ein Morphologiesystem und eine Syntax implementiert werden kann, die sich in ein größeres, multilinguales Grammatiksystem einfügen lässt. Vor allem 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 war es, einen kleinen Beitrag zu einem größeren Projekt zu leisten. Denn die lateinische Sprache war im Grammatical Framework[1], dem für diese Arbeit gewählten multilingualem Grammatik-, Parsing- und Übersetzungssystem, noch nicht in einem funktionsfähigen Umfang vorhanden.

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

1.2. Ziel der Arbeit

Ziel diser Arbeit ist es, zum einen eine 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 zum einen auf der Nähe zu einer gewöhnlichen, im bayerischen Schulunterricht verwendeten, lateinischen Schulgrammatik liegen. Dies spiegelt sich, zum einen teilweise in der Abfolge der Schritte, zum anderen in einigen Entscheidungen beim Entwurf der Grammatik, wieder. So ist eine lateinische Grammatik grob in folgende Abschnitte unterteilt: Phonologie, Wortarten und Wortbildung, Morphologie uns Syntax. Zwar werden die ersten drei Teile in dieser Arbeit größtenteils vernachläßigt, die logische Folge der zwei verbleibenden Teile wird aber auch hier beibehalten. Allerding wird der in Grammatiken nicht in dieser Form auffindbare Teil über die Lexikonentwicklung ihnen vorangestellt.

Zum anderen sollen, um die Verwendung im multilingualen Grammatiksystem des Grammatical Framework ermöglichen zu können, möglichst große Teile der für eine Sprache geforderten Schnittstellen zur Verfügung gestellt werden. Dies beeinflusst ebenfalls die Struktur der Arbeit, durch die Gliederung einer Grammatik in gewisse Module.

1.3. Aufbau der Arbeit

Zu Beginn der Arbeit werden in Kapitel 2 die nötigen Grundlagen für die weiteren Teile der Arbeit erörtert. Zunächsteinmal werden die Grundlagen des Grammatical Framework in Abschnitt 2.1 erklärt. Zuerst eine grundlegende Beschreibung des Umfangs und der Funktionen dieses Programmpakets. Anschließend folgt eine Einführung in den Formalismus der bei der Entwicklung der Grammatiken für das Grammatical Framework verwendet wird. Und dieser Abschnitt wird mit einigen Informationen zur Ressource Grammar Library, der multilingualen Grammatikbibliothek des Grammatical Frameworks abgeschlossen. Nach den technischeren Grundlagen folgen einige Informationen zur lateinischen Sprache in Abschnitt 2.2. Zunächst erfolgt eine sprachwissenschaftliche Einordnung dieser Sprache unter besonderer Hervorhebung einiger interessanter Merkmale. Darauf folgt ein kurzer Abschnitt, der die Relevanz dieser als tot geltenden Sprache in der heutigen Zeit beleuchtet.

Nach dieser allgemeinen Einführung erfolgt im nächsten großen Teil, dem Kapitel 3, die Beschreibung der nötigen Schritte, die umgesetzt wurden um eine Lateingrammatik im Grammatical Framework zu entwickelt. Es ist in die drei Abschnitte Lexikon, Morphologie und Syntax unterteilt, da dies drei getrennte Module in der Ressource Grammar Library bilden. Bei der Entwicklung er Grammatik zeigten sich allerdings oft Anhängigkeiten 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 Beschreibund der lateinischen Wortflexion, 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 waren um eine funktionierende Grammatik zu erhalten, was ja zu den Hauptzielen dieser Arbeit gehörte.

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

2. Grundlagen

2.1. Das Grammatical Framework

Das Grammatical Framework ist ein Softwaresystem mit einer spezialisierten Programmiersprache zur Entwicklung von Grammatiken. Es bietet alle nötigen Möglichkeiten um natürliche Sprachen zu verarbeiten. Dabei benutzt es Formalismen, wie sie auch in modernen funktionalen Programmiersprachen wie Haskell zu finden sind.[1] Somit können einem manche Konzepte bereits vertraut sein, wenn man sich bereits mit den Möglichkeiten der funktionalen Programmierung auseinandergesetzt 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.[2]

Die große Stärke dabei ist die Multilingualität. Grundkonzept dabei ist die Trennung in eine konkrete und eine abstrakte Repräsentation der Grammatik. Dabei ist die abstrakte Struktur verschiedenen Sprachen gemein und die konkrete Syntax beschreibt, wie aus einem sprachunabhängigen Baum 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.[3] Dies Details dieses Formalismuses sollen nun genauer betrachtet werden.

2.1.1. Der Grammatikformalismus

Meist werden im Bereich der Computerlinguistik und Informatik kontextfreie Grammatiken, also Grammatiken von Typ 2 der Chomsky-Hierarchie verwendet.[4] Dies hat meist den Grund, dass die Mächtigkeit dieses Formalismuses meist ausreicht, die gewünschten Sprachen zu beschreiben. So sind in kontextfreien Sprachen geklammerte

[1] vgl. [Ran11] S. vii
[2] vgl. [Ran11] S. 127ff.
[3] vgl. [Ran11] S. 10ff.
[4] vgl. [Sch08] S. 9f

Ausdrücke möglich, die bei Programmiersprachen recht häufig sind.[5] Allerdings ist der Verarbeitungsaufwand vergleichsweise gering zu Grammatiken einer der höheren Stufen und es existieren sehr performante Algorithmen zum Parsen mit kontextfreien Grammatiken, so z.B. der Cocke-Younger-Kasami-Algorithmus, auch bekannt als CYK-Algorithmus.[6] Die in Beispiel 1 gegebene Grammatik ist ein sehr minimalis-

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: Kontextfreie Grammatikfragment

tisches Beispiel für eine kontextfreie Grammatik. Mit ihrer Hilfe kann nur der eine deutsche Satz *Der Mann schläft* hergeleitet werden. Dabei hat eine mögliche Ableitung die in Beispiel 2 gezeigte Form. Dabei wird 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 jedoch dem Ablesen der gegebenen Ableiung 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.

```
S
⇒[1] NP VP
⇒[2] Det N VP
⇒[4] der N VP
⇒[3] der Mann VP
⇒[5] der Mann V
⇒[6] der Mann schläft
```

Beispiel 2: Ableitung des Satzes

Im Formalismus des Grammatical Framework wird die oben gegebene Grammatik in die abstrakte und die konkrete Syntax zerlegt. Dabei entspricht die abstrakte Syntax in etwa dem Syntaxbaum ohne die terminalen Blätter. Die abstrakte Syntax der kontextfreien Grammatik aus Beispiel 1 ist in Listing 2.1 zu sehen. Zunächst

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

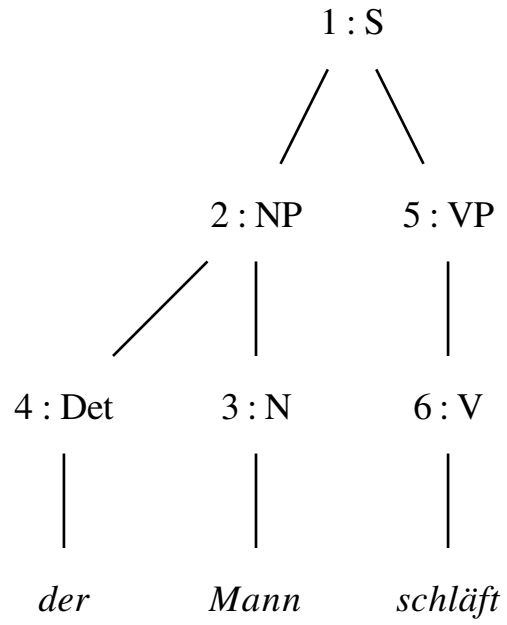

Abbildung 2.1.: Entsprechender Syntaxbaum

gibt das Schlüsselwort `abstract` an, dass es sich um Datei mit einer abstrakten Syntaxbeschreibung handelt. Dieses Schlüsselwort wird vom Namen der Grammatik gefolgt. Anschließend folgt der Inhalt der Grammatik.

Zunächst werden mithilfe 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.[7]

Nach dem Schlüsselwort `cat` folgt eine Liste der nicht-lexikalischen Kategorien oder auch Nonterminal-Symbole. Sie entsprechen in etwa den Datentypen in (funktionalen) Programmiersprachen.

Hauptbestandteil der Grammatik sind offensichtlich die Syntaxregeln. Sie werden nach dem Schlüsselwort `fun` aufgelistet. Die Regeln ähneln der Form von Funktionensignaturen in Sprachen wie Standard ML oder Haskell, denn diese Regeln beschreiben lediglich die Bestandteile aus denen ein Ausdruck einer neuen Kategorie zusammengesetzt werden soll, ohne eine Aussage über das genaue Vorgehen zu treffen. Dies wird unabhänging voneinander in jeder konkreten Grammatik, die diese abstrakte Grammatik implementiert, beschrieben. So sagt die erste Regel mit dem

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

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 tatsächliche Reihenfolge der Bestandteile in konkreten Zeichenketten getroffen. Die letzten drei Regeln führen lediglich die lexikalische Einheiten mit einer entsprechenden Kategorie ein.

```
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

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

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

```
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 deutschen Umlaute nötig das Encoding anzugeben. Für andere Sprachen mit komplett vom lateinischen unterschiedlichen Schriftsystemen, gibt es auch die Möglichkeit statt der direkten Zeichenkodierung eine Transliteration zu verwenden. Dabei wird eine bijektive Abbildung zwischen Unicodezeichen und Zeichenketten der Länge eins oder größer, die nur aus ASCII-Zeichen bestehen.[8]

Das Schlüsselwort `lincat` ist die konkrete Entsprechung zum Schlüsselwort `cat` in der abstrakten Syntax. Hier müssen 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[9], gewählt. Das Grammatical Framework unterstützt auch verschiedene Arten komplexer Datentypen.

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. 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 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 abstrakte

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

[9] Um genau zu sein, eine Liste von Tokens, die am Schluss mit Leerzeichen konkateniert werden

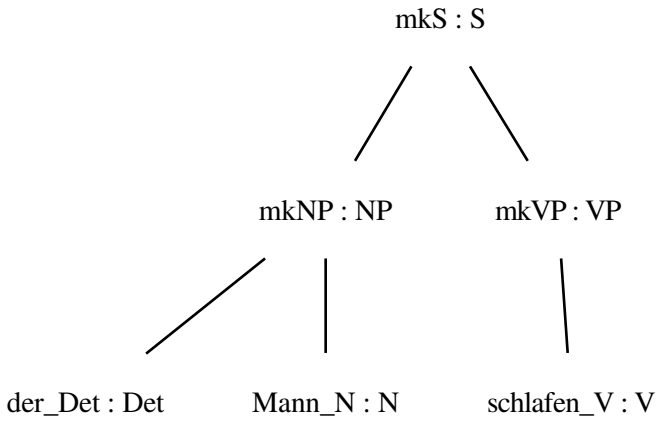

Abbildung 2.2.: Baum der abstrakten Syntax

Representation `(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 auch verschiedene Bäume beim Parsen grafisch darstellen lassen. Zum einen den abstrakten Syntaxbaum und zum anderen den konkreten Parsebaum für eine der implementierten Sprachen. Diese Bäume für die MiniSatz-Grammatik sind in Abb. 2.2 und 2.3 zu sehen. Nun kann

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

Listing 2.4: Erweiterte abstrakte Syntax

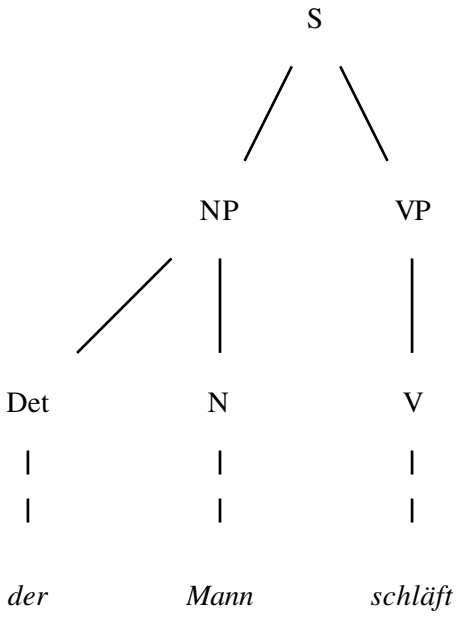

Abbildung 2.3.: Parsebaum der konkreten deutschen Syntax

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 sind nicht sehr umfangreich und 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. Sonst wird noch das “Lexikon”, also die Liste der lexikalischen Einheiten, um die Wörter für Frau, Buch, sehen und lesen erweitert.

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 gesammter 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

```
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

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 möglicherweise verschiedenen Typs in einem einzigen Objekt zusammen. Jedes dieser inneren Objekte hat einen Typ und einen Bezeichner, über den darauf zugegriffen werden kann.[10] So haben z.B. Nomen ein inherentes 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

[10] vgl. [Ran11] S. 62

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 sogenannten Tabellentypen. Ihre Signatur hat die Form Typ_1 `=>` Typ_2, 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.[11] Um es nocheinmal 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 nun müssen sowohl die Verbundtypen als auch die Tabellentypen konkret erzeugt werden. Dies geschieht nun im `lin`-Block, in dem nun die konkreten Grammatikregeln "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. Und diese Tabelle hat den Typ `Numerus => Str`, also eine `Str`-Zeichenkette, deren Wert von einem Numerus abhängt. 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 Bezeicher `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 darauffolgenden 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. Denn da die Verben hier nur in der 3. Person vorkommen hängt die Form lediglich vom Numerus ab. Also hat das Verb `lesen_V` die folgenden zwei Formen, `liest` im Singular und *lesen* im Plural. Nach dem selben Schema funktionieren auch die restlichen Verben und die Nomen. Allerdings haben die Nomen ihr inherentes Genus. Deshalb haben sie in ihrem Verbund zusätzlich den Bezeichner `g`, dem das natürliche Geschlecht des Nomens in der lexikalischen Regel 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 bestimmten Artikel im Singular etwas komplizierter. Sie haben ein inherentes Numerusmerkmal, das den Numerus des Nomens in der Nominanphra-

[11] vgl. [Ran11] S. 59

se 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 `s` zuweisen will, so kann man `table { _ => s }` schreiben. Dies lässt sich im Grammatical Framework weiter verkürzen zu `\\_ => s`. Hat man zwei solche Tabellen ineinandergeschachtelt, also `\\_ => \\_ => s`. So kann man das weiter verkürzen zu `\\_,_ => s`. 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 des Typs `Det` und einem Objekt 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 eine Variable ersetzt wird, so dass der Wert, mit dem ein Eintrag aus der Tabelle ausgewählt werden soll, nicht verloren geht, sondern in dieser Variable verfügbar bleibt. Auf diese Art kann der möglicherweise ausgewählte Kasus in der Regel verwendet werden um mit ihm aus einer anderen Tabelle werte zu wählen. Der Operator um aus einem Objekt eines Tabellentyps einen Wert zu wählen, ist das `!`. Um auf die

verschiedenen Felder in Verbundtypen zuzugreifen wird dagegen der .-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 einen Wert vom Typ `Str`. Um aus einem `Det`-Objekt einen `Str`-Wert zu erhalten, müssen wir zweimal 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 Stringwerte können nun wieder konkateniert werden. Dieser zusammengesetzte Wert wird schließlich in die Tabelle eingefügt. Diese kompakte Tabelle ist die Kurzform für die ausführlichere Tabelle in Listing 2.6. Verbalphrasen haben nahezu den gleichen Typ wie die Verben `V` und `V2`.

```
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

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. Denn sie übernimmt den Wert des Verbs und erweitert lediglich den Verbund um das `o`-Feld mit einer leeren Zeichenkette als Wert, denn bei intransitiven Verben ist kein Akkusativobjekt vorhanden. In der Regel `mkVP2` für intransitive Verben mit Akkusativobjekt dagegen 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 konkateniert, 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 geünschten 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

```
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

`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 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 Objektzeichenkette. 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 die selbe abstrakte Syntax implementieren, so gibt es doch große Unterschiede in den zu berücksichtigenden Merkmalen.

Der gesammte 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[12] als auch eine etwas ausführlichere Sprachbeschreibung[13]. Diese Ressource sind zwar hilfreich, allerdings ändert sich die Sprache in einem gewissen Rahmen schneller weiter, als die Dokumentation aktualisiert werden kann.

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. Sie ist definiert als gemeinsame abstrakte Syntax, die für verschiedenen Sprachen implementiert ist. Auf diese Möglichkeit ist zum eine grundlegende Übersetzung zwischen den unterstützten Sprachen direkt nach der Installation möglich. Allerdings nur im durch das von Haus aus gegebene Grundvokabular. Meist muss also mindestens das nötige Vokabular hinzugefügt werden, um die Möglichkeite dieser Bibliothek in eigenen Anwendungen zu verwenden.

Zunächsteinmal bietet die Ressource Grammar Library die Definition verschiedener, sogenannter geschlossener Kategorien, also Kategorien, deren Werte man zumindest theoretisch komplett aufzählen kann. Dazu gehören Vergleichsadverbien, Konjunktionen, Pronomen, Prepositionen, Subjunktionen. Weiterhin gibt es offene Kategorien wie verschiedene Adverbtypen 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. Insgesammt gibt es ca. 42 geschlossenen 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[14].

Mit Hilfe der in der RGL vorhandenen Konstruktoren kann unser bereits gezeigtes Grammatikfragment nocheinmal 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. So z.B. über die interne Struktur der Kategorien oder die Übereinstimmung zwischen den Merkmalen. Auch die Details der Wortbildung werden größtenteils

[12] `http://www.grammaticalframework.org/doc/gf-reference.html`
[13] `http://www.grammaticalframework.org/doc/gf-refman.html`
[14] `http://www.grammaticalframework.org/lib/doc/synopsis.html`

ausgeblendet, auch wenn, in den folgenden Beispielen, im Deutschen relativ viele Verbformen nötig sind, damit alle Verbformen richtig gebildet werden. 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.

```
1 --# -path=.:alltenses:prelude
2 abstract RglSatzAbs = open Syntax in {
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

Bei der abstrakten Syntax in Listing 2.8 gibt es nur eine kleine Änderung. In der ersten Zeile findet man nun das neue Schlüsselwort `open`. Dieses `open`, ein Modulname des gleichen Typs, wie die einbindende Datei, und ein `it` 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. Der Unterschied zwischen den beiden ist lediglich die Sichtbarkeit der Bestandteile des geladenen Moduls. So sind, wenn ein Modul mit `**` um Code erweitert wird, alle Teile des Moduls im erweiterten Code sichtbar. Wird das Modul hingegen mit *open* geladen, so ist der Zugriff auf die darin enthaltenen Komponenten nur explizit über den Modulnamen möglich.[15] In diesem Falle wird also das Modul `Syntax` aus der Ressource Grammar Library so geladen, dass der Zugriff über den Modulnamen möglich ist.

Die darauf aufbauenden konkreten Grammatiken in Listing 2.9 und 2.10 sind

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

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

Listing 2.9: Konkrete deutsche Syntax mit Hilfe der RGL

diesmal einander sehr ähnlich. Bei der deutschen Grammatik wird analog zum abstrakten Modul `Syntax` das konkrete Modul `SyntaxGer` per `open` 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 übernehmen. Bei den lexikalischen Regeln werden zum einen Regeln aus `ParadigmsLat` 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 so wie dem Genus ein Nomen-Objekt erstellt. Die Funktion `mkV` erstellt aus fünf Verbformen ein Verbobjekt, das das ganze Paradigma beinhaltet. Und die Funktion `mkV2` erzeugt dann aus einem intransitiven Verbobjekt 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 Vorzeitigkeit, also ob das Zeitverhältnis vorzeitig oder gleichzeitig ist, und eine Polarität, also ob der Satz verneint ist oder nicht. Da wir nur positive Präsenssätze behandeln, können wir diese Parameter so festsetzen, dass genau diese Sätze gebildet werden.

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

Listing 2.10: Konkrete englische Syntax mit Hilfe der RGL

In der englischen Grammatik sind die Unterschiede diesmal sehr gering und beziehen sich lediglich auf die Konstruktion der lexikalischen Objekte. Die Nomen können aus teilweise aus einer einzigen Form gebildet werden, wie bei `Buch_N`. Ebenso wie die Verben hier, denn die Anzahl der nötigen Formen ist abhängig von der Regelmäßigkeit der Formenbildung. Und die ist in der englischen Sprache etwas größer als im Deutschen. Der Rest der Grammatik ist identisch, abgesehen davon, dass natürlich die englische Version des konkreten `Syntax`- und `Paradigms`-Moduls benutzt wird. Es ist offensichtlich Grammatiken selbst für unterschiedliche Sprachen sehr ähnlich werden, wenn man zu ihrer Implementierung die Ressource Grammar Library benutzt. Dies ermöglicht unter anderem Grammatiken für Sprachen zu entwickeln, die man nicht komplett beherrscht, und damit auch natürlichsprachliche Anwendungen

für diese Sprachen zu entwickeln.

2.2. Die Lateinische Sprache

2.2.1. Sprachwissenschaftliche Einordnung

Die lateinische Sprache, auch als oskisch-umbrische Sprache bezeichnet, gehört zur indogermanische Sprachfamilie und dort zur Unterfamilie der italischen Sprachen. Durch diese Verwandschaft 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).[16] 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. BAYER-LINDAUER 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.[17] 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 Zeitaum 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 nachklassische Phase kann wiederum in verschiedene Epochen unterteilt werden, in denen unter anderem die romanischen Volkssprachen entstanden sind, bis hin zum sogenannten Neulatein, das noch vom 15. Jahrhundert bis hin zum beginn des 20. Jahrhundert die Sprache der Wissenschaft darstellte und auch heute noch großen Einfluss auf Begriffe des Alltags ausübt.[18]

Auch heute noch am bedeutendsten ist jedoch 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

[16] vgl. [BL94] S.1
[17] vgl. [Glu04] Lateinisch: S. 5359
[18] vgl. [Mul06] S. 27ff.

Sprachepoche stützen, wird diese primär in dieser Arbeit betrachtet.[19].

In der Sprachwissenschaft ist jedoch auch weiterhin umstritten, in welchem Verhältnis das klassische Latein zum sogenanten Vulgärlatein steht. Heutzutage geht man davon aus, dass das klasische Latein eine kaum wirklich gesprochene Sprache war und das Vulgärlatein nicht nur eine nachklassische Sprachvariante ist, sondern bereits parallel zum klasischen 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 hin zu den romanischen Sprachen entwickelte.[20]

Formal gehört Latein den stark flektierenden Sprachen. Das heißt das in der lateinischen Sprache, wie für synthetische Sprachen üblich, syntaktische Klassen und Verhältnisse über Wortsuffixe ausgedrückt werden.[21]. Allerdings drücken bei flektierenden Sprachen, im Gegensatz zu agglutinierenden Sprachen, die Affixe meist mehr als en grammatisches Merkmal aus.[22] So ist bei der Verbform *audio* das *audi* der Verbstamm, um genau zu sein den Präsensstamm, des Verbs *audire* und das Suffix *-o* kodiert folgende Merkmale: 1. Person, Singular, Präsens, Indikativ, Aktiv.[23]

Es gibt fünf zum Teil genusbasierte Flexionsklassen, also verschiedene Typen der Flexion innerhalb einer Wortart,[24] für Nomen, sechs verschiedene Kasus (Nominativ, Genitiv, Dativ, Akkusativ, Ablativ und Vokativ), drei Genera (Maskulin, Feminin, Neutrum), ein voll flektierendes Pronomensystem und vier relativ stark synthetische Flexionsklassen für Verben.[25] Zu den Kasus sei anzumerken, dass der Ablativ im Lateinischen ein eigenständiger Kasus ist, jedoch der Vokativ oft mit dem Nominativ zusammenfällt.[26]

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.[27]

[19] quelle
[20] vgl. [Glu04] Lateinisch: S. 5359 und Vulgärlatein: S. 10719
[21] vgl. [Glu04] Synthetisch: S. 9690
[22] vgl. [Glu04] Flektierende Sprache: S. 3009
[23] vgl. [BL94] S. 75
[24] vgl. [Glu04] Flexion: S. 3011
[25] [Glu04] Lateinisch: S. 5359
[26] vgl. [BL94] S. 20f.
[27] [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 nur 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 gilt Latein galt seit Jahrhunderten, und gilt weiterhin, als produktive Quelle für Fachbegriffe aus Wissenschaft, Forschung und Technik. So haben viele moderne Begriffe wie Computer[28] und Monitor[29] lateinische Wurzeln. Auch im Universitätsalltat wird man oft mit lateinischen Lehnwörtern konfrontiert. Man trifft sich zum Essen in der Mensa[30] und studiert an Fakultäten[31].

Vor allem in der Sprachwissenschaft hat Latein eine besondere Bedeutung, da sie bei einem Vergleich verschiedener indogermanischer Sprachen als eine Art *default*-Sprache angesehen werden kann, denn sie bietet fast alle nötigen 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.[32]

Als etwas skurile aber auch recht moderne Verwendung einer Variation der lateinischen Sprache kann *latino sine flexione* gelten. 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 Wortsschatz, der auch durch modernes romanisches Vokabular erweitert werden kann, und einer stark vereinfachten Morphologie.[33]

[28] von *lat.* computere - berechnen
[29] *lat.* f. der Mahner, von *lat.* monere - mahnen
[30] *lat.* mensa - Tisch, Tafel
[31] von *lat.* facultas - Vermögen, Fähigkeit
[32] vgl. [Mul06] S. 10
[33] vgl. [Glu04] Latino sine flexione S. 5374

3. Grammatikerstellung

Nach der Einführung in die nötigen Grundlagen, um die folgenden Schritte zu verstehen, die nötig sind um im Grammatical Framework eine Grammatik zu entwickeln, folgt nun eine Schilderung der konkreten Schritte die nötig waren um eine Lateingrammatik im Grammatical Framework zu entwickeln.

Es sei noch anzumerken, dass es bereits früher Bestrebungen von Aarne Ranta gab, eine Lateingrammatik für die Ressource Grammar Library des Grammatical Frameworks zu entwickeln. Diese Arbeit baut auf der Arbeit Rantas auf, kann aber insofern als selbständige und vollwertige Arbeit angsehen werden, da die bisherige implementierung sehr rudimentär und noch nicht funktionstüchtig war und seit ca. 2005 nicht mehr weiterentwickelt wurde. Im Anhang ist der Quelltext meiner Arbeit im Verhältnis zum Zustand vor Beginn der Arbeit zu finden.

Die Gliedering folgt dem gewählten Vorgehen bei der Implementierung. Die Begründung für die Reihenfolge der einzelnen Schritte wird jeweils zu Beginn der einzelnen Kapitel kurz dargelegt.

3.1. Lexikon

Den Beginn dieser Grammatikimplementierung bildete die Erstellung des minimal nötigen Lexikons. Durch die abstrakte Syntax der 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 Funktionen aus dem abstrakten Lexikon die zugeordneten Zeichenketten zugeordnet werden. Und zum anderen muss das Lexikon auch all jene Informationen enthalten, die später zum bilden der Vollformen und zur Konstruktion grammatischer Einheiten nötig sind. So z.B. bei Nomen das Geschlecht, wenn es nicht abgeleitet werden kann.

Der erste Schritt ist also, einfach das Lexikon einer anderen Sprache, in diesem Falle Englisch, zu kopieren. Normalerweise ist es vernünftiger, mit einer Sprache zu beginnen, die der zu implementierenden Sprache möglichst nahe steht.[2]. Allerdings wurde dieser Schritt bereits von Aaarne Ranta dadurch begonnen die englischen lexikalischen Ressourcen zu kopieren und anzupassen. Das ist auch insofern verständlich, da für die verwendeten Bezeichner in der Grammar Library bereits die englischen Begriffe zusammen mit einem Marker für die Wortart gewählt wurden. Anschließend werden zunächst alle Zeichenketten durch mögliche lateinische Entsprechungen ersetzt. Um eine mögliche Übersetzung für die verschiedenen Lexikoneinträge zu finden, müssten 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 den deutschen Entsprechungen, die für das weitere Vorgehen verwendet wurden, nicht genauer erläutert. Es sei nur so viel gesagt, dass die Bedeutung der meisten Bezeichner ohne weitere Hilfsmittel ersichtlich ist. Im Falle dessen, dass einmal Unklarheiten herrschten, wurde ein bekanntes Onlinewörterbuch[3] 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]),

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

soweit ein entsprechender Eintrag im diesem Wörterbuch zu finden war. 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 angeführe 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 konnte. 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 alte 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 lateinisch-deutschen Teil enthält, und somit mehr moderne Begriffe abdeckt, allerdings gibt es auch dort Begriffe, für die auch hier 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 interessantes Quelle für moderne Begriffe aus dem Breich 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

[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

gepflegt werden. Natürlich muss man immer bedenken, dass es keine 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 von der größe der an dem Projekt arbeitenden Comunity zusammenhängt. Neben der Wikipedia, die vom Konzept her eigentlich eine allgemeine Enzyklopädie ist, und nur im Nebeneffekt linguistische Resourcen 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 Lateinresourcen bei der Perseus Digital Libary[9].

Das erstere 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 direkten Übersetzung aus einer anderen Sprachee, es bietet aber für ein umfangreiches Vokabular sowohl eine morphologische Analyse für viele Wortformen als auch detailierte 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 Wörtrebücher sind.[10] Jedoch bietet Perseus die Möglichkeit einer erweiterten Suchfunktion so wie einer Angabe zur Wortfrequenz im verfügbaren Korpus.

Eine der Onlinequellen für moderne lateinische Begriffe wurde nicht verwendet, da sie nur zwischen Latein und Italienisch übersetzt. Dies würde aus verschiedenen Gründen zu Problemen führen. Diese Quelle soll allerdings trotzdem kurz erwähnt werden, denn sie ist die offizielle Liste des Vatikans zur Übersetzung moderner All-

[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`)

tagsbegrifft[11].

3.1.3. Geschlossene Kategorien

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

Adverbien gehören eigentlich nicht wirklich 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 strukturale Bestandteile aufgefasst werden können. Meist werden Adverbien aus Adjektiven gebildet, weswegen man sie zu den offenen Kategorien rechnen sollte. Jedoch ist dies nur eine von verschiedenen Möglichkeiten zur Verwendung von Adverbien. Vor allem 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] 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].

[11] `http://www.vatican.va/roman_curia/institutions_connected/latinitas/documents/rc_latinitas_20040601_lexicon_it.html`
[12] vgl. [BL94] S.12
[13] vgl. [BL94] S.44
[14] verb-phrase-modifying adverb vgl. [Ran11] S. 298
[15] interrogative adverb ebd.
[16] vgl. wer? - lat. *quis*, Dat. *cur*

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. 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.

Nichts direkt mit Pronomen zu tun haben die folgenden vergleichenden Adverbien. Um genau zu sein sind es Ausdrücke von 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*) dagegen drückt aus, dass erwas größer ist. Dagegen drückt `as_CAdv` (lat. *ita ... ut*) die Gleichheit aus. Objekte dieser Kategorie werden mit der Funktion `mkCAdv` aus den beiden Zeichenketten erstellt.

Eine weitere recht interessante Kategorie ist die Kategorie des Determinans. Diese werden meist auf Basis von Adjektiven gebildet. Dadurch ist es möglich mit einer einzigen Wortform auszukommen, um alle nötigen Formen zu bilden.

Die letzte hier zu erwähnende, einfache Kategorie von Wörtern sind Präpositionen. Präpositionen werden gemeinhin verwendet um die Funktion verschiedener Kasus genauer zu spezifizieren. Deshalb gibt es zu jeder Präposition zwingend auch eine Angabe, mit welchem Kasus sie verwendet werden kann.[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 Latein 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 in GF durch sogenannte freie Variationen ermöglicht (Listing 3.1). So kann *in* sowohl zusammen mit Akkusativ oder Ablativ gebraucht werden. Es gibt in diesem Teil des Lexikons noch einige weitere einfache Kategorien wie

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

Listing 3.1: Beispiel für freie Variation

[17] vgl. [BL94] S. 160f.

Konjunktionen, deren Einträge allerdings so selbsterklärend sind, dass sie hier nicht gesondert aufgeführt werden müssen. Allerdings sind hier auch einige komplette Nominamphrasen enthalten. Viele davon werden wieder durch Pronomen ausgedrückt, wie z.B. `everybody_NP`, `somebody_NP`, `something_NP`, `nobody_NP` und `nothing_NP`. Diese werden allgemein durch Indefinitpronomina ausgedrückt. Allerdings müssen sie unter Berücksichtigung aller Kasus-Formen, dem Genus und dem Numerus mit Hilfe der Funktion `regNP` in die Form einer Nominalphrase gebracht werden. Lediglich `everything_NP` wird passender durch das Nomen *omnis* im Plural ausgedrückt. Auch hier kommt die Funktion `regNP` unter Angabe aller Kasus-Formen zum Einsatz.

3.1.4. Offene Kategorien

Das Lexikon der offenen Kategorien, **LexiconLat.gf**, ethält eine kleine Anzahl aus Wörtern aus den sogenannten offenen Kategorien, vornehmlich Nomen, Verben und Adjektive. Die Einträge haben unterschiedliche Umfang. Dies ist zum einen abhängig von der Menge an Informationen, die nötig ist um das gesamte Paradigma des generieren. Wovon dies abhängig ist wird im Kapitel über die Morphologie genau beschrieben. Im allgemeinen reicht, bei regelmäßiger Deklination[18] und Konjugation[19], eine einzelne Wortform aus um daraus das gesammte 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 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. Bie 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 3.1). 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. Des halb heißt das

[18] Nomenflektion
[19] Verbflexion

entsprechende Symbol `cousin_N` ausgederückt. Die lateinische Übersetzung ist aber *consobrinus* für den männlichen Cousin und *consobrina* für das weibliche Pendant. Auch in diesem Falle kann die freie Variation, wenn auch auf einem höheren Level eingesetzt werden. Es wird nicht nur der Wert eines Parameters variiert, sondern über zwei verschiedene, komplette Nomen-Objekte.

Ein besoderer Nomeneintrag ist allerdings 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 zunächsteinmal die Phrase aus ihren Bestandteilen konstruiert werden bevor sie in die Form eines einfachen Nomens gebracht wird. Dazu werden sowohl Syntaxfunktionen verwendet, die im entsprechenden Abschnitt beschrieben werden, um die Phrase zu erzeugen, als auch eine Hilfsfunktion `useCNasN`, die die Phrase in die Form eines einfachen Nomens bringt.

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 genauere Bedeutung dieser Funktion wird im Laufe der Morphologie genauer geschildert. Ein solches Nomen ist `science_N` das mit *literae*, der Pluralform von *litera* (Buchstabe), uebersetzt 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 Pronomen kodiert, das heißt diese Nomen haben in ihrem Lexikoneintrag zusätzlich zu den Informationen, die nötig sind das Paradigma zu generieren, auch die Informationen zur Verwendung in Form der Angabe eines oder mehrerer Pronomens.

Nun bleiben noch einige der moderneren Begriffe aus dem Bereich der Nomen zu nennen, nämlich `airplane_N`, `bank_N`, `bike_N`, `car_N`, `carpet_N`, `computer_N`, `fridge_N`, `paper_N`, `planet_N`, `plastic_N`, `radio_N`, `train_N` und noch einige mehr. Die Übersetzungen dieser Begriffe wurden nach geschilderter Methode erfolgreich gefunden. Abgesehn vom Auffinden einer möglichen Übersetzung ist die Bildung allerdings relativ unproblematisch.

Die Einträge für Adjektive in diesem Lexikon sind alles in allem unproblematisch. Adjektive der ersten und zweiten Deklination können wieder aus einer einzigern Wortform, der maskulinen Nominativ-Singular-Form, erstellt werden. Lediglich bei

Adjektiven der dritten Deklination wird zusätzlich die Genitiv-Singular-Form benötigt. Es gibt auch im Bereich des Lexikons kein Adjektiv dessen Übersetzung in irgendeiner Form problematisch gewesen wäre.

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 zwei einzigen Verben, deren Übersetzung problematisch war, sind `switch8off_V2` und `switch8on_V2`. Da hier auch nicht die Wikipedia von Hilfe sein konnte, wurden zwei naheliegende 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 sollten alle problematischen oder interessanten Aspekte des Lexikons erwähnt worden sein. 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 Lateingrammtik, ist die Morphologie. Dies bedingt daraus, dass Latein eine flektierende Sprache ist, und deshalb viele grammatische Merkmale in der Wortform kodiert. Dadurch beding ist eine große Menge an Wortformen für jeden Lexikoneintrag. Um so wichtiger ist es, möglichst viele dieser Formen mit möglichst wenig Informationen zu generieren. Deshalb ist es ratsam, das Konzept der sogenannten Smart Paradigms zu implementieren. Dabei wird versucht Mit Hilfe von Stringanalysen möglichst viele Informationen zur Wortbildung zu extrahieren. Im Falle der lateinischen Sprache werden dabei Wortsuffixe zu Rate gezogen. 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.

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

Listing 3.2: Beispiel für ein Smart Paradigm mit Hilfe von Pattern matching

3.2.1. Nomenflektion

Reguläre Nomenformen

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. Den zur Identifikation kann man den Kennlaut am leichtesten nach Abtrennung der Endung *-um* im Genitiv Plural erkennen. [20]

[20] vgl. [BL94] S. 21

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 Genus Femininum. Es gibt keine wirklich relevanten 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.

Bei der zweiten Deklinationsklasse gibt es eine größere Anzahl möglicher Wortausgänge, nämlich -*us*, -*um* und -*er* bzw. -*ir*. Grundätzlich sind Nomen mit dem Ausgang -*um* Neutra, Nomen mit den Endungen -*us* und -*r* Maskulina.

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. Auf Grund dessen gehört die dritte Deklination zu den schwerer zu handhabenden Flexionsklassen. In Folge dessen reicht auch nicht eine einzige Wortform für die Generierung des Paradigmas aus. [todo: blablabla]

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* angewadt werden, da die Endung -*us* schon die zweite Deklination identifiziert. In diesem Falle wird ebenfalls das Paradigma mit Hilfe den drei Parameter Nominativ Singular, Genitiv Singular und Geschlecht bestimmt.

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.

Die Bildung der Wortformen für Nomen ist fast schon trivial. Der Wortstamm wird meist dadurch gefunden, dass man, wenn nötig, die Endung abtrennt. Anschließend werden alle zwölf Wortformen, für die zwei Numeri und die sechs Kasus, durch anfügen der passenden Endung gebildet.

Bei der ersten Deklination ist der Wortstamm angenehmerweise gleich der No-

minativ 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.[21]

Wortstamm		Endung
terr	a	e
Wortstock		Wortausgang

Tabelle 3.1.: Bestandteile eines lateinischen Nomens im Genitiv Singular (Vgl. **BAYER-LINDAUER** S. 21)

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, 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. Somit ist logischerweise der Nominativ Singular nicht einheitlich sondern hat die 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.[22]. 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 allen drei Fällen Formen mit der Endung *-a*.[23] 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

[21] vgl. [BL94] S.21f.
[22] vgl. [BL94] S. 24
[23] vgl. [BL94] S. 23f.

implementiert, da diese Funktionen noch von so einer niedrigen Komplexität sind, dass die Wiederverwendung von Codeteilen keinen deutlichen Vorteil bringt.

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

Die Unterscheidung ist alles andere als unproblematisch, wenn man, wie bei dieser Arbeit, darauf verzichten möchte Vokalqualitäten zu unterscheiden. Denn die Regel 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 Langvokalen. Dies würde jedoch die Anwendung der Grammatik behindern. Die Markierung im Lexikon währe 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 unterschieden werden, was kaum einem Benutzer zugemutet werden sollte.

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, an statt Regeln für die Generierung zu entwerfen. Für einige andere Nomen wird direkt festgelegt nach welchem Deklinationsschema die Formen gebildet werden sollen. So wird *nix* wie ein Nomen des *ĭ*-Stammes und *sedes*, *canis*, *iuvenis*, *mensis* und *sal* wie Nomen der Konsonantenstämmen dekliniert, obwohl dies nach den nachfolgenden Regeln nicht so wäre. Diese Wörter werden aber auch in Grammatikbüchern als aufnahmen gelistet.[24]

Die nachfolgenden Regeln 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. 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 Pattern versucht anzu-

[24] vgl. [BL94] S. 28

nähern. So kann sich bei einer Ablautung des letzten Vokals bei einer Nominativ Singular-Form auf *-ter* 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. Lediglich die Regel, dass sich auch nur die Vokallänge ändern kann, konnte nicht verwirklicht werden. Relativ problemlos dagegen ist wieder die Regel, dass Nomen, deren Wortstock auf zwei oder mehr Konsonanten enden, zu den *ĭ*-Stämmen gehören. Und die letzte Regel 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 die selbe Länge, so gehört das Wort zu den *ĭ*-Stämmen, und sonst gehört es zu den Konsonantenstämmen.[25]

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 komplette 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 angehoren können, und die Akkusativ Singular-, Nominativ Plural- und Akkusativ Plural-Form geschlechtsabhängig sind, müssen diese Endungen abhängig vom Geschlecht des Wortes bestimmt werden. Dabei sind bei weiblichen und männlichen Nomen die Akkusativ Singular-Form mit der Endung *-em* und die beiden Pluralfälle bilden Formen mit der selben 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 gebildet, mit den folgenden Singularformen: 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. Bei den *ĭ*-Stämmen werden im Singular die For-

[25] vgl. [BL94] S. 26ff.

men 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, bilden den Ablativ mit der Endung *-i*, alle anderen, wie die Konsonantenstämme, mit *-e*. Im Plural weicht die Genitivform ab, denn die Endung lautet hier *-ium* statt *-um*.[26]

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 in für Dativ und Ablativ Plural explizit entfernt. Es werden also für die Erzeugung des Paradigmas zwei temporäre Zeichenketten verwendet, zum einen den Wortstamm und zum anderen den Wortstamm mit dem *-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. Dann den Wortstamm, der zwei mal mit Endungen vorkommt. Und schließlich den Wortstamm mit der Endung *-ua*, der immerhin drei mal im Paradigma vertreten ist.[27]

Und die letzte Deklinationsklasse, 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 *-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.[28]

Betrachtet man die Nomenendungen im Gesamtüberblick so kann man auch ober-

[26] vgl. [BL94] S. 28
[27] vgl. [BL94] S. 33
[28] vgl. [BL94] S. 34

halb der Deklinationsklassen Muster erkenne, 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.

Sonderformen

Die häufigste Sonderform von Nomen düften 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 ######, die signalisiert, dass diese Formen nicht existieren.

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 ziemlich viele Wortformen, 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 die 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 auch für Makulina und Neutra jeweils analog zu Nomen der zweiten Deklination auf *-us* (Maskulina) und *-um* (Neutra). Alle weiteren Adjektive werden zur 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*-Stämme. Denn diese werden auch hier wieder unterschieden.[29]

Bei Adjektiven der ersten und zweiten Deklination kann wieder das ganze Paradigma aus einer einzigen Zeichenkette erzeugt werden. Ebenfalls ist die bei Adjektiven auf *-is* und *-x* möglich, die zur dritten Deklination gehören. Sollte eine Zeichenkette

[29] vgl. [BL94] S. 38

nicht genügen, wie bei den meisten Adjektiven aus der dritten Deklinationsklasse, so gibt es, wie bereits von den Nomen bekannt, die Möglichkeit, das Paradigma aus zwei gegebenen Formen, Nominativ Singular und Genitiv Singular der maskulinen Form, zu generieren. Für sehr seltene Adjektive, für die auch diese Möglichkeit nicht ausreichend ist, kann aus drei Wortformen, nämlich den drei Nominativ Singular-Formen, das Paradigma generiert werden. Diese Option wird jedoch im bisherigen Lexikon nicht benötigt.[30]

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 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 Geniti 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 auftretten, so muss man zur Erzeugung des Paradigmas die Funktion verwenden, die die drei Nominativ Singular-Formen verwendet.[31]

Das Paradigma für Adjektive der ersten und zweiten Deklination wird folgendermaßen gebildet. Zunächst wird der Wortstamm bestimmt. Normalerweise entspricht der Wortstamm der Nominativ Singular-Form ohne die geschlechtsspezifische Endung. Also in diesem Fall, bei Maskulina, *-us*. Ended 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 genau die oben genannte Methode verwendet, um ein Nomenparadigma der zweiten Deklination auf *-us* zu bilden. Damit ist theoretisch schon ein Drittel des Adjektivparadigmas im Positiv, also der ungesteigerten Form, erzeugt. Bevor diese Steigerungsstufe ver-

[30] **ParadigmsLat.gf** und **MorphoLat.gf**
[31] vgl. [BL94] S. 36 u. S. 38

follständigt wird, werden aber zuerst die Konparativ- und Superlativformen, also die Zwei Steigerungsstufen, erzeugt. 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 entsprechendenden Adverben 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 Adverben *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 übliche Endungen angehängt werden.[32]

Die Bildung des neuen Wortstammes ist nicht ganz trivial. Zunächsteinmal 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* (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, so wie zur Generierung der Komparativformen, wird als kennzeichnende Form die Genitiv Singular-Form des bereits erstellten Nomenparadigmas verwendet. Dieser hat den Vorteil, dass er bei allen Adjektiven den wirklichen Wortstamm enthält, was im Nominativ wie schon ausgeführt, nicht der Fall ist. Der Superlativ dagegen wird aus der Nominativ-Form gebildet.[33]

Der Komparativ wird üblicherweise durch das Nominativ-Suffix *-ior* für Femina 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-

[32] vgl. [BL94] S. 36f
[33] vgl. [BL94] S. 40ff.

/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.[34]

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. Endet diese auf *-er* so werden die Suffixe *-rimus,-a,-um* verwendet, dagegen bei Wörtern auf *-lis* verwendet man 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.[35]

Für die oben genannten Adjektive, die ihre Komparation durch Adverbien und nicht durch Morphologie kodieren, wird zusätzlich zur entsprechenden Steigerungsstufe das passende Adverb gespeichert, oder wenn keines benötigt wird, der leere Zeichenketten. Die Gesamtstruktur der Adjektive wird also gebildet aus dem Positiv, der wiederum aus den Nomenparadigmen der ersten und zweiten Deklination für die drei Genera besteht, und den beschriebenen Komparativ- und Superlativformen, zusammen mit den möglicherweise benötigten Adverbien.

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. Zunächst wird die Endung von der 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. Anschließend wird das Nomenparadigma für ein Maskulinum gebildet.[36]

In diesem Falle kann nichto 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

[34] vgl. [BL94] S. 40f.
[35] vgl. [BL94] S. 42
[36] vgl. [BL94] S 38f.

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 geschlechtsunanhängige 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.[37]

Darauf folgt die Bildung der Komparativ- und Superlativformen genauso wie bei den Adjektiven der vorherigen Klasse. Abschließend werden die Nomenparadigmen für die zwei fehlenden Geschlechter, genauso wie bei der maskulinen Form, gebildet und schlußendlich zu einer Adjektivstruktur zusammengesetzt. Bei Adjektiven dieser Deklination werden die Steigerungsformen immer durch Flexion kodiert. Deshalb bleiben die Felder für die Steigerungsadverbien immer leer.

3.2.3. Verbflexion

Verben bilden im Lateinischen von allen Wortarten die meisten Formen. Denn bei finiten Verben werden folgende Merkmale unterschieden: Person, Numerus, Tempus, Modus, und Generum, 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 Genera[38] (Aktiv und Passiv). Hinzukommen infinitifische 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.[39] Für jede dieser Formen ist im Datentyp (vgl. Listing 3.3 für lateinische Verben im Grammatical Framework ein Feld vorhanden. In den Typen der Felder sind 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[40]. Dagegen gibt es im Lateinischen sogenannte Deponentia, die zwar aktivisch verwen-

[37] `noun3adj` in **ResLat.gf**
[38] vielleicht passender Diathesen
[39] vgl. [BL94] S. 66
[40] vgl. [BL94] S. 67

det werden, allerdings nur passive Formen bilden[41]. Ganz zu schweigen von all den Besonderheiten der unregelmäßigen Verben[42]. Um fehlende Verbformen zu markieren wurde eine Zeichenkette gewählt, die in Eingaben mit an Sicherheit grenzender Wahrscheinlichkeit nicht vorkommt. Die gewählte Zeichenkette ist *######*. 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 aus Haskell und anderen funktionalen Programmiersprachen bekannten[43] Maybe- oder auch Option-Datentyps ermöglicht werden. Dieser polymorphe Datentyp hat für einen konkreten Datentyp zwei mögliche Werte, entweder *Just a* mit `a` einem konkreten Wert eines anderen Datentyps, wenn solch ein wert vorhanden ist, oder `Nothing`, wenn kein Wert vorhanden ist. Diese Möglichkeit mit dem Maybe-Datentyp zu arbeiten war allerdings zu Beginn der Arbeit noch nicht gegeben.

Konjugationsklassen

In der lateinischen Sprache gibt es für die Konjugation[44] 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 -$\breve{u}$- oder -$\breve{i}$-.[45]

Diese Klassen gelten sowohl für die Konjugation der regulären Verben als auch für die Deponentia. Für die Bildung des Verbparadigmas gibt es wieder zwei Arten von Verben. 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-

[41] vgl. [BL94] S. 83
[42] vgl. [BL94] S. 105ff.
[43] ???
[44] Verbflexion
[45] vgl. [BL94] S. 68f.

Präsens-Aktiv-Form wird die 1.-Person-Singular-Präsens-Indikativ-Aktiv-Form, die 1.-Person-Singular-Perfekt-Indikativ-Aktiv-Form, so wie die Partizip-Perfekt-Passiv-Form.

Denn für die Verben der ersten, zweiten und vierten Konjugation kann die Zugehörigkeit zur entsprechenden Konjugationsklasse am Wortausgang[46] 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. 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. Endet die Verbform 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 gefolg 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 an 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.[47]

[46] Kennlaut zusammen mit der Endung
[47] vgl. [BL94] S. 68f.

```
param
  Agr = Ag Gender Number Case ; -- Agreement for NP et al.
  VActForm  = VAct VAnter VTense Number Person ;
  -- No anteriority in passive because perfect forms are built using participle
  VPassForm = VPass VTense Number Person ;
  VInfForm  = VInfActPres | VInfActPerf Gender | VInfActFut Gender
            | VInfPassPres | VInfPassPerf Gender | VinfPassFut ;
  VImpForm  = VImp1 Number | VImp2 Number Person ;
  VGerund   = VGenAcc | VGenGen |VGenDat | VGenAbl ;
  VSupine   = VSupAcc | VSupAbl ;
  VPartForm = VActPres | VActFut | VPassPerf ;

oper
  Verb : Type = {
    act   : VActForm => Str ;
    pass  : VPassForm => Str ;
    inf   : VInfForm => Str ;
    imp   : VImpForm => Str ;
    ger   : VGerund => Str ;
    geriv : Agr => Str ;
    sup   : VSupine => Str ;
    part  : VPartForm =>Agr => Str ;
    } ;
```

Listing 3.3: Datentyp eines Verbs im Grammatical Framework

Verbstämme

Für jede Konjugationsklasse werden nun einige Wortformen und -stämme gebildet, aus denen das gesammte 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 in 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 Perfekt dagegen wird verwendet um die Perfekt-, Plusquamperfekt- und Futur II-Formen im Aktiv so wie den Infinitiv Perfekt im Aktiv zu bilden. Und zuletzt den Partizipialstamm, von dem die Perfek-, Plusquamperfekt- und Futur II-Formen im Passiv, zusammen mit dem Hilfsverb esse, so wie das Partizip und der Infinitiv Futur im Aktiv gebildet werden.[48]

Zur zusätzlichen Erleichterung bei der Formenbildung werden zusätzlich zu den drei Verbstämmen für jede Zeitform und jeden Modus der entsprechende Worstock und der Infinitiv-Präsens-Aktiv verwendet. Alles in allem wird das gesammte Ver-

[48] vgl. [BL94] S. 66

bparadigma also aus 15, bei Deponentia lediglich 9 wegen des unvollständigen Paradigmas, verschiedenen Zeichenketten durch Anhängen der Endungen gebildet, die die Merkmale wie Person, Numerus, Diathese, etc. kodieren. In der lateinischen Sprache werden grammatische Merkmale bei Verben an verschiedenen Stellen kodiert. Einierseits 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, das Zeitverhältnis[49], der Modus des Imperativs, vor allem jedoch Numerus und Person, kodiert.[50]

Diese 15 bzw. 9 Formen werden für jede Konjugationsklasse unabhängig gebildet. Für reguläre Verben und Deponentia der ersten, zweiten und dritten Konjugation werden alle Zeichenkette 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 Worstock 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 Worstock 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 Formen[51] 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-

[49]???

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

[51]Perfekt, Plusquamperfekt und Futur II im gegensatz zu Präsens, Imperfekt und Futur

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 Veben und Deponentia der zweiten Konjugation wird zusätzlich der Kennvokal *-ē-* zu einem *-i-*. Zusammen mit der als Ausgangsbasis gewählten Inifnitiv sind damit alle 15 bzw. 9 Formen bzw. Formenbestandteile gebildet, die verwendet werden um das Paradigma zu generieren.[52]

Die Bildung der soeben genannten Formen ist für die Verben und Deponentia der dritten Konjugation etwas anders, vor allem weil die Formen 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 unterschielich gebildet. Zunächst wird bei beiden die Infinitiv-Endung, diesmal *-ere*, abgetrennt. Und bei den kurzvokalischen Stämmen wird stattdessen ein Suffix *-i-* angefügt. Bei den Deponentia wird bei den Konsonantenstämmen die Endung *-or* von der 1.-Person-Singular Präsens-Indikativ-Aktiv abgetrennt, bei den Kurzvokalstämmen fällt der Präsensstamm mit der 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 zusammen, bei den kurzvokaligen 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 Perfketstamm gebildet. Der Partizipstamm, als letzte nötige Zeichenkette, wird aus der gegebenen Partizip-Form durch Abtrennen der Endung *-us* gebildet.[53]

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 zunächsteinmal anhand des Modus in zwei Gruppen unterteilt, die Endungen für Indikativ und Konjunktiv so wie die Imperativ-Endungen. Erstere sind

[52] vgl. [BL94] S. 68ff.
[53] 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[54] (*-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).[55] Allerdings kommen dabei immer wieder Lautgesetze zu tragen, weswegen es immer wieder besonderheiten bei der Formenbildung gibt. So werden, wenn der entsprechende Wortstamm auf einen gewissen Laut endet, so wird bei einigen Formen zwischen Stamm und Endung noch ein zusätzlicher Vokal eingefügt.

Reguläre Formenbildung

Zunächsteinmal soll die Paradigmenbildung der regulären Verben geschildert werden. 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-Stamm, 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* eingefüft, und 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 einfach 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 anschließend

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

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. Wobei bei der 1. Person Singular wieder ein Sonderfall eintritt. 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.[56]

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, einige allerdings weichen vom grundlegenden Schema 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 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 dir Endung einfach an den Präsens-Stamm angehängt. Die restlichen Formen des Präsens-Indikatik verhalten sich wie bei den 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 vehält 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.[57]

Damit sind die häufigsten Formen des Verparadigmas 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 Infiinitiv-Perfekt-Aktiv-Form dagegen muss aus dem Perfekt-

[56] vgl. [BL94] S. 74f., S. 78f. u. S. 84f.
[57] vgl. [BL94] S. 76f u. S. 80f.

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 las 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*. Und der Infinitiv-Futur-Passiv wird aus dem Partizipialstamm mit der Endung *-um* und dem Verbform *iri*[58] gebildet.

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 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 er endet auf einen Konsonanten. Dann wird der Vokal *-e* 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 ganau 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.[59]

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

Das Gerundiv hingegen wird adjektivisch verwendet und verhält sich so wie ein drei-endiges Adjektiv der ersten und zweiten Deklination. Die Nominativformen sind ananlog zum Gerung und bestehen aus dem Präsens-Stamm, unter Umständen ge-

[58] Infinitiv-Präsens-Passiv des Verbs *ire*
[59] vgl. [BL94] S. 82

folgt von einem *-e-* und den Endungen *-ndus*, *-nda* und *-ndum*. Die restlichen Formen werden, wie von den Adjektiven gewohnt, gebildet.

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 ein-endiges Adjektiv gebildet. Die Nominativ-Form 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*. Die beiden letzten Verbformen sind die Supin-Formen. Diese Formen werden gebildet wie die Maskulin-Akkusativ- und Maskulin-Ablativ-Singular-Form des Partizip-Perfekt-Passivs.[60]

Auf die Verwendung einiger dieser Formen wird später im Syntaxteil noch genauer eingegangen. Die meisten der soeben beschriebenen Formen ist die Verwendung außerhalb des Bereichs, der in dieser Arbeit abgedeckt ist. An dieser Stelle kann man auch einen kurzen Gedanken daran verschwenden, ob es sinnvoll ist all diese Verbformen an einer einzelnen Stelle zu bilden. Denn lediglich die Aktiv- und 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[61] ansehen, die an dieser Stelle dargestellt werden sollte. Alle anderen Formen, die hier gebildet werden, kann man dagegen als Derivationsmorphologie[62] auffassen, die als eigenständige Funktion anderorts 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

Die zweitgrößte Gruppe der lateinischen Verben nach den regulären Verben sind die sogenannten Deponentia. Deren name kommt vom Verb *deponere* (ablegen), da man sagen kann, dass diese Verben ihre aktiven Formen bzw. ihre passive Bedeutung

[60] vgl. [BL94] S. 82
[61] FlexMorph
[62] DerivMorph

abgelegt haben.[63]

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.

Den Anfang bilden wieder die Aktiv-Formen. Die Präsens-Formen sind wieder 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 *-o* 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 gefüfgt, 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.

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.

[63] vgl. [BL94] S. 83

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

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 Imperaitv-I-Singular entspricht, wenn der infinitiv-Stamm auf *-ri* endet, einfach aus dem Imperativ-Stamm, im anderen Falle aber aus dem Imperativ-Stamm bei dem der letzte Buchstabe durch ein *-e* ersetzt ist. Der Imperativ-I-Plural 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*.

Die Bildung des Gerunds ist relativ analog zur Bildung des Gerundivs 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*.

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*.

Als nächstes folgen wieder 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.

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.[64]

[64] vgl. [BL94] S. 86

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 gesondert von den andren 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 Weise gebildet, so z.B. das Kopula *esse*. 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 die Funktion verwendet um aus den gegebenen Stämmen ein komplettes Paradigma zu erzeugen. Und als letzter Schritt werden in diesem kompletten Paradigma nocheinmal 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.

Anders verhalten sich dagegen die sogenannten *Verba impersonalia*. Sie kommen gewöhnlich nur in der 3. Person Singular oder im Infinitiv auf.[65] 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 ist aus mehreren Gründen nicht unproblematisch. Zunächsteinmal gehören Pronomen zu den geschlossenenen 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 drei oder vier Wörter zu dieser Klasse gehören. Oder ob es einfacher wäre im Lexikon einfach alle Wortformen im entsprechenden Eintrag aufzulisten. Verschlimmert wird diese Problematik dadurch, dass die Wortart der Pronomen keine homegene Wortart ist, sondern aus verschiedenstens Unterklassen

[65] vgl. [BL94] S. 111

besteht, 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 Personanpronomen, werden wie Nomen dekliniert, andere, wie Possesivpronomen, werden wie eher wie Adjektive dekliniert.[66]

Es wurde deshalb ein Mittelweg gewählt. Denn für die Personal- und Possesivpronomen 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 ja bereits im Lexikonkapitel kurz erwähnt. Deshalb soll hier die Flexion von Personal- und Possesivpronomen erläutert werden. Dabei wird ein Typ names Objekt definiert, der als feste Felder Genus, Numerus und Person hat. Zusätzlich hat er zwei Tabellenfelder, eines für das diesen fixen Merkmalen 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 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 bzw. hier etwas allgemeiner, welche alternative Form das Personalpronomen in der Subjektposition annehmen kann.[67] 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.

Zunächsteinmal 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 Vokativformen *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

[66] vgl. [BL94] S. 48ff.
[67] vgl. [Glu04] S. 7585

entsprechend *tu*, *tui*, *tibi*, *te* und *te* und die possesiven Formen folgen wieder der ersten und zweiten Deklination bei den Grundforen *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. In der 3. Person Singular und Plural gibt es nun mehr Formen. Zum einen werden unterschiedliche Formen bei reflexivem und irreflexivem Gebracuh 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 Possesiv-Formen der 3. Person existieren eigentlich nicht, werden aberdurch 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*.[68]

Damit sind alle möglichen Formen der Personal- und Possesivpronomen behandelt. Um im Lexikon gezielt auf ein einzelnes Pronomen, gekennzeichnet durch Genus, Numerus und Person, zugreifen zu können, existiert die Funktion *mkPronoun*, die anhand dieser Merkmale die passenden Pronomenformen aus der Menge aller möglichen Formen auswählt. Anschließend werden in je einem Feld diese Merkmale fest gespeichert und in dem Feld `pers` das gewählte Personalpronomen so wie im Feld `poss` das entsprechende Possesivpronomen abgelegt. Die gesammte Pronomenbehandlung findet in der Datei **ResLat.gf** statt.

Nachdem hier die Formbildung 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 im allgemeinen in Form von Syntaxregeln, deren Form und Aufbau der letzte Teil der Grammatik gewidmet ist.

[68] 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. Dieses Kapitel ist auch in diesem Sinne aufgebaut. 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. Äußerungen (Utterances, `Utt`) in der Nomenklatur des Grammatical Framework.

3.3.1. Nominalphrasen

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

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

Listing 3.4: Die Syntaxregel `DetCN`

Die erste Regel names `DetCN` in der abstrakten Syntax gibt an, dass ein Determinans zusammen mit einee 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 Noen. Deshalb sind sie im Grunde aufgebaut wie Nomen, haben also ein Feld für das inherente Genus und ein `s`-Feld für die Wortformen. Zusätzlich hat ein Common Noun aber auc zwei Felder, in denen Adjektive bzw. genauer Adjektivphrasen, die das Nomen näher beschreiben, abgelegt werden. 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.[69]

[69] vgl. [BL94] S. 118

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.[70] 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.

Zunächst muss aber erklärt werden, wie Listen im Grammatical Framework erzeugt werden könne. Im Grammatical Framework werden Listentypen mit Kategorien in eckigen Klammern geschrieben. So ist `[AP]` eine Liste von `AP`s. 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 devor 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`[71] 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. Dazu wird eine fixe Zeichenkette verwendet, in diesem Falle pauschal die Konjunktion `and_Conj`. Mit diesen Regeln können Adjektivphrasen erstellt werden, die aus mehreren einzelnen `AP`s bestehen, die mit Konjunktionen verbunden sind.

Die einfachste Möglichkeit Common Nouns zu bilden ist es, einfach Nomen als 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 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 Nomengenus werden dabei einfach auch im Common Noun übernommen.

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

[70] vgl. **AdjectiveLat.gf**
[71] vgl. **Coordination.gf** in abstract

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. Solch ein Quantifikator kann mit Hilfe der `DetQuant` zu einem Determiner umgewandelt werden. Dazu wird zusätzlich ein sogenanntes 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[72]. So erhält man für die bestimmten und unbestimmten Artikel vier verschiedene Objekte der Kategorie `Det`, die alle keine Zeichenkette erzeugen, und von denen zwei das Merkmal des Singular und zwei den Plural tragen. Bestimmte und unbestimmte artikel sind also nicht wirklich zu unterscheiden, da beide ja in der gewohnten Form nicht existieren.

Damit haben wir aber endlich alle nötigen Bestandteile für die `DetCN`-Regel (Listing 3.4). 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 Variable `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 sogenannter 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

```
1 param
2    Agr = Ag Gender Number Case ; -- Agreement for NP et al.
```

Listing 3.5: Abhängiger Typ für Agreement

Numerus des Determinans übernimmt, so wie den Kasus uas `c`. Als letztes folgen die Adjektivphrasen nach dem Nomen aus dem Feld `postap` analog zu `preap`. Durch

[72] vgl. [Glu04] S. 6706

die Konkatenation 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 inherente Merkmale. Zusätzlich hat sie als Person die 3. Person, die die Verbform beeinflusst. Dies war auch schon die komplizierteste der bisher implementierten Regeln um Nominalphrasen zu erzeugen.

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 Eigennames, 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 ProDrop-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.

Diese drei Regeln Nominalphrasen zu bilden 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 Regel wovon eine einzige wirklich eine neue `NP` erzeugt. Die restlichen Regeln modifizieren lediglich die Nominalphrasen. Allerdings sind diese Regeln noch zu implementieren, wenn die Grammatik in der Zukunft erweitert werden sollte.

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 besprochen werden.

Die einfachste Form einer Nominalphrase wird aus einem intransitiven Verb gebildet, also einem Verb, dass kein Objekt benötigt. Die entsprechende Regel heißt `UseV` und benutzt lediglich eine Hilfsfunktion namens `predV`. Diese Funktion baut die Datenstruktur, die die für eine Verbalphrase nötigen Felder enthält (Listing 3.6). Dies 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`

```
oper
  VerbPhrase : Type = {
    fin : VActForm => Str ;
    inf : VInfForm => Str ;
    obj : Str ;
    adj : Agr => Str
  } ;
```

Listing 3.6: Datenstruktur für Verbalphrasen

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 Verwandlung in eine Verbalphrase auch schon beendet.

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).[73] 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 eine Präposition mit leerer Zeichenkette, die ein Akkusativobjekt nach sich zieht, wird angenommen. Die Transformation eines transitiven Verbs in eine `VPSlash` erfolgt analog zu den intransitiven Verben mit einer Funktion namens `predV2`, wobei das `V2` für transitive Verben steht. 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`.

3.3.3. Einfache Sätze

[73] vgl. [Ran11] S. 217

4. Ausblick

Literatur

[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.

Teil I.

Anhang

4.1. Quelltext

```
diff --git a/lib/src/latin/AdjectiveLat.gf b/lib/src/latin/AdjectiveLat.gf
index 9b0f345..7aa096a 100644
--- a/lib/src/latin/AdjectiveLat.gf
+++ b/lib/src/latin/AdjectiveLat.gf
@@ -1,39 +1,45 @@
 concrete AdjectiveLat of Adjective = CatLat ** open ResLat, Prelude in {


   lin

-    PositA  a = a ;
+    PositA  a = -- A -> AP
+      {
+       s = table { Ag g n c  => a.s ! Posit ! Ag g n c } ;
+      };

 {-
     ComparA a np = {
       s = \\_ => a.s ! AAdj Compar ++ "than" ++ np.s ! Nom ;
       isPre = False
       } ;

 -- $SuperlA$ belongs to determiner syntax in $Noun$.

     ComplA2 a np = {
       s = \\_ => a.s ! AAdj Posit ++ a.c2 ++ np.s ! Acc ;
       isPre = False
       } ;

     ReflA2 a = {
       s = \\ag => a.s ! AAdj Posit ++ a.c2 ++ reflPron ! ag ;
       isPre = False
       } ;

     SentAP ap sc = {
       s = \\a => ap.s ! a ++ sc.s ;
       isPre = False
       } ;
 -}

-    AdAP ada ap = {
-      s = \\g,n,c => ada.s ++ ap.s ! g ! n ! c ;
-      isPre = ap.isPre
-      } ;
+    -- AdAP ada ap = {
+    --   s = \\g,n,c => ada.s ++ ap.s ! g ! n ! c ;
+    --   isPre = ap.isPre
+    --   } ;

-    UseA2 a = A2 -> AP
+    -- UseA2 a = A2 -> AP
+    --   {
+    --    s = table { Ag g n c  => a.s ! Posit ! Ag g n c } ;
+    --   } ;

 }
diff --git a/lib/src/latin/AdverbLat.gf b/lib/src/latin/AdverbLat.gf
index b959ab8..d3cb693 100644
--- a/lib/src/latin/AdverbLat.gf
+++ b/lib/src/latin/AdverbLat.gf
@@ -1,21 +1,21 @@
 concrete AdverbLat of Adverb = CatLat ** open ResLat, Prelude in {

-  lin
+--  lin
--    PositAdvAdj a = {s = a.s ! AAdv} ;
--    ComparAdvAdj cadv a np = {
--      s = cadv.s ++ a.s ! AAdv ++ "than" ++ np.s ! Nom
--      } ;
--    ComparAdvAdjS cadv a s = {
--      s = cadv.s ++ a.s ! AAdv ++ "than" ++ s.s
--      } ;
```

```
73
74 -     PrepNP prep np = {s = appPrep prep np.s} ;
75 +--     PrepNP prep np = {s = appPrep prep np.s} ;
76
77  --     AdAdv = cc2 ;
78  --
79  --     SubjS = cc2 ;
80  ----b     AdvSC s = s ; -- this rule give stack overflow in ordinary parsing
81  --
82  --     AdnCAdv cadv = {s = cadv.s ++ "than"} ;
83  --
84  }
85 diff --git a/lib/src/latin/AllLat.gf b/lib/src/latin/AllLat.gf
86 index 79f9117..902c652 100644
87 --- a/lib/src/latin/AllLat.gf
88 +++ b/lib/src/latin/AllLat.gf
89 @@ -1,6 +1,6 @@
90  --#-path=.:../abstract:../common:prelude
91
92  concrete AllLat of AllLatAbs =
93 -  LangLat
94 --   ExtraLat
95 +  LangLat,
96 +  ExtraLat
97    ** {} ;
98 diff --git a/lib/src/latin/AllLatAbs.gf b/lib/src/latin/AllLatAbs.gf
99 index 7b7af3f..ba468a1 100644
100 --- a/lib/src/latin/AllLatAbs.gf
101 +++ b/lib/src/latin/AllLatAbs.gf
102 @@ -1 +1,7 @@
103 -abstract AllLatAbs = Lang ;
104 +--#-path=.:../abstract:../common:prelude
105 +
106 +abstract AllLatAbs =
107 +  Lang,
108 +  IrregLatAbs-[can_VV,go_V],
109 +  ExtraLatAbs
110 +  ** {} ;
111 diff --git a/lib/src/latin/CatLat.gf b/lib/src/latin/CatLat.gf
112 index 8d85714..af48f02 100644
113 --- a/lib/src/latin/CatLat.gf
114 +++ b/lib/src/latin/CatLat.gf
115 @@ -1,88 +1,91 @@
116  concrete CatLat of Cat = CommonX ** open ResLat, Prelude in {
117
118    flags optimize=all_subs ;
119
120    lincat
121
122  ---- Tensed/Untensed
123  --
124 --     S  = {s : Str} ;
125 --     QS = {s : QForm => Str} ;
126 +    S  = {s : Str} ;
127 +    QS = {s : QForm => Str} ;
128  --     RS = {s : Agr => Str ; c : Case} ; -- c for it clefts
129  --     SSlash = {s : Str ; c2 : Str} ;
130  --
131  ---- Sentence
132  --
133 -    Cl = {s : VAnter => VTense => Polarity => Str} ;
134 +    Cl = { s : Tense => Anteriority => Polarity => Order => Str } ;
135  --     ClSlash = {
136 --       s : ResLat.Tense => Anteriority => CPolarity => Order => Str ;
137 +--       s : ResLat.Tense => Anteriority => Polarity => Order => Str ;
138  --       c2 : Str
139  --       } ;
140 --     Imp = {s : CPolarity => ImpForm => Str} ;
141 +    Imp = {s : Polarity => ImpForm => Str} ;
142  --
143  ---- Question
144  --
145 --     QCl = {s : ResLat.Tense => Anteriority => CPolarity => QForm => Str} ;
146 --     IP = {s : Case => Str ; n : Number} ;
147 --     IComp = {s : Str} ;
```

```
148 ---     IDet = {s : Str ; n : Number} ;
149 ---     IQuant = {s : Number => Str} ;
150 +     QCl = {s : ResLat.Tense => Anteriority => Polarity => QForm => Str} ;
151 +     IP = {s : Case => Str ; n : Number} ;
152 +     IComp = {s : Str} ;
153 +     IDet = Determiner ; ---{s : Str ; n : Number} ;
154 +     IQuant = {s : Agr => Str} ;
155 ---
156 ----- Relative
157 ---
158 ---     RCl = {
159 ---       s : ResLat.Tense => Anteriority => CPolarity => Agr => Str ;
160 ---       c : Case
161 ---       } ;
162 ---     RP = {s : RCase => Str ; a : RAgr} ;
163 ---
164 ----- Verb
165 ---
166 -     VP = ResLat.VP ;
167 -     VPSlash = ResLat.VP ** {c2 : Preposition} ;
168 -     Comp = {s : Gender => Number => Case => Str} ;
169 +     VP = ResLat.VerbPhrase ;
170 +     VPSlash = VP ** {c2 : Preposition} ;
171 +     Comp = {s : Agr => Str} ;
172 ---
173 ----- Adjective
174 ---
175 -     AP = Adjective ** {isPre : Bool} ; ----- {s : Agr => Str ; isPre : Bool} ;
176 +---     AP = Adjective ** {isPre : Bool} ; ----- {s : Agr => Str ; isPre : Bool} ;
177 +     AP =
178 +       {
179 +        s : Agr => Str ;
180 +---     isPre : Bool ; --- should have no use in latin because adjectives can appear variably before and after nouns
181 +       } ;
182 ---
183 ----- Noun
184 ---
185 -     CN = {s : Number => Case => Str ; g : Gender} ;
186 -     NP, Pron = {s : Case => Str ; g : Gender ; n : Number ; p : Person} ;
187 +     CN = ResLat.CommonNoun ;
188 +     NP = ResLat.NounPhrase ;
189 +     Pron = ResLat.Pronoun ;
190       Det = Determiner ;
191 ---     Predet, Ord = {s : Str} ;
192 +     Predet, Ord = {s : Str} ;
193       Num = {s : Gender => Case => Str ; n : Number} ;
194 ---     Card = {s : Str ; n : Number} ;
195       Quant = Quantifier ;
196 ---
197 ----- Numeral
198 ---
199 ---     Numeral = {s : CardOrd => Str ; n : Number} ;
200       Digits = {s : Str ; unit : Unit} ;
201 ---
202 ----- Structural
203 ---
204       Conj = {s1,s2 : Str ; n : Number} ;
205 ---     Subj = {s : Str} ;
206 -     Prep = {s : Str ; c : Case} ;
207 +     Subj = {s : Str} ;
208 +     Prep = ResLat.Preposition ;
209 ---
210 ----- Open lexical classes, e.g. Lexicon
211
212 -     V = Verb ;
213 -     V2 = Verb ** {c : Preposition} ;
214 ---     V, VS, VQ, VA = Verb ; --- = {s : VForm => Str} ;
215 ---     V2, V2A, V2Q, V2S = Verb ** {c2 : Str} ;
216 ---     V3 = Verb ** {c2, c3 : Str} ;
217 ---     VV = {s : VVForm => Str ; isAux : Bool} ;
218 ---     V2V = Verb ** {c2 : Str ; isAux : Bool} ;
219 ---
220 -     A = Adjective ** {isPre : Bool} ;
221 ---     A2 = {s : AForm => Str ; c2 : Str} ;
222 ---
```

```
+    V, VS, VQ, VA = ResLat.Verb ; --- = {s : VForm => Str} ;
+    V2, V2A, V2Q, V2S = Verb ** {c : Prep } ;
+    V3 = Verb ** {c2, c3 : Prep} ;
+    VV = ResLat.VV ;
+    V2V = Verb ** {c2 : Str ; isAux : Bool} ;
+
+    A = Adjective ;
+
     N = Noun ;
---    N2 = {s : Number => Case => Str ; g : Gender} ** {c2 : Str} ;
---    N3 = {s : Number => Case => Str ; g : Gender} ** {c2,c3 : Str} ;
-    PN = {s : Case => Str ; g : Gender} ;
---
+    N2 = Noun ** { c : Prep } ;
+    N3 = Noun ** { c : Prep ; c2 : Prep } ;
+    PN = Noun ;
+    A2 = Adjective ** { c : Prep} ;
 }
diff --git a/lib/src/latin/ConjunctionLat.gf b/lib/src/latin/ConjunctionLat.gf
index a857eef..125767d 100644
--- a/lib/src/latin/ConjunctionLat.gf
+++ b/lib/src/latin/ConjunctionLat.gf
@@ -1,60 +1,57 @@
---concrete ConjunctionLat of Conjunction =
---  CatLat ** open ResLat, Coordination, Prelude in {
+concrete ConjunctionLat of Conjunction =
+  CatLat ** open ResLat, StructuralLat, Coordination, Prelude in {
   --
   flags optimize=all_subs ;
   --
---  lin
+  lin
   --
---    ConjS = conjunctDistrSS ;
+    ConjS = conjunctDistrSS ;
   --
---    ConjAdv = conjunctDistrSS ;
+    ConjAdv = conjunctDistrSS ;
   --
---    ConjNP conj ss = conjunctDistrTable Case conj ss ** {
---      a = conjAgr (agrP3 conj.n) ss.a
---      } ;
+    ConjNP conj ss = conjunctDistrTable Case conj ss ** {
+      a = conjAgr (agrP3 conj.n) ss.a
+      } ;
   --
---    ConjAP conj ss = conjunctDistrTable Agr conj ss ** {
---      isPre = ss.isPre
---      } ;
+    ConjAP conj ss = conjunctDistrTable Agr conj ss ** {
+      isPre = ss.isPre
+      } ;
   --
---{--b
+{--b
   --
---    ConjS = conjunctSS ;
---    DConjS = conjunctDistrSS ;
+    ConjS = conjunctSS ;
+    DConjS = conjunctDistrSS ;
   --
---    ConjAdv = conjunctSS ;
---    DConjAdv = conjunctDistrSS ;
+    ConjAdv = conjunctSS ;
+    DConjAdv = conjunctDistrSS ;
   --
---    ConjNP conj ss = conjunctTable Case conj ss ** {
---      a = conjAgr (agrP3 conj.n) ss.a
---      } ;
---    DConjNP conj ss = conjunctDistrTable Case conj ss ** {
---      a = conjAgr (agrP3 conj.n) ss.a
---      } ;
+    ConjNP conj ss = conjunctTable Case conj ss ** {
+      a = conjAgr (agrP3 conj.n) ss.a
+      } ;
+    DConjNP conj ss = conjunctDistrTable Case conj ss ** {
+      a = conjAgr (agrP3 conj.n) ss.a
+      } ;
   --
---    ConjAP conj ss = conjunctTable Agr conj ss ** {
---      isPre = ss.isPre
---      } ;
---    ConjAP conj ss = conjunctTable Agr conj ss ;
---
+    ConjAP conj ss = conjunctTable Agr conj ss ** {
+      isPre = ss.isPre
+      } ;
+    ConjAP conj ss = conjunctTable Agr conj ss ;
+
     DConjAP conj ss = conjunctDistrTable Agr conj ss ** {
       isPre = ss.isPre
       } ;
---}
+}
   --
--- These fun's are generated from the list cat's.
+  -- These fun's are generated from the list cat's.
   --
```

```
298  --      BaseS = twoSS ;
299  --      ConsS = consrSS comma ;
300 ---      BaseAdv = twoSS ;
301 ---      ConsAdv = consrSS comma ;
302 +     BaseAdv = twoSS ;
303 +     ConsAdv = consrSS "et" ;
304  --      BaseNP x y = twoTable Case x y ** {a = conjAgr x.a y.a} ;
305  --      ConsNP xs x = consrTable Case comma xs x ** {a = conjAgr xs.a x.a} ;
306 ---      BaseAP x y = twoTable Agr x y ** {isPre = andB x.isPre y.isPre} ;
307 ---      ConsAP xs x = consrTable Agr comma xs x ** {isPre = andB xs.isPre x.isPre} ;
308 +     BaseAP x y = lin A ( twoTable Agr x y ) ;
309 +     ConsAP xs x = lin A ( consrTable Agr and_Conj.s2 xs x );
310  --
311 ---   lincat
312 ---      [S] = {s1,s2 : Str} ;
313 ---      [Adv] = {s1,s2 : Str} ;
314 +   lincat
315 +      [S] = {s1,s2 : Str} ;
316 +      [Adv] = {s1,s2 : Str} ;
317  --      [NP] = {s1,s2 : Case => Str ; a : Agr} ;
318 ---      [AP] = {s1,s2 : Agr => Str ; isPre : Bool} ;
319 +      [AP] = {s1,s2 : Agr => Str } ;
320  --
321 ---}
322 +}
323  diff --git a/lib/src/latin/ExtraLat.gf b/lib/src/latin/ExtraLat.gf
324  index 9dad2f9..e6c8728 100644
325 --- a/lib/src/latin/ExtraLat.gf
326 +++ b/lib/src/latin/ExtraLat.gf
327 @@ -1,3 +1,11 @@
328   concrete ExtraLat of ExtraLatAbs = CatLat **
329     open ResLat, Coordination, Prelude in {
330 +   lin
331 +     UsePronNonDrop p = -- Pron -> NP
332 +       {
333 +         g = p.g ;
334 +         n = p.n ;
335 +         p = p.p ;
336 +         s = p.pers ! PronNonDrop ! PronRefl ;
337 +       } ;
338   }
339  diff --git a/lib/src/latin/ExtraLatAbs.gf b/lib/src/latin/ExtraLatAbs.gf
340  new file mode 100644
341  index 0000000..f2d1729
342 --- /dev/null
343 +++ b/lib/src/latin/ExtraLatAbs.gf
344 @@ -0,0 +1,5 @@
345 +abstract ExtraLatAbs = Extra ** {
346 +   fun
347 +     UsePronNonDrop : Pron -> NP ;
348 +
349 +}
350  diff --git a/lib/src/latin/GrammarLat.gf b/lib/src/latin/GrammarLat.gf
351  index efa7f69..09966e9 100644
352 --- a/lib/src/latin/GrammarLat.gf
353 +++ b/lib/src/latin/GrammarLat.gf
354 @@ -1,19 +1,18 @@
355  --#-path=.:../abstract:../common:prelude
356
357   concrete GrammarLat of Grammar =
358     NounLat,
359     VerbLat,
360     AdjectiveLat,
361     AdverbLat,
362     NumeralLat,
363     SentenceLat,
364  --   QuestionLat,
365  --   RelativeLat,
366  --   ConjunctionLat,
367 ---   PhraseLat,
368 +   PhraseLat,
369     TextX,
370 -   StructuralLat,
371 +   StructuralLat
372  --   IdiomLat
```

```
373 -   TenseX
374       ** {
375    } ;
376  diff --git a/lib/src/latin/IrregLat.gf b/lib/src/latin/IrregLat.gf
377  index 20657f0..2c3cb65 100644
378  --- a/lib/src/latin/IrregLat.gf
379  +++ b/lib/src/latin/IrregLat.gf
380  @@ -1,181 +1,458 @@
381  ----# -path=.:prelude:../abstract:../common
382  ----
383  ----concrete IrregLat of IrregLatAbs = CatLat ** open ParadigmsLat in {
384  +--# -path=.:prelude:../abstract:../common
385  +
386  +concrete IrregLat of IrregLatAbs = CatLat ** open Prelude, ParadigmsLat, ResLat in {
387   --
388   --flags optimize=values ;
389   --
390  ----  lin
391  ----  awake_V = irregV "awake" "awoke" "awoken" ;
392  ----  bear_V = irregV "bear" "bore" "born" ;
393  ----  beat_V = irregV "beat" "beat" "beat" ;
394  ----  become_V = irregV "become" "became" "become" ;
395  ----  begin_V = irregV "begin" "began" "begun" ;
396  ----  bend_V = irregV "bend" "bent" "bent" ;
397  ----  beset_V = irregV "beset" "beset" "beset" ;
398  ----  bet_V = irregDuplV "bet" "bet" "bet" ;
399  ----  bid_V = irregDuplV "bid" (variants {"bid" ; "bade"}) (variants {"bid" ; "bidden"}) ;
400  ----  bind_V = irregV "bind" "bound" "bound" ;
401  ----  bite_V = irregV "bite" "bit" "bitten" ;
402  ----  bleed_V = irregV "bleed" "bled" "bled" ;
403  ----  blow_V = irregV "blow" "blew" "blown" ;
404  ----  break_V = irregV "break" "broke" "broken" ;
405  ----  breed_V = irregV "breed" "bred" "bred" ;
406  ----  bring_V = irregV "bring" "brought" "brought" ;
407  ----  broadcast_V = irregV "broadcast" "broadcast" "broadcast" ;
408  ----  build_V = irregV "build" "built" "built" ;
409  ----  burn_V = irregV "burn" (variants {"burned" ; "burnt"}) (variants {"burned" ; "burnt"}) ;
410  ----  burst_V = irregV "burst" "burst" "burst" ;
411  ----  buy_V = irregV "buy" "bought" "bought" ;
412  ----  cast_V = irregV "cast" "cast" "cast" ;
413  ----  catch_V = irregV "catch" "caught" "caught" ;
414  ----  choose_V = irregV "choose" "chose" "chosen" ;
415  ----  cling_V = irregV "cling" "clung" "clung" ;
416  ----  come_V = irregV "come" "came" "come" ;
417  ----  cost_V = irregV "cost" "cost" "cost" ;
418  ----  creep_V = irregV "creep" "crept" "crept" ;
419  ----  cut_V = irregDuplV "cut" "cut" "cut" ;
420  ----  deal_V = irregV "deal" "dealt" "dealt" ;
421  ----  dig_V = irregDuplV "dig" "dug" "dug" ;
422  ----  dive_V = irregV "dive" (variants {"dived" ; "dove"}) "dived" ;
423  ----  do_V = mk5V "do" "does" "did" "done" "doing" ;
424  ----  draw_V = irregV "draw" "drew" "drawn" ;
425  ----  dream_V = irregV "dream" (variants {"dreamed" ; "dreamt"}) (variants {"dreamed" ; "dreamt"}) ;
426  ----  drive_V = irregV "drive" "drove" "driven" ;
427  ----  drink_V = irregV "drink" "drank" "drunk" ;
428  ----  eat_V = irregV "eat" "ate" "eaten" ;
429  ----  fall_V = irregV "fall" "fell" "fallen" ;
430  ----  feed_V = irregV "feed" "fed" "fed" ;
431  ----  feel_V = irregV "feel" "felt" "felt" ;
432  ----  fight_V = irregV "fight" "fought" "fought" ;
433  ----  find_V = irregV "find" "found" "found" ;
434  ----  fit_V = irregDuplV "fit" "fit" "fit" ;
435  ----  flee_V = irregV "flee" "fled" "fled" ;
436  ----  fling_V = irregV "fling" "flung" "flung" ;
437  ----  fly_V = irregV "fly" "flew" "flown" ;
438  ----  forbid_V = irregDuplV "forbid" "forbade" "forbidden" ;
439  ----  forget_V = irregDuplV "forget" "forgot" "forgotten" ;
440  ----  forgive_V = irregV "forgive" "forgave" "forgiven" ;
441  ----  forsake_V = irregV "forsake" "forsook" "forsaken" ;
442  ----  freeze_V = irregV "freeze" "froze" "frozen" ;
443  ----  get_V = irregDuplV "get" "got" "gotten" ;
444  ----  give_V = irregV "give" "gave" "given" ;
445  ----  go_V = mk5V "go" "goes" "went" "gone" "going" ;
446  ----  grind_V = irregV "grind" "ground" "ground" ;
447  ----  grow_V = irregV "grow" "grew" "grown" ;
```

```
448 ---- hang_V = irregV "hang" "hung" "hung" ;
449 ---- have_V = mk5V "have" "has" "had" "had" "having" ;
450 ---- hear_V = irregV "hear" "heard" "heard" ;
451 ---- hide_V = irregV "hide" "hid" "hidden" ;
452 ---- hit_V = irregDuplV "hit" "hit" "hit" ;
453 ---- hold_V = irregV "hold" "held" "held" ;
454 ---- hurt_V = irregV "hurt" "hurt" "hurt" ;
455 ---- keep_V = irregV "keep" "kept" "kept" ;
456 ---- kneel_V = irregV "kneel" "knelt" "knelt" ;
457 ---- knit_V = irregDuplV "knit" "knit" "knit" ;
458 ---- know_V = irregV "know" "knew" "know" ;
459 ---- lay_V = irregV "lay" "laid" "laid" ;
460 ---- lead_V = irregV "lead" "led" "led" ;
461 ---- leap_V = irregV "leap" (variants {"leaped" ; "lept"}) (variants {"leaped" ; "lept"}) ;
462 ---- learn_V = irregV "learn" (variants {"learned" ; "learnt"}) (variants {"learned" ; "learnt"}) ;
463 ---- leave_V = irregV "leave" "left" "left" ;
464 ---- lend_V = irregV "lend" "lent" "lent" ;
465 ---- let_V = irregDuplV "let" "let" "let" ;
466 ---- lie_V = irregV "lie" "lay" "lain" ;
467 ---- light_V = irregV "light" (variants {"lighted" ; "lit"}) "lighted" ;
468 ---- lose_V = irregV "lose" "lost" "lost" ;
469 ---- make_V = irregV "make" "made" "made" ;
470 ---- mean_V = irregV "mean" "meant" "meant" ;
471 ---- meet_V = irregV "meet" "met" "met" ;
472 ---- misspell_V = irregV "misspell" (variants {"misspelled" ; "misspelt"}) (variants {"misspelled" ; "misspelt"}) ;
473 ---- mistake_V = irregV "mistake" "mistook" "mistaken" ;
474 ---- mow_V = irregV "mow" "mowed" (variants {"mowed" ; "mown"}) ;
475 ---- overcome_V = irregV "overcome" "overcame" "overcome" ;
476 ---- overdo_V = mk5V "overdo" "overdoes" "overdid" "overdone" "overdoing" ;
477 ---- overtake_V = irregV "overtake" "overtook" "overtaken" ;
478 ---- overthrow_V = irregV "overthrow" "overthrew" "overthrown" ;
479 ---- pay_V = irregV "pay" "paid" "paid" ;
480 ---- plead_V = irregV "plead" "pled" "pled" ;
481 ---- prove_V = irregV "prove" "proved" (variants {"proved" ; "proven"}) ;
482 ---- put_V = irregDuplV "put" "put" "put" ;
483 ---- quit_V = irregDuplV "quit" "quit" "quit" ;
484 ---- read_V = irregV "read" "read" "read" ;
485 ---- rid_V = irregDuplV "rid" "rid" "rid" ;
486 ---- ride_V = irregV "ride" "rode" "ridden" ;
487 ---- ring_V = irregV "ring" "rang" "rung" ;
488 ---- rise_V = irregV "rise" "rose" "risen" ;
489 ---- run_V = irregDuplV "run" "ran" "run" ;
490 ---- saw_V = irregV "saw" "sawed" (variants {"sawed" ; "sawn"}) ;
491 ---- say_V = irregV "say" "said" "said" ;
492 ---- see_V = irregV "see" "saw" "seen" ;
493 ---- seek_V = irregV "seek" "sought" "sought" ;
494 ---- sell_V = irregV "sell" "sold" "sold" ;
495 ---- send_V = irregV "send" "sent" "sent" ;
496 ---- set_V = irregDuplV "set" "set" "set" ;
497 ---- sew_V = irregV "sew" "sewed" (variants {"sewed" ; "sewn"}) ;
498 ---- shake_V = irregV "shake" "shook" "shaken" ;
499 ---- shave_V = irregV "shave" "shaved" (variants {"shaved" ; "shaven"}) ;
500 ---- shear_V = irregV "shear" "shore" "shorn" ;
501 ---- shed_V = irregDuplV "shed" "shed" "shed" ;
502 ---- shine_V = irregV "shine" "shone" "shone" ;
503 ---- shoe_V = irregV "shoe" "shoed" (variants {"shoed" ; "shod"}) ;
504 ---- shoot_V = irregV "shoot" "shot" "shot" ;
505 ---- show_V = irregV "show" "showed" (variants {"showed" ; "shown"}) ;
506 ---- shrink_V = irregV "shrink" "shrank" "shrunk" ;
507 ---- shut_V = irregDuplV "shut" "shut" "shut" ;
508 ---- sing_V = irregV "sing" "sang" "sung" ;
509 ---- sink_V = irregV "sink" "sank" "sunk" ;
510 ---- sit_V = irregDuplV "sit" "sat" "sat" ;
511 ---- sleep_V = irregV "sleep" "slept" "slept" ;
512 ---- slay_V = irregV "slay" "slew" "slain" ;
513 ---- slide_V = irregV "slide" "slid" "slid" ;
514 ---- sling_V = irregV "sling" "slung" "slung" ;
515 ---- slit_V = irregDuplV "slit" "slit" "slit" ;
516 ---- smite_V = irregV "smite" "smote" "smitten" ;
517 ---- sow_V = irregV "sow" "sowed" (variants {"sowed" ; "sown"}) ;
518 ---- speak_V = irregV "speak" "spoke" "spoken" ;
519 ---- speed_V = irregV "speed" "sped" "sped" ;
520 ---- spend_V = irregV "spend" "spent" "spent" ;
521 ---- spill_V = irregV "spill" (variants {"spilled" ; "spilt"}) (variants {"spilled" ; "spilt"}) ;
522 ---- spin_V = irregDuplV "spin" "spun" "spun" ;
```

```
523 --   spit_V = irregDuplV "spit" (variants {"spit" ; "spat"}) "spit" ;
524 --   split_V = irregDuplV "split" "split" "split" ;
525 --   spread_V = irregV "spread" "spread" "spread" ;
526 --   spring_V = irregV "spring" (variants {"sprang" ; "sprung"}) "sprung" ;
527 --   stand_V = irregV "stand" "stood" "stood" ;
528 --   steal_V = irregV "steal" "stole" "stolen" ;
529 --   stick_V = irregV "stick" "stuck" "stuck" ;
530 --   sting_V = irregV "sting" "stung" "stung" ;
531 --   stink_V = irregV "stink" "stank" "stunk" ;
532 --   stride_V = irregV "stride" "strod" "stridden" ;
533 --   strike_V = irregV "strike" "struck" "struck" ;
534 --   string_V = irregV "string" "strung" "strung" ;
535 --   strive_V = irregV "strive" "strove" "striven" ;
536 --   swear_V = irregV "swear" "swore" "sworn" ;
537 --   sweep_V = irregV "sweep" "swept" "swept" ;
538 --   swell_V = irregV "swell" "swelled" (variants {"swelled" ; "swollen"}) ;
539 --   swim_V = irregDuplV "swim" "swam" "swum" ;
540 --   swing_V = irregV "swing" "swung" "swung" ;
541 --   take_V = irregV "take" "took" "taken" ;
542 --   teach_V = irregV "teach" "taught" "taught" ;
543 --   tear_V = irregV "tear" "tore" "torn" ;
544 --   tell_V = irregV "tell" "told" "told" ;
545 --   think_V = irregV "think" "thought" "thought" ;
546 --   thrive_V = irregV "thrive" (variants {"thrived" ; "throve"}) "thrived" ;
547 --   throw_V = irregV "throw" "threw" "thrown" ;
548 --   thrust_V = irregV "thrust" "thrust" "thrust" ;
549 --   tread_V = irregV "tread" "trod" "trodden" ;
550 --   understand_V = irregV "understand" "understood" "understood" ;
551 --   uphold_V = irregV "uphold" "upheld" "upheld" ;
552 --   upset_V = irregDuplV "upset" "upset" "upset" ;
553 --   wake_V = irregV "wake" "woke" "woken" ;
554 --   wear_V = irregV "wear" "wore" "worn" ;
555 --   weave_V = irregV "weave" (variants {"weaved" ; "wove"}) (variants {"weaved" ; "woven"}) ;
556 --   wed_V = irregDuplV "wed" "wed" "wed" ;
557 --   weep_V = irregV "weep" "wept" "wept" ;
558 --   wind_V = irregV "wind" "wound" "wound" ;
559 --   win_V = irregDuplV "win" "won" "won" ;
560 --   withhold_V = irregV "withhold" "withheld" "withheld" ;
561 --   withstand_V = irregV "withstand" "withstood" "withstood" ;
562 --   wring_V = irregV "wring" "wrung" "wrung" ;
563 --   write_V = irregV "write" "wrote" "written" ;
564 --}
565 +
566 +  lin
567 +    science_N = pluralN (mkN "litera" ) ; -- only pl. (Langenscheidts)
568 +
569 +    -- Bayer-Lindauer 93 1
570 +    be_V =
571 +      let
572 +        pres_stem = "s" ;
573 +        pres_ind_base = "su" ;
574 +        pres_conj_base = "si" ;
575 +        impf_ind_base = "era" ;
576 +        impf_conj_base = "esse" ;
577 +        fut_I_base = "eri" ;
578 +        imp_base = "es" ;
579 +        perf_stem = "fu" ;
580 +        perf_ind_base = "fu" ;
581 +        perf_conj_base = "fueri" ;
582 +        pqperf_ind_base = "fuera" ;
583 +        pqperf_conj_base = "fuisse" ;
584 +        fut_II_base = "fueri" ;
585 +        part_stem = "fut" ;
586 +        verb = mkVerb "esse" pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base
587 +          imp_base perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
588 +      in
589 +      {
590 +        act =
591 +          table {
592 +            VAct VSim (VPres VInd)  n  p  =>
593 +              table Number [ table Person [ "sum" ; "es" ; "est" ] ;
594 +                             table Person [ "sumus" ; "estis" ; "sunt" ]
595 +              ] ! n ! p ;
596 +            a => verb.act ! a
597 +          };
```

```
598 +       pass =
599 +         \\_ => "######" ; -- no passive forms
600 +       inf =
601 +         verb.inf ;
602 +       imp =
603 +         table {
604 +           VImp1 Sg => "es" ;
605 +           VImp1 Pl => "este" ;
606 +           VImp2 Pl P2 => "estote" ;
607 +           a => verb.imp ! a
608 +         } ;
609 +       sup =
610 +         \\_ => "######" ; -- no supin forms
611 +       ger =
612 +         \\_ => "######" ; -- no gerund forms
613 +       geriv =
614 +         \\_ => "######" ; -- no gerundive forms
615 +       part = table {
616 +         VActFut =>
617 +           verb.part ! VActFut ;
618 +         VActPres =>
619 +           \\_ => "######" ; -- no such participle
620 +         VPassPerf =>
621 +           \\_ => "######" -- no such participle
622 +         }
623 +     } ;
624 +
625 +   -- Bayer-Lindauer 93 2.2
626 +   can_VV =
627 +     let
628 +       pres_stem = "pos" ;
629 +       pres_ind_base = "pos" ;
630 +       pres_conj_base = "possi" ;
631 +       impf_ind_base = "potera" ;
632 +       impf_conj_base = "posse" ;
633 +       fut_I_base = "poteri" ;
634 +       imp_base = "" ;
635 +       perf_stem = "potu" ;
636 +       perf_ind_base = "potu" ;
637 +       perf_conj_base = "potueri" ;
638 +       pqperf_ind_base = "potuera" ;
639 +       pqperf_conj_base = "potuisse" ;
640 +       fut_II_base = "potueri" ;
641 +       part_stem = "" ;
642 +       verb = mkVerb "posse" pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base
643 +         imp_base perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
644 +     in
645 +     {
646 +       act =
647 +         table {
648 +           VAct VSim (VPres VInd)  n  p  =>
649 +             table Number [ table Person [ "possum" ; "potes" ; "potest" ] ;
650 +                            table Person [ "possumus" ; "potestis" ; "possunt" ]
651 +             ] ! n ! p ;
652 +           a => verb.act ! a
653 +         } ;
654 +       pass =
655 +         \\_ => "######" ; -- no passive forms
656 +       inf =
657 +         table {
658 +           VInfActFut _ => "######" ;
659 +           a => verb.inf ! a
660 +         } ;
661 +       imp =
662 +         \\_ => "######" ;
663 +       sup =
664 +         \\_ => "######" ;
665 +       ger =
666 +         \\_ => "######" ;
667 +       geriv =
668 +         \\_ => "######" ;
669 +       part = table {
670 +         VActFut =>
671 +           \\_ => "######" ; -- no such participle
672 +         VActPres =>
```

```
673 +           \\_ => "######" ; -- no such participle
674 +         VPassPerf =>
675 +           \\_ => "######" -- no such participle
676 +         } ;
677 +       isAux = False
678 +     };
679 +
680 +   -- Bayer-Lindauer 94
681 +   bring_V =
682 +     let
683 +       pres_stem = "fer" ;
684 +       pres_ind_base = "fer" ;
685 +       pres_conj_base = "fera" ;
686 +       impf_ind_base = "fereba" ;
687 +       impf_conj_base = "ferre" ;
688 +       fut_I_base = "fere" ;
689 +       imp_base = "fer" ;
690 +       perf_stem = "tul" ;
691 +       perf_ind_base = "tul" ;
692 +       perf_conj_base = "tuleri" ;
693 +       pqperf_ind_base = "tulera" ;
694 +       pqperf_conj_base = "tulisse" ;
695 +       fut_II_base = "tuleri" ;
696 +       part_stem = "lat" ;
697 +       verb = mkVerb "ferre" pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base
698 +         imp_base perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
699 +     in
700 +     {
701 +       act =
702 +         table {
703 +           VAct VSim (VPres VInd) n p =>
704 +             table Number [ table Person [ "fero" ; "fers" ; "fert" ] ;
705 +                            table Person [ "ferimus" ; "fertis" ; "ferunt" ]
706 +             ] ! n ! p ;
707 +           a => verb.act ! a
708 +         } ;
709 +       pass =
710 +         table {
711 +           VPass (VPres VInd) n p =>
712 +             table Number [ table Person [ "feror" ; "ferris" ; "fertur" ] ;
713 +                            table Person [ "ferimur" ; "ferimini" ; "feruntur" ]
714 +             ] ! n ! p ;
715 +           a => verb.pass ! a
716 +         } ;
717 +       inf =
718 +         verb.inf ;
719 +       imp =
720 +         table {
721 +           VImp1 n => table Number [ "fer" ; "ferte" ] ! n ;
722 +           VImp2 Sg ( P2 | P3 ) => "ferto" ;
723 +           VImp2 Pl P2 => "fertote" ;
724 +           a => verb.imp ! a
725 +         } ;
726 +       sup =
727 +         verb.sup ;
728 +       ger =
729 +         verb.ger ;
730 +       geriv =
731 +         verb.geriv ;
732 +       part = verb.part ;
733 +     };
734 +
735 +   -- Bayer-Lindauer 95
736 +   want_V =
737 +     let
738 +       pres_stem = "vel" ;
739 +       pres_ind_base = "vol" ;
740 +       pres_conj_base = "veli" ;
741 +       impf_ind_base = "voleba" ;
742 +       impf_conj_base = "volle" ;
743 +       fut_I_base = "vole" ;
744 +       imp_base = "" ;
745 +       perf_stem = "volu" ;
746 +       perf_ind_base = "volu" ;
747 +       perf_conj_base = "volueri" ;
```

```
748 +       pqperf_ind_base = "voluera" ;
749 +       pqperf_conj_base = "voluisse" ;
750 +       fut_II_base = "volueri" ;
751 +       part_stem = "volet" ;
752 +       verb = mkVerb "velle" pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base
753 +         imp_base perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
754 +     in
755 +     {
756 +       act =
757 +         table {
758 +           VAct VSim (VPres VInd)  n  p  =>
759 +             table Number [ table Person [ "volo" ; "vis" ; "vult" ] ;
760 +                            table Person [ "volumus" ; "vultis" ; "volunt" ]
761 +             ] ! n ! p ;
762 +           a => verb.act ! a
763 +         } ;
764 +         pass =
765 +           \\_ => "######" ;
766 +         ger =
767 +           verb.ger ;
768 +         geriv =
769 +           verb.geriv ;
770 +         imp =
771 +           \\_ => "######" ;
772 +         inf =
773 +           verb.inf ;
774 +         part = table {
775 +           VActFut =>
776 +             \\_ => "######" ;
777 +           VActPres =>
778 +             verb.part ! VActPres ;
779 +           VPassPerf =>
780 +             \\_ => "######"
781 +           } ;
782 +         sup =
783 +           verb.sup ;
784 +     } ;
785 +
786 +   -- Bayer-Lindauer 96 1
787 +   go_V =
788 +     let
789 +       pres_stem = "i" ;
790 +       pres_ind_base = "i" ;
791 +       pres_conj_base = "ea" ;
792 +       impf_ind_base = "iba" ;
793 +       impf_conj_base = "ire" ;
794 +       fut_I_base = "ibi" ;
795 +       imp_base = "i" ;
796 +       perf_stem = "i" ;
797 +       perf_ind_base = "i" ;
798 +       perf_conj_base = "ieri" ;
799 +       pqperf_ind_base = "iera" ;
800 +       pqperf_conj_base = "isse" ;
801 +       fut_II_base = "ieri" ;
802 +       part_stem = "it" ;
803 +       verb = mkVerb "ire" pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base
804 +         imp_base perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
805 +     in
806 +     {
807 +       act =
808 +         table {
809 +           VAct VSim (VPres VInd)  n  p  =>
810 +             table Number [ table Person [ "eo" ; "is" ; "it" ] ;
811 +                            table Person [ "imus" ; "itis" ; "eunt" ]
812 +             ] ! n ! p ;
813 +           VAct VAnt (VPres VInd)  Sg P2 => "isti" ;
814 +           VAct VAnt (VPres VInd)  Pl P2 => "istis" ;
815 +           a => verb.act ! a
816 +         } ;
817 +       pass =
818 +         \\_ => "######"; -- no passive forms
819 +       ger =
820 +         table VGerund [ "eundum" ; "eundi" ; "eundo" ; "eundo" ] ;
821 +       geriv =
822 +         verb.geriv ;
```

```
823 +       imp =
824 +         table {
825 +           VImp2 Pl P3 => "eunto" ;
826 +           a => verb.imp ! a
827 +         } ;
828 +       inf =
829 +         table {
830 +           VInfActPerf _ => "isse" ;
831 +           a =>verb.inf ! a
832 +         };
833 +       part = table {
834 +         VActFut =>
835 +           verb.part ! VActFut ;
836 +         VActPres =>
837 +           table {
838 +             Ag ( Fem | Masc ) n c =>
839 +               ( mkNoun ( "iens" ) ( "euntem" ) ( "euntis" )
840 +                   ( "eunti" ) ( "eunte" ) ( "iens" )
841 +                   ( "euntes" ) ( "euntes" ) ( "euntium" )
842 +                   ( "euntibus" )
843 +                   Masc ).s ! n ! c ;
844 +             Ag Neutr n c =>
845 +               ( mkNoun ( "iens" ) ( "iens" ) ( "euntis" )
846 +                   ( "eunti" ) ( "eunte" ) ( "iens" )
847 +                   ( "euntia" ) ( "euntia" ) ( "euntium" )
848 +                   ( "euntibus" )
849 +                   Masc ).s ! n ! c
850 +           } ;
851 +         VPassPerf =>
852 +           \\_ => "######" -- no such participle
853 +         } ;
854 +       sup =
855 +         \\_ => "######" -- really no such form?
856 +     } ;
857 +
858 +   -- Bayer-Lindauer 97
859 +   become_V =
860 +     let
861 +       pres_stem = "fi" ;
862 +       pres_ind_base = "fi" ;
863 +       pres_conj_base = "fia" ;
864 +       impf_ind_base = "fieba" ;
865 +       impf_conj_base = "fiere" ;
866 +       fut_I_base = "fie" ;
867 +       imp_base = "fi" ;
868 +       perf_stem = "" ;
869 +       perf_ind_base = "" ;
870 +       perf_conj_base = "" ;
871 +       pqperf_ind_base = "" ;
872 +       pqperf_conj_base = "" ;
873 +       fut_II_base = "" ;
874 +       part_stem = "fact" ;
875 +
876 +       verb =
877 +         mkVerb "fieri" pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base
878 +         perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
879 +     in
880 +     {
881 +       act =
882 +         table {
883 +           VAct VSim (VPres VInd) Sg P1 => "fio" ;
884 +           VAct VAnt _             _ _ => "######" ; -- perfect expressed by participle
885 +           a => verb.act ! a
886 +         } ;
887 +       pass =
888 +         \\_ => "######" ; -- no passive forms
889 +       ger =
890 +         \\_ => "######" ; -- no gerund form
891 +       geriv =
892 +         \\_ => "######" ; -- no gerundive form
893 +       imp =
894 +         verb.imp ;
895 +       inf =
896 +         table {
897 +           VInfActPerf _ => "factus" ;
```

```
898 +           VInfActFut Masc => "futurum" ;
899 +           VInfActFut Fem => "futura" ;
900 +           VInfActFut Neutr => "futurum" ;
901 +           a => verb.inf ! a
902 +         } ;
903 +       part = table {
904 +         VActFut =>
905 +           \\_ => "######" ; -- no such participle
906 +         VActPres =>
907 +           \\_ => "######" ; -- no such participle
908 +         VPassPerf =>
909 +           verb.part ! VPassPerf
910 +         } ;
911 +       sup =
912 +         \\_ => "######" -- no supin
913 +     } ;
914 +
915 +   -- Source ?
916 +   rain_V =
917 +     {
918 +       act =
919 +         table {
920 +           VAct VSim (VPres VInd) Sg P3 => "pluit" ;
921 +           VAct VSim (VPres VInd) Pl P3 => "pluunt" ;
922 +           VAct VSim (VImpf VInd) Sg P3 => "pluebat" ;
923 +           VAct VSim (VImpf VInd) Pl P3 => "pluebant" ;
924 +           VAct VSim VFut Sg P3 => "pluet" ;
925 +           VAct VSim VFut Pl P3 => "pluent" ;
926 +           VAct VAnt (VPres VInd) Sg P3 => "pluvit" ;
927 +           VAct VAnt (VPres VInd) Pl P3 => "pluverunt" ;
928 +           VAct VAnt (VImpf VInd) Sg P3 => "pluverat" ;
929 +           VAct VAnt (VImpf VInd) Pl P3 => "pluverat" ;
930 +           VAct VAnt VFut Sg P3 => "pluverit" ;
931 +           VAct VAnt VFut Pl P3 => "pluverint" ;
932 +           VAct VSim (VPres VConj) Sg P3 => "pluat" ;
933 +           VAct VSim (VPres VConj) Pl P3 => "pluant" ;
934 +           VAct VSim (VImpf VConj) Sg P3 => "plueret" ;
935 +           VAct VSim (VImpf VConj) Pl P3 => "pluerent" ;
936 +           VAct VAnt (VPres VConj) Sg P3 => "pluverit" ;
937 +           VAct VAnt (VPres VConj) Pl P3 => "pluverint" ;
938 +           VAct VAnt (VImpf VConj) Sg P3 => "pluvisset" ;
939 +           VAct VAnt (VImpf VConj) Pl P3 => "pluvissent" ;
940 +           _ => "######" -- no such forms
941 +         } ;
942 +       pass =
943 +         \\_ => "######" ; -- no passive forms
944 +       inf = table {
945 +         VInfActPres => "pluere" ;
946 +         VInfActPerf _ => "pluvisse" ;
947 +         _ => "######"
948 +         } ;
949 +       imp =
950 +         table {
951 +           VImp2 Sg ( P2 | P3 ) => "pluito" ;
952 +           VImp2 Pl P2 => "pluitote" ;
953 +           VImp2 Pl P3 => "pluunto" ;
954 +           _ => "######"
955 +         } ;
956 +       ger =
957 +         \\_ => "######" ; -- no gerund forms
958 +       geriv =
959 +         \\_ => "######" ; -- no gerundive forms
960 +       sup =
961 +         \\_ => "######" ; -- no supin forms
962 +       part = table {
963 +         VActPres =>
964 +           \\_ => "pluens" ;
965 +         VActFut =>
966 +           \\_ => "######" ; -- no such participle
967 +         VPassPerf =>
968 +           \\_ => "######" -- no such participle
969 +         }
970 +     } ;
971 +
972 +   -- Bayer–Lindauer 98
```

```
973 +    hate_V =
974 +      let
975 +       pres_stem = "" ;
976 +       pres_ind_base = "" ;
977 +       pres_conj_base = "" ;
978 +       impf_ind_base = "" ;
979 +       impf_conj_base = "" ;
980 +       fut_I_base = "" ;
981 +       imp_base = "" ;
982 +       perf_stem = "od" ;
983 +       perf_ind_base = "od" ;
984 +       perf_conj_base = "oderi" ;
985 +       pqperf_ind_base = "odera" ;
986 +       pqperf_conj_base = "odissem" ;
987 +       fut_II_base = "oderi" ;
988 +       part_stem = "os" ;
989 +       verb =
990 +         mkVerb "odisse" pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base
991 +         perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
992 +      in {
993 +       act = table {
994 +         VAct VSim t n p => verb.act ! VAct VAnt t n p ;
995 +         _ => "#######" -- no such verb forms
996 +         } ;
997 +       pass = \\_ => "#######" ; -- no passive forms
998 +       ger = \\_ => "#######" ; -- no gerund forms
999 +       geriv = \\_ => "#######" ; -- no gerundive forms
1000 +       imp = \\_ => "#######" ; -- no imperative form
1001 +       inf = table {
1002 +         VInfActPres => verb.inf ! VInfActPres ;
1003 +         VInfActFut g => verb.inf ! VInfActFut g ; -- really ?
1004 +         _ => "#######"
1005 +         } ;
1006 +       part = table {
1007 +         VActFut =>
1008 +           verb.part ! VActFut ;
1009 +         VActPres =>
1010 +           \\_ => "#######" ; -- no such participle form
1011 +         VPassPerf =>
1012 +           \\_ => "#######" -- no such participle form
1013 +         } ;
1014 +       sup = \\_ => "#######" ; -- no such supine form
1015 +      } ;
1016 +}
1017 diff --git a/lib/src/latin/IrregLatAbs.gf b/lib/src/latin/IrregLatAbs.gf
1018 new file mode 100644
1019 index 0000000..a5e86f7
1020 --- /dev/null
1021 +++ b/lib/src/latin/IrregLatAbs.gf
1022 @@ -0,0 +1,12 @@
1023 +abstract IrregLatAbs = Cat ** {
1024 +  fun
1025 +    science_N : N ;
1026 +    be_V : V ;
1027 +    can_VV : VV ;
1028 +    bring_V : V ;
1029 +    want_V : V ;
1030 +    go_V : V;
1031 +    become_V : V;
1032 +    rain_V : V;
1033 +    hate_V : V;
1034 +}
1035 diff --git a/lib/src/latin/LangLat.gf b/lib/src/latin/LangLat.gf
1036 index e727a11..a61a880 100644
1037 --- a/lib/src/latin/LangLat.gf
1038 +++ b/lib/src/latin/LangLat.gf
1039 @@ -1,10 +1,12 @@
1040  --# -path=.:../abstract:../common:../prelude
1041
1042   concrete LangLat of Lang =
1043     GrammarLat,
1044 +   ParadigmsLat,
1045 +   ConjunctionLat,
1046     LexiconLat
1047     ** {
```

```
1048
1049   flags startcat = Phr ; unlexer = text ; lexer = text ;
1050
1051   } ;
1052  diff --git a/lib/src/latin/LexiconLat.gf b/lib/src/latin/LexiconLat.gf
1053  index 4c0fc16..14bdba2 100644
1054  --- a/lib/src/latin/LexiconLat.gf
1055  +++ b/lib/src/latin/LexiconLat.gf
1056  @@ -1,377 +1,395 @@
1057   --#-path=.:prelude
1058
1059  +--1 Basic Latin Lexicon.
1060  +
1061  +-- Aarne Ranta pre 2013, Herbert Lange 2013
1062  +
1063  +-- This lexicon implements all the words in the abstract Lexicon.
1064  +-- For each entry a source is given, either a printed dictionary, a
1065  +-- printed grammar book or a link to an online source. The used printed
1066  +-- dictionaries are Langescheidts Schulwörterbuch Lateinisch 17. Edition
1067  +-- 1984 (shorter: Langenscheidts), PONS Schulwörterbuch Latein 1. Edition
1068  +-- 2012 (Shorter: Pons) and Der kleine Stowasser 3. Edition 1991 (shorter:
1069  +-- Stowasser). The Grammar book is Bayer-Lindauer: Lateinische Schulgrammatik
1070  +-- 2. Edition 1994.
1071  +
1072   concrete LexiconLat of Lexicon = CatLat ** open
1073     ParadigmsLat,
1074  ---  IrregLat,
1075  +  IrregLat,
1076  +  ResLat,
1077  +  StructuralLat,
1078  +  NounLat,
1079  +  AdjectiveLat,
1080  +  VerbLat,
1081     Prelude in {
1082
1083   flags
1084     optimize=values ;
1085  -
1086  +  coding = utf8;
1087   lin
1088  ---  airplane_N = mkN "airplane" ;
1089  ---  answer_V2S = mkV2S (regV "answer") toP ;
1090  ---  apartment_N = mkN "apartment" ;
1091  ---  apple_N = mkN "apple" ;
1092  -  art_N = mkN "ars" ;
1093  ---  ask_V2Q = mkV2Q (regV "ask") noPrep ;
1094  -  baby_N = mkN "infans" ;
1095  -  bad_A = mkA "malus" ;
1096  ---  bank_N = mkN "bank" ;
1097  -  beautiful_A = mkA "pulcher" ;
1098  ---  become_VA = mkVA (irregV "become" "became" "become") ;
1099  -  beer_N = mkN "cerevisia" ;
1100  ---  beg_V2V = mkV2V (regDuplV "beg") noPrep toP ;
1101  -  big_A = mkA "magnus" ;
1102  ---  bike_N = mkN "bike" ;
1103  -  bird_N = mkN "avis" "avis" masculine ;
1104  -  black_A = mkA "niger" ;
1105  ---  blue_A = regADeg "blue" ;
1106  ---  boat_N = mkN "boat" ;
1107  -  book_N = mkN "liber" ;
1108  ---  boot_N = mkN "boot" ;
1109  ---  boss_N = mkN human (mkN "boss") ;
1110  -  boy_N = mkN "liber" ;
1111  -  bread_N = mkN "panis" "panis" masculine ;
1112  -  break_V2 = mkV2 (mkV "rumpo" "rupi" "ruptum" "rumpere") ;
1113  ---  broad_A = regADeg "broad" ;
1114  ---  brother_N2 = mkN2 (mkN masculine (mkN "brother")) (mkPrep "of") ;
1115  ---  brown_A = regADeg "brown" ;
1116  ---  butter_N = mkN "butter" ;
1117  ---  buy_V2 = dirV2 (irregV "buy" "bought" "bought") ;
1118  ---  camera_N = mkN "camera" ;
1119  ---  cap_N = mkN "cap" ;
1120  ---  car_N = mkN "car" ;
1121  ---  carpet_N = mkN "carpet" ;
1122  -  cat_N = mkN "felis" ;
```

```
1123 ----  ceiling_N = mkN "ceiling" ;
1124 ----  chair_N = mkN "chair" ;
1125 ----  cheese_N = mkN "cheese" ;
1126 ----  child_N = mk2N "child" "children" ;
1127 ----  church_N = mkN "church" ;
1128 -  city_N = mkN "urbs" "urbis" feminine ;
1129 ----  clean_A = regADeg "clean" ;
1130 ----  clever_A = regADeg "clever" ;
1131 ----  close_V2 = dirV2 (regV "close") ;
1132 ----  coat_N = mkN "coat" ;
1133 ----  cold_A = regADeg "cold" ;
1134 ----  come_V = (irregV "come" "came" "come") ;
1135 ----  computer_N = mkN "computer" ;
1136 ----  country_N = mkN "country" ;
1137 ----  cousin_N = mkN human (mkN "cousin") ;
1138 ----  cow_N = mkN "cow" ;
1139 ----  die_V = (regV "die") ;
1140 ----  dirty_A = regADeg "dirty" ;
1141 ----  distance_N3 = mkN3 (mkN "distance") fromP toP ;
1142 ----  doctor_N = mkN human (mkN "doctor") ;
1143 ----  dog_N = mkN "dog" ;
1144 ----  door_N = mkN "door" ;
1145 ----  drink_V2 = dirV2 (irregV "drink" "drank" "drunk") ;
1146 ----  easy_A2V = mkA2V (regA "easy") forP ;
1147 ----  eat_V2 = dirV2 (irregV "eat" "ate" "eaten") ;
1148 ----  empty_A = regADeg "empty" ;
1149 ----  enemy_N = mkN "enemy" ;
1150 ----  factory_N = mkN "factory" ;
1151 ----  father_N2 = mkN2 (mkN masculine (mkN "father")) (mkPrep "of") ;
1152 ----  fear_VS = mkVS (regV "fear") ;
1153 ----  find_V2 = dirV2 (irregV "find" "found" "found") ;
1154 ----  fish_N = mk2N "fish" "fish" ;
1155 ----  floor_N = mkN "floor" ;
1156 ----  forget_V2 = dirV2 (irregDuplV "forget" "forgot" "forgotten") ;
1157 ----  fridge_N = mkN "fridge" ;
1158 ----  friend_N = mkN human (mkN "friend") ;
1159 ----  fruit_N = mkN "fruit" ;
1160 ----  fun_AV = mkAV (regA "fun") ;
1161 ----  garden_N = mkN "garden" ;
1162 ----  girl_N = mkN feminine (mkN "girl") ;
1163 ----  glove_N = mkN "glove" ;
1164 -  gold_N = mkN "aurum" ;
1165 -  good_A = mkA "bonus" ;
1166 ----  go_V = mk5V "go" "goes" "went" "gone" "going" ;
1167 ----  green_A = regADeg "green" ;
1168 ----  harbour_N = mkN "harbour" ;
1169 ----  hate_V2 = dirV2 (regV "hate") ;
1170 ----  hat_N = mkN "hat" ;
1171 ----  have_V2 = dirV2 (mk5V "have" "has" "had" "had" "having") ;
1172 ----  hear_V2 = dirV2 (irregV "hear" "heard" "heard") ;
1173 ----  hill_N = mkN "hill" ;
1174 ----  hope_VS = mkVS (regV "hope") ;
1175 ----  horse_N = mkN "horse" ;
1176 ----  hot_A = duplADeg "hot" ;
1177 ----  house_N = mkN "house" ;
1178 ----  important_A = compoundADeg (regA "important") ;
1179 ----  industry_N = mkN "industry" ;
1180 ----  iron_N = mkN "iron" ;
1181 ----  king_N = mkN masculine (mkN "king") ;
1182 ----  know_V2 = dirV2 (irregV "know" "knew" "known") ;
1183 ----  lake_N = mkN "lake" ;
1184 ----  lamp_N = mkN "lamp" ;
1185 ----  learn_V2 = dirV2 (regV "learn") ;
1186 ----  leather_N = mkN "leather" ;
1187 ----  leave_V2 = dirV2 (irregV "leave" "left" "left") ;
1188 ----  like_V2 = dirV2 (regV "like") ;
1189 ----  listen_V2 = prepV2 (regV "listen") toP ;
1190 ----  live_V = (regV "live") ;
1191 ----  long_A = regADeg "long" ;
1192 ----  lose_V2 = dirV2 (irregV "lose" "lost" "lost") ;
1193 -  love_N = mkN "amor" ;
1194 -  love_V2 = mkV2 "amare" ;
1195 ----  man_N = mkN masculine (mk2N "man" "men") ;
1196 ----  married_A2 = mkA2 (regA "married") toP ;
1197 ----  meat_N = mkN "meat" ;
```

```
1198 ----   milk_N = mkN "milk" ;
1199 ----   moon_N = mkN "moon" ;
1200 ----   mother_N2 = mkN2 (mkN feminine (mkN "mother")) (mkPrep "of") ;
1201 ----   mountain_N = mkN "mountain" ;
1202 ----   music_N = mkN "music" ;
1203 ----   narrow_A = regADeg "narrow" ;
1204 ----   new_A = regADeg "new" ;
1205 ----   newspaper_N = mkN "newspaper" ;
1206 ----   oil_N = mkN "oil" ;
1207 ----   old_A = regADeg "old" ;
1208 ----   open_V2 = dirV2 (regV "open") ;
1209 ----   paint_V2A = mkV2A (regV "paint") noPrep ;
1210 ----   paper_N = mkN "paper" ;
1211 ----   paris_PN = mkPN (mkN nonhuman (mkN "Paris")) ;
1212 ----   peace_N = mkN "peace" ;
1213 ----   pen_N = mkN "pen" ;
1214 ----   planet_N = mkN "planet" ;
1215 ----   plastic_N = mkN "plastic" ;
1216 ----   play_V2 = dirV2 (regV "play") ;
1217 ----   policeman_N = mkN masculine (mkN "policeman" "policemen") ;
1218 ----   priest_N = mkN human (mkN "priest") ;
1219 ----   probable_AS = mkAS (regA "probable") ;
1220 ----   queen_N = mkN feminine (mkN "queen") ;
1221 ----   radio_N = mkN "radio" ;
1222 ----   rain_V0 = mkV0 (regV "rain") ;
1223 ----   read_V2 = dirV2 (irregV "read" "read" "read") ;
1224 ----   red_A = duplADeg "red" ;
1225 ----   religion_N = mkN "religion" ;
1226 ----   restaurant_N = mkN "restaurant" ;
1227 ----   river_N = mkN "river" ;
1228 ----   rock_N = mkN "rock" ;
1229 ----   roof_N = mkN "roof" ;
1230 ----   rubber_N = mkN "rubber" ;
1231 ----   run_V = (irregDuplV "run" "ran" "run") ;
1232 ----   say_VS = mkVS (irregV "say" "said" "said") ;
1233 ----   school_N = mkN "school" ;
1234 ----   science_N = mkN "science" ;
1235 ----   sea_N = mkN "sea" ;
1236 ----   seek_V2 = dirV2 (irregV "seek" "sought" "sought") ;
1237 ----   see_V2 = dirV2 (irregV "see" "saw" "seen") ;
1238 ----   sell_V3 = dirV3 (irregV "sell" "sold" "sold") toP ;
1239 ----   send_V3 = dirV3 (irregV "send" "sent" "sent") toP ;
1240 ----   sheep_N = mk2N "sheep" "sheep" ;
1241 ----   ship_N = mkN "ship" ;
1242 ----   shirt_N = mkN "shirt" ;
1243 ----   shoe_N = mkN "shoe" ;
1244 ----   shop_N = mkN "shop" ;
1245 ----   short_A = regADeg "short" ;
1246 ----   silver_N = mkN "silver" ;
1247 ----   sister_N = mkN2 (mkN feminine (mkN "sister")) (mkPrep "of") ;
1248 -   sleep_V = mkV "dormio" "dormivi" "dormitus" "dormire" ;
1249 ----   small_A = regADeg "small" ;
1250 ----   snake_N = mkN "snake" ;
1251 ----   sock_N = mkN "sock" ;
1252 ----   speak_V2 = dirV2 (irregV "speak" "spoke" "spoken") ;
1253 ----   star_N = mkN "star" ;
1254 ----   steel_N = mkN "steel" ;
1255 ----   stone_N = mkN "stone" ;
1256 ----   stove_N = mkN "stove" ;
1257 ----   student_N = mkN human (mkN "student") ;
1258 ----   stupid_A = regADeg "stupid" ;
1259 ----   sun_N = mkN "sun" ;
1260 ----   switch8off_V2 = dirV2 (partV (regV "switch") "off") ;
1261 ----   switch8on_V2 = dirV2 (partV (regV "switch") "on") ;
1262 ----   table_N = mkN "table" ;
1263 ----   talk_V3 = mkV3 (regV "talk") toP aboutP ;
1264 ----   teacher_N = mkN human (mkN "teacher") ;
1265 ----   teach_V2 = dirV2 (irregV "teach" "taught" "taught") ;
1266 ----   television_N = mkN "television" ;
1267 ----   thick_A = regADeg "thick" ;
1268 ----   thin_A = duplADeg "thin" ;
1269 ----   train_N = mkN "train" ;
1270 ----   travel_V = (regDuplV "travel") ;
1271 ----   tree_N = mkN "tree" ;
1272 ---- ------ trousers_N = mkN "trousers" ;
```

```
1273 ——   ugly_A = regADeg "ugly" ;
1274 ——   understand_V2 = dirV2 (irregV "understand" "understood" "understood") ;
1275 ——   university_N = mkN "university" ;
1276 ——   village_N = mkN "village" ;
1277 ——   wait_V2 = prepV2 (regV "wait") forP ;
1278 ——   walk_V = (regV "walk") ;
1279 ——   warm_A = regADeg "warm" ;
1280 ——   war_N = mkN "war" ;
1281 ——   watch_V2 = dirV2 (regV "watch") ;
1282 ——   water_N = mkN "water" ;
1283 ——   white_A = regADeg "white" ;
1284 ——   window_N = mkN "window" ;
1285 ——   wine_N = mkN "wine" ;
1286 ——   win_V2 = dirV2 (irregDuplV "win" "won" "won") ;
1287 ——   woman_N = mkN feminine (mk2N "woman" "women") ;
1288 ——   wonder_VQ = mkVQ (regV "wonder") ;
1289 ——   wood_N = mkN "wood" ;
1290 ——   write_V2 = dirV2 (irregV "write" "wrote" "written") ;
1291 ——   yellow_A = regADeg "yellow" ;
1292 ——   young_A = regADeg "young" ;
1293 ——
1294 ——   do_V2 = dirV2 (mk5V "do" "does" "did" "done" "doing") ;
1295 ——   now_Adv = mkAdv "now" ;
1296 ——   already_Adv = mkAdv "already" ;
1297 ——   song_N = mkN "song" ;
1298 ——   add_V3 = dirV3 (regV "add") toP ;
1299 ——   number_N = mkN "number" ;
1300 ——   put_V2 = prepV2 (irregDuplV "put" "put" "put") noPrep ;
1301 ——   stop_V = regDuplV "stop" ;
1302 ——   jump_V = regV "jump" ;
1303 ——
1304 ——   left_Ord = ss "left" ;
1305 ——   right_Ord = ss "right" ;
1306 ——   far_Adv = mkAdv "far" ;
1307 ——   correct_A = (regA "correct") ;
1308 ——   dry_A = regA "dry" ;
1309 ——   dull_A = regA "dull" ;
1310 ——   full_A = regA "full" ;
1311 ——   heavy_A = regA "heavy" ;
1312 ——   near_A = regA "near" ;
1313 ——   rotten_A = (regA "rotten") ;
1314 ——   round_A = regA "round" ;
1315 ——   sharp_A = regA "sharp" ;
1316 ——   smooth_A = regA "smooth" ;
1317 ——   straight_A = regA "straight" ;
1318 ——   wet_A = regA "wet" ; ———
1319 ——   wide_A = regA "wide" ;
1320 ——   animal_N = mkN "animal" ;
1321 ——   ashes_N = mkN "ash" ; -- FIXME: plural only?
1322 ——   back_N = mkN "back" ;
1323 ——   bark_N = mkN "bark" ;
1324 ——   belly_N = mkN "belly" ;
1325 ——   blood_N = mkN "blood" ;
1326 ——   bone_N = mkN "bone" ;
1327 ——   breast_N = mkN "breast" ;
1328 ——   cloud_N = mkN "cloud" ;
1329 ——   day_N = mkN "day" ;
1330 ——   dust_N = mkN "dust" ;
1331 ——   ear_N = mkN "ear" ;
1332 ——   earth_N = mkN "earth" ;
1333 ——   egg_N = mkN "egg" ;
1334 ——   eye_N = mkN "eye" ;
1335 ——   fat_N = mkN "fat" ;
1336 ——   feather_N = mkN "feather" ;
1337 ——   fingernail_N = mkN "fingernail" ;
1338 ——   fire_N = mkN "fire" ;
1339 ——   flower_N = mkN "flower" ;
1340 ——   fog_N = mkN "fog" ;
1341 ——   foot_N = mk2N "foot" "feet" ;
1342 ——   forest_N = mkN "forest" ;
1343 ——   grass_N = mkN "grass" ;
1344 ——   guts_N = mkN "gut" ; -- FIXME: no singular
1345 ——   hair_N = mkN "hair" ;
1346 ——   hand_N = mkN "hand" ;
1347 ——   head_N = mkN "head" ;
```

```
1348 ----  heart_N = mkN "heart" ;
1349 ----  horn_N = mkN "horn" ;
1350 ----  husband_N = mkN masculine (mkN "husband") ;
1351 ----  ice_N = mkN "ice" ;
1352 ----  knee_N = mkN "knee" ;
1353 ----  leaf_N = mk2N "leaf" "leaves" ;
1354 ----  leg_N = mkN "leg" ;
1355 ----  liver_N = mkN "liver" ;
1356 ----  louse_N = mk2N "louse" "lice" ;
1357 ----  mouth_N = mkN "mouth" ;
1358 ----  name_N = mkN "name" ;
1359 ----  neck_N = mkN "neck" ;
1360 ----  night_N = mkN "night" ;
1361 ----  nose_N = mkN "nose" ;
1362 ----  person_N = mkN human (mkN "person") ;
1363 ----  rain_N = mkN "rain" ;
1364 ----  road_N = mkN "road" ;
1365 ----  root_N = mkN "root" ;
1366 ----  rope_N = mkN "rope" ;
1367 ----  salt_N = mkN "salt" ;
1368 ----  sand_N = mkN "sand" ;
1369 ----  seed_N = mkN "seed" ;
1370 ----  skin_N = mkN "skin" ;
1371 ----  sky_N = mkN "sky" ;
1372 ----  smoke_N = mkN "smoke" ;
1373 ----  snow_N = mkN "snow" ;
1374 ----  stick_N = mkN "stick" ;
1375 ----  tail_N = mkN "tail" ;
1376 ----  tongue_N = mkN "tongue" ;
1377 ----  tooth_N = mk2N "tooth" "teeth" ;
1378 ----  wife_N = mkN feminine (mk2N "wife" "wives") ;
1379 ----  wind_N = mkN "wind" ;
1380 ----  wing_N = mkN "wing" ;
1381 ----  worm_N = mkN "worm" ;
1382 ----  year_N = mkN "year" ;
1383 ----  blow_V = IrregLat.blow_V ;
1384 ----  breathe_V = dirV2 (regV "breathe") ;
1385 ----  burn_V = IrregLat.burn_V ;
1386 ----  dig_V = IrregLat.dig_V ;
1387 ----  fall_V = IrregLat.fall_V ;
1388 ----  float_V = regV "float" ;
1389 ----  flow_V = regV "flow" ;
1390 ----  fly_V = IrregLat.fly_V ;
1391 ----  freeze_V = IrregLat.freeze_V ;
1392 ----  give_V3 = dirV3 give_V toP ;
1393 ----  laugh_V = regV "laugh" ;
1394 ----  lie_V = IrregLat.lie_V ;
1395 ----  play_V = regV "play" ;
1396 ----  sew_V = IrregLat.sew_V ;
1397 ----  sing_V = IrregLat.sing_V ;
1398 ----  sit_V = IrregLat.sit_V ;
1399 ----  smell_V = regV "smell" ;
1400 ----  spit_V = IrregLat.spit_V ;
1401 ----  stand_V = IrregLat.stand_V ;
1402 ----  swell_V = IrregLat.swell_V ;
1403 ----  swim_V = IrregLat.swim_V ;
1404 ----  think_V = IrregLat.think_V ;
1405 ----  turn_V = regV "turn" ;
1406 ----  vomit_V = regV "vomit" ;
1407 ----
1408 ----  bite_V2 = dirV2 IrregLat.bite_V ;
1409 ----  count_V2 = dirV2 (regV "count") ;
1410 ----  cut_V2 = dirV2 IrregLat.cut_V ;
1411 ----  fear_V2 = dirV2 (regV "fear") ;
1412 ----  fight_V2 = dirV2 fight_V ;
1413 ----  hit_V2 = dirV2 hit_V ;
1414 ----  hold_V2 = dirV2 hold_V ;
1415 ----  hunt_V2 = dirV2 (regV "hunt") ;
1416 ----  kill_V2 = dirV2 (regV "kill") ;
1417 ----  pull_V2 = dirV2 (regV "pull") ;
1418 ----  push_V2 = dirV2 (regV "push") ;
1419 ----  rub_V2 = dirV2 (regDuplV "rub") ;
1420 ----  scratch_V2 = dirV2 (regV "scratch") ;
1421 ----  split_V2 = dirV2 split_V ;
1422 ----  squeeze_V2 = dirV2 (regV "squeeze") ;
```

```
1423 ---  stab_V2 = dirV2 (regDuplV "stab") ;
1424 ---  suck_V2 = dirV2 (regV "suck") ;
1425 ---  throw_V2 = dirV2 throw_V ;
1426 ---  tie_V2 = dirV2 (regV "tie") ;
1427 ---  wash_V2 = dirV2 (regV "wash") ;
1428 ---  wipe_V2 = dirV2 (regV "wipe") ;
1429 ---
1430 -----  other_A = regA "other" ;
1431 ---
1432 ---  grammar_N = mkN "grammar" ;
1433 ---  language_N = mkN "language" ;
1434 ---  rule_N = mkN "rule" ;
1435 ---
1436 ----- added 4/6/2007
1437 ---    john_PN = mkPN (mkN masculine (mkN "John")) ;
1438 ---    question_N = mkN "question" ;
1439 ---    ready_A = regA "ready" ;
1440 ---    reason_N = mkN "reason" ;
1441 ---    today_Adv = mkAdv "today" ;
1442 ---    uncertain_A = regA "uncertain" ;
1443 ---
1444 ---oper
1445 ---  aboutP = mkPrep "about" ;
1446 ---  atP = mkPrep "at" ;
1447 ---  forP = mkPrep "for" ;
1448 ---  fromP = mkPrep "from" ;
1449 ---  inP = mkPrep "in" ;
1450 ---  onP = mkPrep "on" ;
1451 ---  toP = mkPrep "to" ;
1452 ---
1453 +  airplane_N = mkN "aeroplanum" ; -- -i n. (http://la.wikipedia.org/wiki/A%C3%ABroplanum / Pons)
1454 +  answer_V2S = mkV2S ( mkV "respondere" ) Dat_Prep ; -- -spondeo, -spondi, -sponsum 2 (Langenscheidts) alicui; ad, contra, adversus aliquid (
1455 +  apartment_N = mkN "domicilium" ; -- -i n. (Langenscheidts)
1456 +  apple_N = mkN "malum" ; -- -i n. (Langenscheidts)
1457 +  art_N = mkN "ars" "artis" feminine ; -- Ranta; artis f. (Langenscheidts)
1458 +  ask_V2Q = mkV2Q ( mkV "rogare" ) Acc_Prep ; -- rogo 1 (Langenscheidts) aliquem aliquid (Stowasser)
1459 +  baby_N = mkN "infans" "infantis" ( variants { feminine ; masculine } );  -- Ranta; -antis m./f. (Langenscheidts)
1460 +  bad_A = mkA "malus" ; -- Ranta; peior, pessimus 3 (Langenscheidts)
1461 +  bank_N = mkN "argentaria" ; -- -ae f. (http://la.wikipedia.org/wiki/Argentaria / Pons)
1462 +  beautiful_A = mkA "pulcher" ; -- -chra, -chrum (Langenscheidts)
1463 +  become_VA = mkVA (mkV "fieri")  ; -- fio, factus (Langenscheidts)
1464 +  beer_N = mkN ( "cervisia" ) ; -- Ranta; -ae f. (http://la.wikipedia.org/wiki/Cervisia / Pons)
1465 +  beg_V2V = mkV2V ( mkV "petere" "peto" "petivi" "petitum" ) "ab" False ; -- peto, -tivi/tii, -titum 3 (Langenscheidts) ab aliquo (Stowasser)
1466 +  big_A = mkA "magnus" ; -- Ranta; maior, maximus 3 (Langenscheidts)
1467 +  bike_N = mkN "birota" ; -- -ae f. (http://la.wikipedia.org/wiki/Birota / Pons)
1468 +  bird_N = mkN "avis" "avis" feminine ; -- Ranta; -is f. (Langenscheidts)
1469 +  black_A = mkA "niger" ;  -- Ranta; -gra, -grum (Langenscheidts)
1470 +  blue_A = mkA ( variants { "caeruleus" ; "caerulus" } ) ; -- 3 (Langenscheidts)
1471 +  boat_N = mkN "navicula" ; -- -ae f. (Langenscheidts)
1472 +  book_N = mkN "liber"; -- Ranta; -bri m. (Langenscheidts)
1473 +  boot_N = mkN "calceus" ; -- -i m. (Langenscheidts)
1474 +  boss_N = mkN "dux" "ducis" ( variants { feminine ; masculine } ) ; -- ducis m./f. (Langenscheidts)
1475 +  boy_N = mkN "puer" "pueri" masculine ; -- -eri m. (Langenscheidts)
1476 +  bread_N = variants { (mkN "panis" "panis" masculine ) ; (mkN "pane" "panis" neuter ) } ; -- -is m./n. (Langenscheidts)
1477 +  break_V2 = mkV2 ( mkV "rumpere" "rumpo" "rupi" "ruptum" ) ; -- Ranta; 3 (Langenscheidts) aliquem (Bayer–Lindauer 110)
1478 +  broad_A = mkA "latus" ; -- 3 (Langenscheidts)
1479 +  brother_N2 = mkN2 ( mkN "frater" "fratris" masculine ) Gen_Prep; -- -tris m. (Langenscheidts) alicuius (Bayer–Lindauer 125.2)
1480 +  brown_A = mkA "fulvus" ; -- 3 (Langenscheidts)
1481 +  butter_N = mkN "butyrum" ; -- -i n. (http://la.wikipedia.org/wiki/Butyrum / Pons)
1482 +  buy_V2 = mkV2 (mkV "emere") ; -- emo, emi, emptum 3 (Langenscheidts) (Stowasser) ab, de aliquo (Stowasser)
1483 +  -- Trying to work with Machina ++ photographica
1484 +  camera_N = ResLat.useCNasN (AdjCN (PositA (mkA "photographicus") ) (UseN (mkN "machina" ) ) )  ; -- (http://la.wikipedia.org/wiki/Machina_
1485 +  cap_N = mkN "galerus" ; -- -i m. (Langenscheidts)
1486 +  car_N = mkN "autoreada" ; -- -ae f. (Pons / http://la.wikipedia.org/wiki/Autocinetum)
1487 +  carpet_N = mkN "stragulum" ; -- -i n. (Pons / http://la.wikipedia.org/wiki/Teges_pavimenti)
1488 +  cat_N = mkN ( variants {"feles" ; "felis" } ) "felis" feminine ; -- -is f. (Langenscheidts)
1489 +  ceiling_N = mkN "tegimentum" ; -- -i n. (Langenscheidts)
1490 +  chair_N = mkN "sedes" "sedis" feminine; -- -is f. (Langenscheidts)
1491 +  cheese_N = mkN "caseus" ; -- -i m. (Langenscheidts)
1492 +  child_N = mkN "proles" "prolis" feminine ; -- -is f. (Langenscheidts)
1493 +  church_N = mkN "ecclesia" ; -- -ae f. (Langenscheidts)
1494 +  city_N = mkN "urbs" "urbis" feminine; -- Ranta; urbis f. (Langenscheidts)
1495 +  clean_A = mkA "lautus" ; -- 3 (Langenscheidts)
1496 +  clever_A = mkA "callidus" ; -- 3 (Langenscheidts)
1497 +  close_V2 = mkV2 (mkV "claudere") ; -- claudo, clasi, clausum 3 (Langenscheidts) aliquem (Bayer–Lindauer 110)
```

```
1498 +  coat_N = mkN "pallium" ; --- -i n. (Langenscheidts)
1499 +  cold_A = mkA "frigidus" ; --- 3 (Langenscheidts)
1500 +  come_V = mkV "venire" ; --- veno, veni, ventum 4 (Langenscheidts)
1501 +  computer_N = mkN "computatrum" ; --- -i n. (http://la.wikipedia.org/wiki/Computatrum / Pons)
1502 +  country_N = mkN "terra" ; --- -ae f. (Langenscheidts)
1503 +  cousin_N = mkN ( variants {"consobrinus" ; "consobrina" } ) ; --- -i/-ae m./f. (Langenscheidts)
1504 +  cow_N = mkN "bos" "bovis" ( variants { feminine ; masculine } ) ; --- bovis (gen. pl. boum, dat./abl. pl. bobus/bubus) m./f. (Langenscheidts
1505 +  die_V = mkV "mori" "mortuus" "morturus"; --- morior, mortuus sum, morturus (Langenscheidts)
1506 +  dirty_A = mkA "sordidus" ; --- 3 (Langenscheidts)
1507 +
1508 +  distance_N3 = mkN3 (mkN "distantia") from_Prep to_Prep ; --- -ae f. (Langenscheidts) ab, ad aliquem; alicuius; aliquem (???)
1509 +  doctor_N = mkN "medicus" ; --- -i m. (Langenscheidts)
1510 +  dog_N = mkN "canis" "canis" ( variants { masculine ; feminine } ) ; --- -is m./f. (Langenscheidts)
1511 +  door_N = mkN "porta" ; --- -ae f. (Langenscheidts)
1512 +  drink_V2 = mkV2 (mkV "bibere") ; --- bibo, potum 3 (Langenscheidts) aliquem (Bayer-Lindauer 110)
1513 +  easy_A2V = mkA2V (mkA "facilis" "facile" ) for_Prep ; --- -e sup -illimus (Langenscheidts)
1514 +  eat_V2 = mkV2 (mkV "cenare") ; --- ceno 1 (Langenscheidts) aliquem (Bayer-Lindauer 110)
1515 +  empty_A = mkA "vacuus" ; --- 3 (Langenscheidts)
1516 +  enemy_N = mkN "hostis" "hostis" ( variants { masculine ; feminine } ) ; --- -is m./f. (Langenscheidts)
1517 +  factory_N = mkN "officina" ; --- -ae f. (Langenscheidts)
1518 +  father_N2 = mkN2 (mkN "pater" "patris" masculine ) Gen_Prep ; --- -tris m. gen pl -um (Langenscheidts) alicuius (Bayer-Lindauer 125.2)
1519 +  fear_VS = mkVS (mkV "timere") ; --- timeo, timui, ---- 2 (Langenscheidts)
1520 +  find_V2 = mkV2 (mkV "reperire") ; --- reperio, repperi, repertum 4 (Langenscheidts) aliquem
1521 +  fish_N = mkN "piscis" "piscis" masculine ; --- -is m. (Langenscheidts)
1522 +  floor_N = mkN "pavimentum" ; --- -i n. (Langenscheidts)
1523 +  forget_V2 = mkV2 (mkV "oblivisci" "obliviscor" "oblitus" ) ; --- obliscor, oblitus sum 3 (Langenscheidts)
1524 +  fridge_N = mkN "frigidarium" ; --- --i n. (Pons / http://la.wikipedia.org/wiki/Armarium_frigidarium)
1525 +  friend_N = mkN ( variants { "amicus" ; "amica" } ) ; --- -i/-ae m./f. (Langenscheidts)
1526 +  fruit_N = mkN "fructus" "fructus" masculine; --- -us m. (Langenscheidts)
1527 +  fun_AV = mkAV (mkA "iocosus") ; --- 3 (Langenscheidts)
1528 +  garden_N = mkN "hortus" ; --- -i m. (Langenscheidts)
1529 +  girl_N = mkN "puella" ; --- -ae f. (Langenscheidts)
1530 +  glove_N = mkN "caestus" "caestus" masculine ; ---us m. (Langenscheidts)
1531 +  gold_N = mkN "aurum" ; --- Ranta; -i n. (Langenscheidts)
1532 +  good_A = mkA "bonus" ; --- Ranta; 3 comp melior, -us; sup optimus 3; adv bene
1533 +  go_V = IrregLat.go_V ; --- eo, i(v)i, itum (Langenscheidts)
1534 +  green_A = mkA "viridis" "viride"; --- -e (Langenscheidts)
1535 +  harbour_N = mkN "portus" "portus" masculine ; --- -us m. (Langenscheidts)
1536 +  hate_V2 = mkV2 IrregLat.hate_V Acc_Prep ; --- odi, osurus/odivi (Langenscheidts)
1537 +  hat_N = mkN "petasus" ; --- -i m. (Langenscheidts)
1538 +  hear_V2 = mkV2 (mkV "audire") ; --- 4 (Langenscheidts)
1539 +  hill_N = mkN "collis" "collis" masculine ; --- -is m. (Langenscheidts)
1540 +  hope_VS = mkVS (mkV "sperare") ; --- 1 (Langenscheidts)
1541 +  horse_N = mkN "equus" ; --- -i m. (Langenscheidts)
1542 +  hot_A = mkA "calidus" ; --- 3 (Langenscheidts)
1543 +  house_N = mkN "domus" "domus" feminine ; --- -us f. (Langenscheidts)
1544 +  important_A = mkA "gravis" "grave"; --- -e (Langenscheidts)
1545 +  industry_N = mkN "industria" ; --- -ae f. (http://la.wikipedia.org/wiki/Industria / Pons)
1546 +  iron_N = mkN "ferrum" ; --- -i m. (Langenscheidts)
1547 +  king_N = mkN "rex" "regis" masculine; --- regis m. (Langenscheidts)
1548 +  know_V2 = mkV2 (mkV "scire") ; --- scio, scivi/scii, scitum 4 (Langenscheidts)
1549 +  know_VQ = mkV "scire" ;
1550 +  know_VS = mkV "scire" ;
1551 +  lake_N = mkN "lacus" "lacus" masculine; --- -us m. (Langenscheidts)
1552 +  lamp_N = mkN "lucerna" ; --- -ae f. (Langenscheidts)
1553 +  learn_V2 = mkV2 (mkV "discere" "disco" "didici") ; --- disco, didici, - 3 (-isc-?) (Langenscheidts)
1554 +  leather_N = mkN "scortum" ; --- -i n. (Langenscheidts)
1555 +  leave_V2 = mkV2 (mkV "relinquere" "relinquo" "relinqui" "relictum") ; --- relinquo, relinqui, relictum 3 (Langenscheidts)
1556 +  like_V2 = mkV2 ( IrregLat.want_V ) ; --- vello, velli (volsi, vulsi), vulsum 3 (Langenscheidts)
1557 +  listen_V2 = mkV2 (mkV "auscultare") ; --- ausculto 1 (Langenscheidts)
1558 +  live_V = mkV "vivere" "vivo" "vixi" "victum" ; --- vivo, vixi, victurus 3 (Langenscheidts)
1559 +  long_A = mkA "longus" ; --- 3 (Langenscheidts)
1560 +  lose_V2 = mkV2 (mkV "amittere") ; --- amitto, amissi, amissum 3 (Langenscheidts)
1561 +  love_N = mkN "amor" "amoris" masculine ; --- Ranta; -oris m. (Langenscheidts)
1562 +  love_V2 = mkV2 "amare" ; --- Ranta; amo 1 (Langenscheidts)
1563 +  man_N = mkN "vir" "viri" masculine ; --- viri m. (Langenscheidts)
1564 +  --- Category not yet implemented
1565 +  married_A2 = mkA2 (mkA "coniunctus") to_Prep; --- 3 (http://www.perseus.tufts.edu/hopper/text?doc=Perseus:text:1999.04.0060:entry=coniunctus
1566 +  meat_N = mkN "carnis" "carnis" feminine ; --- -is f. (Langenscheidts)
1567 +  milk_N = mkN "lac" "lactis" neuter ; --- --- lactis n. (Langenscheidts)
1568 +  moon_N = mkN "luna" ; --- -ae f. (Langenscheidts)
1569 +  mother_N2 = mkN2 ( mkN "mater" "matris" feminine ) Gen_Prep ; --- matris f. (Langenscheidts)
1570 +  mountain_N = mkN "mons" "montis" masculine ; --- montis m. (Langenscheidts)
1571 +  music_N = mkN "musica" ; --- -ae f. L..
1572 +  narrow_A = mkA "angustus" ; --- 3 (Langenscheidts)
```

```
1573 +  new_A = mkA "novus" ; -- 3 (Langenscheidts)
1574 +  newspaper_N = mkN "diurnum" ; -- -i n. (Pons)
1575 +  oil_N = mkN "oleum" ; -- -i n. (Langenscheidts)
1576 +  old_A = mkA "vetus" "veteris"; -- (Langenscheidts)
1577 +  open_V2 = mkV2 (mkV "aperire") ; -- aperio, aperui, apertum 4 (Langenscheidts)
1578 +  paint_V2A = mkV2A (mkV "pingere" "pingo" "pinxi" "pictum" ) Acc_Prep ; -- pingo, pinxi, pictum 3 (Langenscheidts)
1579 +  paper_N = mkN "charta" ; -- -ae f. (http://la.wikipedia.org/wiki/Charta / Pons)
1580 +  paris_PN = mkPN (mkN "Lutetia") ; -- -ae f. (http://la.wikipedia.org/wiki/Lutetia)
1581 +  peace_N = mkN "pax" "pacis" feminine ; -- pacis f. (Langenscheidts)
1582 +  pen_N = mkN "stilus" ; -- -i m. (Langenscheidts)
1583 +  planet_N = mkN "planeta" ; -- -ae m. (http://la.wikipedia.org/wiki/Planeta)
1584 +  plastic_N = mkN "plastica" "plasticae" feminine ; -- -ae f. (http://la.wikipedia.org/wiki/Plasticum)
1585 +  play_V2 = mkV2 (mkV "ludere" "ludo" "lusi" "lusum" ) ; -- ludo, lusi, lusum 3 (Langenscheidts)
1586 +  policeman_N = mkN "custos" "custodis" ( variants { masculine ; feminine } ) ; -- -odis m./f. (Langenscheidts)
1587 +  priest_N = mkN "sacerdos" "sacerdotis" ( variants { masculine ; feminine } ) ; -- -dotis m./f. (Langenscheidts)
1588 +  probable_AS = mkAS (mkA "verisimilis" "verisimile") ; -- -e (Langenscheidts)
1589 +  queen_N = mkN "regina" ; -- -ae f. (Langenscheidts)
1590 +  radio_N = mkN "radiophonum" ; -- -i n. (Pons / http://la.wikipedia.org/wiki/Radiophonia)
1591 +  rain_V0 = mkV0 ( IrregLat.rain_V ) ; -- (Langenscheidts)
1592 +  read_V2 = mkV2 (mkV "legere" "lego" "legi" "lectum") ; -- lego, legi, lectum 3 (Langenscheidts)
1593 +  red_A = mkA "ruber" ; -- rubra, rubrum (Langenscheidts)
1594 +  religion_N = mkN "religio" "religionis" feminine ; -- -onis f. (Langenscheidts)
1595 +  restaurant_N = mkN "taberna" ; -- -ae f. (Langenscheidts)
1596 +  river_N = mkN "fluvius" ; -- -i m. (Langenscheidts)
1597 +  rock_N = mkN "saxum" ; -- -i n. (Langenscheidts)
1598 +  roof_N = mkN "tectum" ; -- -i n. (Langenscheidts)
1599 +  rubber_N = mkN "cummis" "cummis" feminine ; -- -is f. Der kleine Stowasser
1600 +  run_V = mkV "currere" "curro" "cucurri" "cursum" ; -- curro, cucurri, cursum 3 (Langenscheidts)
1601 +  say_VS = mkVS (mkV "dicere" "dico" "dixi" "dictum") ; -- dico, dixi, dictum 3 (Langenscheidts)
1602 +  school_N = mkN "schola" ; -- -ae f. (Langenscheidts)
1603 +  -- Irregular
1604 +  science_N = IrregLat.science_N ;
1605 +  sea_N = mkN "mare" "maris" neuter ; -- -is n. (Langenscheidts)
1606 +  seek_V2 = mkV2 (mkV "quaerere" "quaero" "quaesivi" "quaesitum" ) ; -- quaero, quaesivi, quaesitum 3 (Langenscheidts)
1607 +  see_V2 = mkV2 (mkV "videre") ; -- video, vidi, visum 2 (Langenscheidts)
1608 +  sell_V3 = mkV3 ( mkV "vendere" "vendo" "vendidi" "venditum" ) Acc_Prep Dat_Prep ; -- vendo, vendidi, venditum 3 (Langenscheidts)
1609 +  send_V3 = mkV3 ( mkV "mittere" "mitto" "misi" "missum" ) Acc_Prep Dat_Prep ; -- mitto, misi, missum 3 (Langenscheidts)
1610 +  sheep_N = mkN "ovis" "ovis" feminine ; -- -is f. (Langenscheidts)
1611 +  ship_N = mkN "navis" "navis" feminine ; -- -is f. acc. -em (-im) abl meist -i (Langenscheidts)
1612 +  shirt_N = mkN "tunica" ; -- -ae f. (Langenscheidts)
1613 +  shoe_N = boot_N ;
1614 +  shop_N = mkN "institorium" ; -- -i n. (Langenscheidts)
1615 +  short_A = mkA "brevis" "breve" ; -- -e (Langenscheidts)
1616 +  silver_N = mkN "argentum" ; -- -i n. (Langenscheidts)
1617 +  sister_N = mkN "soror" "sororis" feminine; -- -oris f. (Langenscheidts)
1618 +--  sleep_V = mkV "dormio" "dormivi" "dormitus" "dormire" ; -- Ranta;
1619 +  sleep_V = mkV "dormire" ; -- 4 (Langenscheidts)
1620 +  small_A = mkA "parvus" ; -- 3 (Langenscheidts)
1621 +  snake_N = mkN "serpens" "serpentis" ( variants { masculine ; feminine } ) ; -- -entis m./f. (Langenscheidts)
1622 +  sock_N = mkN "impile" "impilis" masculine ; -- -is n. (Pons)
1623 +  speak_V2 = mkV2 (mkV "loqui" "loquor" "locutus" ) ; -- loquor, locutus sum 3 (Langenscheidts)
1624 +  star_N = mkN "stella" ; -- -ae f. (Langenscheidts)
1625 +  steel_N = mkN "chalybs" "chalybis" masculine ; -- chalybis m. (Langenscheidts)
1626 +  stone_N = mkN "lapis" "lapidis" masculine ; -- -idis m. (Langenscheidts)
1627 +  stove_N = mkN "fornax" "formacis" feminine ; -- -acis f. (Langenscheidts)
1628 +  student_N = mkN ( variants { "discipulus"; "discipula" } ) ; -- -i/-ae m./f. (Langenscheidts)
1629 +  stupid_A = mkA "stultus" ; -- 3 (Langenscheidts)
1630 +  sun_N = mkN "sol" "solis" masculine ; -- solis m. (Langenscheidts)
1631 +  switch8off_V2 = mkV2 (mkV "accendere") ; -- -cendo, -cendi, -censum 3 (Langenscheidts)
1632 +  switch8on_V2 = mkV2 ( variants { mkV "exstinguere" "exstingo" "exstinxi" "exstinctum" ; mkV "extinguere" "extingo" "extinxi" "extinctum" }
1633 +  table_N = mkN "mensa" ; -- -ae f. (Langenscheidts)
1634 +  talk_V3 = mkV3 ( lin V speak_V2 ) Dat_Prep Acc_Prep ;
1635 +  teacher_N = mkN "magister" "magistri" masculine ; -- -tri m. (Langenscheidts)
1636 +  teach_V2 = mkV2 (mkV "docere") ; -- doceo, docui, doctum 2 (Langenscheidts)
1637 +  television_N = mkN "televisio" "televisionis" feminine ; -- -onis f. (Pons)
1638 +  thick_A = mkA "crassus" ; -- 3 (Langenscheidts)
1639 +  thin_A = mkA "tenuis" "tenue" ; -- -e (Langenscheidts)
1640 +  train_N = mkN "hamaxostichus" ; -- -i m. (http://la.wikipedia.org/wiki/Hamaxostichus)
1641 +  travel_V = ResLat.useVPasV ( ComplSlash ( SlashV2a ( mkV2 "facere" ) ) ( DetCN ( DetQuant IndefArt NumSg ) ( UseN ( mkN "iter" "itineris" N
1642 +  tree_N = mkN "arbor" "arboris" feminine ; -- -oris f.
1643 +  -- Not even in English implemented
1644 +--  trousers_N = mkN "trousers" ;
1645 +  ugly_A = mkA "foedus" ; -- 3 (Langenscheidts)
1646 +  understand_V2 = mkV2 (mkV "intellegere" "intellego" "intellexi" "intellectum" ) ; -- intellego, intellexi, intellectum 3 (Langenscheidts)
1647 +  university_N = mkN "universitas" "universitatis" feminine ; -- -atis f. (http://la.wikipedia.org/wiki/Universitas) and (Langenscheidts)
```

```
1648 +  village_N = mkN "vicus" ; -- -i m. (Langenscheidts)
1649 +  wait_V2 = mkV2 (mkV "exspectare") ; -- 1 (Langenscheidts)
1650 +  walk_V = mkV "vadere" ; -- 3 (Langenscheidts)
1651 +  warm_A = mkA "calidus" ; -- 3 (Langenscheidts)
1652 +  war_N = mkN "bellum" ; -- -i m. (Langenscheidts)
1653 +  watch_V2 = mkV2 (mkV "spectare") ; -- 1 (Langenscheidts)
1654 +  water_N = mkN "aqua" ; -- -ae f. (Langenscheidts)
1655 +  white_A = mkA "albus" ; -- 3 (Langenscheidts)
1656 +  window_N = mkN "fenestra" ; -- -ae f. (Langenscheidts)
1657 +  wine_N = mkN "vinum" ; -- -i n. (Langenscheidts)
1658 +  win_V2 = mkV2 (mkV "vincere") ; -- vinco, vici, victum 3 (Langenscheidts)
1659 +  woman_N = mkN "femina" ; -- -ae f. (Langenscheidts)
1660 +  wonder_VQ = mkVQ (mkV "mirari") ; -- 1 (Langenscheidts)
1661 +  wood_N = mkN "lignum" ; -- -i n. (Langenscheidts)
1662 +  write_V2 = mkV2 (mkV "scribere") ; -- scribo, scripsi, scriptum 3 (Langenscheidts)
1663 +  yellow_A = mkA "flavus" ; -- 3 (Langenscheidts)
1664 +  young_A = mkA "adulescens" "adulescentis"; -- -entis (Langenscheidts)
1665 +
1666 +  do_V2 = mkV2 (mkV "agere") ; -- ago, egi, actum 3 (Langenscheidts)
1667 +  now_Adv = mkAdv "nunc" ; -- (Langenscheidts)
1668 +  already_Adv = mkAdv "iam" ; -- (Langenscheidts)
1669 +  song_N = mkN "carmen" "carminis" neuter ; -- -inis n. (Langenscheidts)
1670 +  add_V3 = mkV3 ( mkV "addere" "addo" "addidi" "additum" ) Acc_Prep to_P ; -- addo, addidi, additum 3 (Langenscheidts)
1671 +  number_N = mkN "numerus" ; -- -i m.
1672 +  put_V2 = mkV2 (mkV "ponere" "pono" "posui" "positum" ) ; -- pono, posui, positum 3 (Langenscheidts)
1673 +  stop_V = mkV "sistere" "sisto" "steti" "statum" ; -- sisto, stiti/steti, statum 3 (Langenscheidts)
1674 +  jump_V = mkV "saltare" ; -- 1 (Langenscheidts)
1675 +
1676 +  left_Ord = ss "sinister" ; -- -tra, -trum (Langenscheidts)
1677 +  right_Ord = ss "dexter" ; -- -t(e)ra, -t(e)rum (Langenscheidts)
1678 +  far_Adv = mkAdv "longe" ; -- (Langenscheidts)
1679 +  correct_A = mkA "rectus" ; -- 3 (Langenscheidts)
1680 +  dry_A = mkA "aridus" ; -- 3 (Langenscheidts)
1681 +  dull_A = mkA "bardus" ; -- 3 (Langenscheidts) u. (http://www.perseus.tufts.edu/hopper/text?doc=Perseus:text:1999.04.0060:entry=bardus&highl
1682 +  full_A = mkA "plenus" ; -- 3 (Langenscheidts)
1683 +  heavy_A = mkA "gravis" "grave" ; -- -e (Langenscheidts)
1684 +  near_A = mkA "propinquus" ; -- 3 (comp. durch propior, -ius sup. durch proximus 3) (Langenscheidts)
1685 +  rotten_A = mkA "perditus" ; -- 3 (Langenscheidts)
1686 +  round_A = mkA "rotundus" ; -- 3 (Langenscheidts)
1687 +  sharp_A = mkA "acer" "acris" ; -- acris, acre (Langenscheidts)
1688 +  smooth_A = mkA "lubricus" ; -- 3 (Langenscheidts)
1689 +  straight_A = mkA "rectus" ; -- 3 (Langenscheidts)
1690 +  wet_A = mkA "umidus" ; -- 3 (Langenscheidts)
1691 +  wide_A = mkA "vastus" ; -- 3 (Langenscheidts)
1692 +  animal_N = mkN "animal" "animalis" neuter ; -- -alis n. (Langenscheidts)
1693 +  ashes_N = mkN "cinis" "cineris" masculine ; -- -eris m. (Langenscheidts) & Bayer-Lindauer 33 1.2
1694 +  back_N = mkN "tergum" ; -- -i n. (Langenscheidts)
1695 +  bark_N = mkN "cortex" "corticis" ( variants { masculine ; feminine } ) ; -- -icis m./(f.) (Langenscheidts)
1696 +  belly_N = mkN "venter" "ventris" masculine ; -- -tris m. (Langenscheidts)
1697 +  blood_N = variants { mkN "sanguis" "sanguinis" masculine ; mkN "sangis" "sanginis" masculine} ; -- -inis m. (Langenscheidts)
1698 +  bone_N = mkN "os" "ossis" neuter ; -- ossis n. (Langenscheidts)
1699 +  breast_N = mkN "pectus""pectoris" neuter ; -- pectoris n. (Langenscheidts)
1700 +  cloud_N = mkN "nubes" "nubis" feminine ; -- -is f. (Langenscheidts)
1701 +  day_N = mkN "dies" "diei" ( variants { masculine ; feminine } ) ; -- -ei m./f. (Langenscheidts)
1702 +  dust_N = mkN "pulvis" "pulveris" masculine;  -- -veris m. (Langenscheidts)
1703 +  ear_N = mkN "auris" "auris" feminine; -- -is f. (Langenscheidts)
1704 +  earth_N = mkN "terra" ; -- -ae f. (Langenscheidts)
1705 +  egg_N = mkN "ovum" ; -- -i n. (Langenscheidts)
1706 +  eye_N = mkN "oculus" ; -- -i m. (Langenscheidts)
1707 +  fat_N = mkN "omentum" ; -- -i n. (Langenscheidts)
1708 +  feather_N = mkN "penna" ; -- -ae f. (Langenscheidts)
1709 +  fingernail_N = mkN "unguis" "unguis" masculine ; -- -is m. (Langenscheidts)
1710 +  fire_N = mkN "ignis" "ignis" masculine; -- -is m. (Langenscheidts)
1711 +  flower_N = mkN "flos" "floris" masculine ; -- floris m. (Langenscheidts)
1712 +  fog_N = mkN "nebula" ; -- -ae f. (Langenscheidts)
1713 +  foot_N = mkN "pes" "pedis" masculine ; -- pedis m. (Langenscheidts)
1714 +  forest_N = mkN "silva" ; -- -ae f. (Langenscheidts)
1715 +  grass_N = mkN "gramen" "graminis" neuter ; -- -inis n. (Langenscheidts)
1716 +  guts_N = mkN "intestinum" ; -- -i n. (Langenscheidts)
1717 +  hair_N = mkN "capillus" ; -- -i m. (Langenscheidts)
1718 +  hand_N = mkN "manus" "manus" feminine ; -- -us f. (Langenscheidts)
1719 +  head_N = mkN "caput" "capitis" neuter ; -- -itis n. (Langenscheidts)
1720 +  heart_N = mkN "cor" "cordis" neuter; -- cordis n. (Langenscheidts)
1721 +  horn_N = mkN ( variants { "cornu" ; "cornus" } ) "cornus" neuter ; -- -us n. (Langenscheidts)
1722 +  husband_N = mkN "maritus" ; -- -i m. (Langenscheidts)
```

```
1723 +  ice_N = mkN "glacies" "glaciei" feminine ; -- -ei f. (Langenscheidts)
1724 +  knee_N = mkN "genu" "genus" neuter ; -- -us n. (Langenscheidts)
1725 +  leaf_N = mkN "folium" ; -- -i n. (Langenscheidts)
1726 +  leg_N = bone_N ;
1727 +  liver_N = variants { ( mkN "iecur" "iecoris" neuter ) ; ( mkN "iocur" "iocineris" neuter ) } ; -- iecoris/iocineris n. (Langenscheidts)
1728 +  louse_N = mkN "pedis" "pedis" ( variants { masculine ; feminine } ) ; -- -is m./f. (Langenscheidts)
1729 +  mouth_N = mkN "os" "oris" neuter ; -- oris n. (Langenscheidts)
1730 +  name_N = mkN "nomen" "nominis" neuter ; -- -inis n. (Langenscheidts)
1731 +  neck_N = mkN "cervix" "cervicis" feminine ; -- -icis f. (meist pl.) (Langenscheidts)
1732 +  night_N = mkN "nox" "noctis" feminine ; -- noctis f. (Langenscheidts)
1733 +  nose_N = mkN ( variants { "nasus" ; "nasum" } ) ; -- -i m./n. (Langenscheidts)
1734 +  person_N = mkN "persona" ; -- -ae f. (Langenscheidts)
1735 +  rain_N = mkN "pluvia" ; -- -ae f. (Langenscheidts)
1736 +  road_N = mkN "via" ; -- -ae f. (Langenscheidts)
1737 +  root_N = mkN "radix" "radicis" feminine ; -- -icis f. (Langenscheidts)
1738 +  rope_N = mkN "funis" "funis" ( variants { masculine ; feminine } ) ; -- -is m.(/f.) (Langenscheidts)
1739 +  salt_N = mkN "sal" "salis" ( variants { masculine ; neuter } ) ; -- salis m./n. (Langenscheidts)
1740 +  sand_N = mkN "arena" ; -- -ae f. (Langenscheidts)
1741 +  seed_N = mkN "semen" "seminis" neuter ; -- -inis n. (Langenscheidts)
1742 +  skin_N = mkN "cutis" "cutis" feminine ; -- -is f. (Langenscheidts)
1743 +  sky_N = mkN "caelum" ; -- -i n. (Langenscheidts)
1744 +  smoke_N = mkN "fumus" ; -- -i m. (Langenscheidts)
1745 +  snow_N = mkN "nix" "nivis" feminine ; -- nivis (gen. pl. -ium) f. (Langenscheidts)
1746 +  stick_N = mkN ( variants { "baculum" ; "baculus" } ) ; -- -i n./m.
1747 +  tail_N = mkN "cauda" ; -- -ae f. (Langenscheidts)
1748 +  tongue_N = mkN "lingua" ; -- -ae f. (Langenscheidts)
1749 +  tooth_N = mkN "dens" "dentis" masculine; -- dentis m. (Langenscheidts)
1750 +  wife_N = mkN "mulier" "mulieris" feminine; -- -eris f. (Langenscheidts)
1751 +  wind_N = mkN "ventus" ; --  -i m. (Langenscheidts)
1752 +  wing_N = mkN "ala" ; -- -ae f. (Langenscheidts)
1753 +  worm_N = mkN "vermis" "vermis" masculine ; -- -is m. (Langenscheidts)
1754 +  year_N = mkN "annus" ; -- -i m. (Langenscheidts)
1755 +  blow_V = mkV "flare" ; -- flo 1 (Langenscheidts)
1756 +  breathe_V = mkV "spirare" ; -- spiro 1 (Langenscheidts)
1757 +  burn_V = mkV "ardere" ; -- ardeo, arsi, arsum 2 (Langenscheidts)
1758 +  dig_V = mkV "fodere" "fodio" "fodi" "fossum" ; -- fodio, fodi, fossum 3 (Langenscheidts)
1759 +  fall_V = mkV "caedere" "caedo" "cecidi" "caesum" ; -- caedo, cecidi, caesum 3 (Langenscheidts)
1760 +  float_V = mkV "fluitere" ; -- fluito 1 (Langenscheidts)
1761 +  flow_V = mkV "fluere" "fluo" "fluxi" "fluctum" ; -- fluo, fluxi, fluxum 3 (Langenscheidts)
1762 +  fly_V = mkV "volare" ; -- volo 1 (Langenscheidts)
1763 +  freeze_V = mkV "gelare" ; -- gelo 1 (Langenscheidts)
1764 +  -- Category not yet implemented
1765 +  give_V3 = mkV3 (mkV "donare" ) from_Prep to_Prep ;
1766 +  laugh_V = mkV "ridere" ; -- rideo, -si, -sum 2 (Langenscheidts)
1767 +  lie_V = mkV "iacere" ; -- iaceo, iacui, - 2 (Langenscheidts)
1768 +  play_V = mkV "ludere" ; -- ludo, -si, -sum 3 (Langenscheidts)
1769 +  sew_V = mkV "serere" "sero" "sevi" "satum" ; -- sero, sevi, satum 3 (Langenscheidts)
1770 +  sing_V = mkV "cantare" ; -- canto 1 (Langenscheidts)
1771 +  sit_V = mkV "sedere" ; -- sedeo, sedi, sessum 2 (Langenscheidts)
1772 +  smell_V = mkV "olere" ; -- oleo, -ui, - 2 (Langenscheidts)
1773 +  spit_V = mkV "spuere" "spuo" "spui" "sputum" ; -- spuo, -ui, -utum 3 (Langenscheidts)
1774 +  stand_V = mkV "stare" ; -- sto, steti, staturus, statum 1 (Langenscheidts)
1775 +  swell_V = mkV "intumescere" "intumesco" "intumui" ; -- intumesco, -mui, - 3 (Langenscheidts)
1776 +  swim_V = mkV "natare" ; -- nato 1 (Langenscheidts)
1777 +  think_V = mkV "cogitare" ; -- cogito 1 (Langenscheidts)
1778 +  turn_V = mkV "vertere" ; -- verso 1 (Langenscheidts)
1779 +  vomit_V = mkV "vomere" "vomo" "vomui" "vomitum" ; -- vomo, -ui, -itum 3 (Langenscheidts)
1780 +
1781 +  bite_V2 = mkV2 "mordere" ; -- mordeo, momordi, morsum 2 (Langenscheidts)
1782 +  count_V2 = mkV2 (mkV "numerare") ; -- numero 1 (Langenscheidts)
1783 +  cut_V2 = mkV2 (mkV "secare" ) ; -- seco, secui, sectum, secaturus 1 (Langenscheidts)
1784 +  fear_V2 = mkV2 (mkV "timere") ; -- timeo, ui, - 2 (Langenscheidts)
1785 +  fight_V2 = mkV2 (mkV "pugnare" ) ; -- pugno 1 (Langenscheidts)
1786 +  hit_V2 = mkV2 ( mkV "ferire" ) ; -- ferio, -, - 4 (Langenscheidts)
1787 +  hold_V2 = mkV2 ( mkV "tenere" ) ; -- teneo, tenui, tentum 2 (Langenscheidts)
1788 +  hunt_V2 = mkV2 (mkV "agitare") ; -- agito 1 (Langenscheidts)
1789 +  kill_V2 = mkV2 (mkV "necare") ; -- neco 1 (Langenscheidts)
1790 +  pull_V2 = mkV2 (mkV "trahere" "traho" "traxi" "tractum" ) ; -- traho, traxi, tractum 3 (Langenscheidts)
1791 +  push_V2 = mkV2 (mkV "premere" "premo" "pressi" "pressum" ) ; -- premo, pressi, pressum 3 (Langenscheidts)
1792 +  rub_V2 = mkV2 (mkV "radere" "rado" "rasi" "rasum" ) ; -- raso, -si, -sum 3 (Langenscheidts)
1793 +  scratch_V2 = mkV2 (mkV "scalpere" "scalpo" "scalpsi" "scalptum" ) ; -- scalpo, -psi, -ptum 3 (Langenscheidts)
1794 +  split_V2 = mkV2 ( mkV "scindere" "scindo" "scidi" "scissum" ) ; -- scindo, -idi, -issum 3 (Langenscheidts)
1795 +  squeeze_V2 = mkV2 (mkV "premere" "premo" "pressi" "pressum" ) ; -- premo, pressi, pressum 3 (Langenscheidts)
1796 +  stab_V2 = mkV2 (mkV "transfigere" "transfigo" "transfixi" "transfixum") ; -- -figo, -fixi, fixum 3 (Langenscheidts)
1797 +  suck_V2 = mkV2 (mkV "fellare") ; -- fel(l)o 1 (Langenscheidts)
```

```
1798 +  throw_V2 = mkV2 (mkV "iacere" "iacio" "ieci" "iactum" ) ; -- iacio, ieci, iactum 3 (Langenscheidts)
1799 +  tie_V2 = mkV2 (mkV "vincire") ; -- vincio, vinxi, vinctum 4 (Langenscheidts)
1800 +  wash_V2 = mkV2 (mkV "lavare") ; -- lavo, lavi, lautum (lotum)/lavatum 1 (Langenscheidts)
1801 +  wipe_V2 = mkV2 (mkV "detergere") ; -- detergeo, -tersi, -tersum 2/ detergo, -, - 3 (Langenscheidts)
1802 +
1803 +---  other_A = mkA "other" ;
1804 +
1805 +  grammar_N = mkN "grammatica" ; -- -ae/-orum f./n. (http://la.wikipedia.org/wiki/Grammatica) and (Langenscheidts)
1806 +  language_N = mkN "lingua" ; -- -ae f. (Langenscheidts)
1807 +  rule_N = mkN "regula" ; -- -ae f. (Langenscheidts)
1808 +
1809 +-- added 4/6/2007
1810 +    john_PN = mkPN (mkN "Iohannes") ; -- (http://en.wikipedia.org/wiki/John_(given_name))
1811 +    question_N = mkN "rogatio" "rogationis" feminine; -- -onis f. (Langenscheidts)
1812 +    ready_A = mkA "paratus" ; -- 3 (Langenscheidts)
1813 +    reason_N = mkN "causa" ; -- -ae f. (Langenscheidts)
1814 +    today_Adv = mkAdv "hodie" ; -- (Langenscheidts)
1815 +    uncertain_A = mkA "incertus" ; -- 3 (Langenscheidts)
1816 +
1817 +    alas_Interj = ss "eheu" ;
1818  }
1819 diff --git a/lib/src/latin/MorphoLat.gf b/lib/src/latin/MorphoLat.gf
1820 index 1236145..9c11286 100644
1821 --- a/lib/src/latin/MorphoLat.gf
1822 +++ b/lib/src/latin/MorphoLat.gf
1823 @@ -1,197 +1,624 @@
1824 ----#-path=.:../../prelude
1825 ----
1826 -----1 A Simple Latlish Resource Morphology
1827 -----
1828 ----- Aarne Ranta 2002 -- 2005
1829 -----
1830 ----- This resource morphology contains definitions needed in the resource
1831 ----- syntax. To build a lexicon, it is better to use $ParadigmsLat$, which
1832 ----- gives a higher-level access to this module.
1833 ----
1834 ---resource MorphoLat = ResLat ** open Prelude, (Predef=Predef) in {
1835 +--#-path=.:../prelude
1836 +
1837 +--1 A Simple Latin Resource Morphology.
1838 +
1839 +-- Herbert Lange 2013
1840 +
1841 +-- This resource morphology contains definitions needed in the resource
1842 +-- syntax. To build a lexicon, it is better to use $ParadigmsLat$, which
1843 +-- gives a higher-level access to this module.
1844 +
1845 +resource MorphoLat = ParamX, ResLat ** open Prelude in {
1846  --
1847  --  flags optimize=all ;
1848  --
1849 +
1850  ----2 Phonology
1851 -----
1852 ----- To regulate the use of endings for both nouns, adjectives, and verbs:
1853 ----
1854 ---oper
1855 ----  y2ie : Str -> Str -> Str = \fly,s ->
1856 ----    let y = last (init fly) in
1857 ----    case y of {
1858 ----      "a" => fly + s ;
1859 ----      "e" => fly + s ;
1860 ----      "o" => fly + s ;
1861 ----      "u" => fly + s ;
1862 ----    _   => init fly + "ie" + s
1863 ----    } ;
1864 ----
1865 ----
1866 -----2 Nouns
1867 -----
1868 ----- For conciseness and abstraction, we define a worst-case macro for
1869 ----- noun inflection. It is used for defining special case that
1870 ----- only need one string as argument.
1871 ----
1872 ---oper
```

```
1873 ---  CommonNoun : Type = {s : Number => Case => Str} ;
1874 ---
1875 ---  nounGen : Str -> CommonNoun = \dog -> case last dog of {
1876 ---    "y" => nounY "dog" ;
1877 ---    "s" => nounS (init "dog") ;
1878 ---    _   => nounReg "dog"
1879 ---    } ;
1880 ---
1881 ----- These are auxiliaries to $nounGen$.
1882 ---
1883 ---  nounReg : Str -> CommonNoun = \dog ->
1884 ---    mkNoun dog (dog + "s") (dog + "'s") (dog + "s'");
1885 ---  nounS   : Str -> CommonNoun = \kiss ->
1886 ---    mkNoun kiss (kiss + "es") (kiss + "'s") (kiss + "es'") ;
1887 ---  nounY   : Str -> CommonNoun = \fl ->
1888 ---    mkNoun (fl + "y") (fl + "ies") (fl + "y's") (fl + "ies'") ;
1889 ---
1890 ---
1891 +oper
1892 +  -- sounds and sound changes
1893 +  vowel : pattern Str = #( "a" | "e" | "o" | "u" | "y" );
1894 +  semivowel : pattern Str = #( "j" | "w" );
1895 +  consonant : pattern Str = #( "p" | "b" | "f" | "v" | "m" | "t" | "d" | "s" | "z" | "n" | "r" | "c" | "g" | "l" | "q" | "qu" | "h" );
1896 +  stop : pattern Str = #( "p" | "b" | "t" | "d" | "c" | "q" | "q" );
1897 +  fricative : pattern Str = #( "f" | "v" | "s" | "z" | "h" );
1898 +  nasal : pattern Str = #( "m" | "n" );
1899 +  liquid : pattern Str = #( "r" | "l" );
1900 +--  consonant : pattern Str = #(#stop | #fricative | #nasal | #liquid ); -- not working
1901 +----2 Nouns
1902 +
1903 +-- declensions
1904 +oper
1905 +
1906 +  -- a-Declension
1907 +  noun1 : Str -> Noun = \mensa ->
1908 +    let
1909 +      mensae = mensa + "e" ;
1910 +      mensis = init mensa + "is" ;
1911 +    in
1912 +    mkNoun
1913 +      mensa (mensa +"m") mensae mensae mensa mensa
1914 +      mensae (mensa + "s") (mensa + "rum") mensis
1915 +      Fem ;
1916 +
1917 +  -- o-Declension
1918 +  noun2us : Str -> Noun = \servus ->
1919 +    let
1920 +      serv = Predef.tk 2 servus ;
1921 +      servum = serv + "um" ;
1922 +      servi = serv + "i" ;
1923 +      servo = serv + "o" ;
1924 +    in
1925 +    mkNoun
1926 +      servus servum servi servo servo (serv + "e")
1927 +      servi (serv + "os") (serv + "orum") (serv + "is")
1928 +      Masc ;
1929 +
1930 +  noun2er : Str -> Str -> Noun = \liber,libri ->
1931 +    let
1932 +      libr : Str = Predef.tk 1 libri;
1933 +      librum = libr + "um" ;
1934 +      libri = libr + "i" ;
1935 +      libro = libr + "o" ;
1936 +    in
1937 +    mkNoun
1938 +      liber librum libri libro libro liber
1939 +      libri ( libr + "os" ) ( libr + "orum" ) ( libr + "is" )
1940 +      Masc ;
1941 +
1942 +  noun2um : Str -> Noun = \bellum ->
1943 +    let
1944 +      bell = Predef.tk 2 bellum ;
1945 +      belli = bell + "i" ;
1946 +      bello = bell + "o" ;
1947 +      bella = bell + "a" ;
```

```
1948 +    in
1949 +    mkNoun
1950 +      bellum bellum belli bello bello (bell + "um")
1951 +      bella bella (bell + "orum") (bell + "is")
1952 +      Neutr ;
1953 +
1954 +  -- Consonant declension
1955 +  noun3c : Str -> Str -> Gender -> Noun = \rex,regis,g ->
1956 +    let
1957 +      reg : Str = Predef.tk 2 regis ;
1958 +      regemes : Str * Str = case g of {
1959 +       Masc | Fem => < reg + "em" , reg + "es" > ;
1960 +       Neutr => < rex , reg + "a" >
1961 +       } ;
1962 +    in
1963 +    mkNoun
1964 +      rex regemes.p1 regis ( reg + "i" ) ( reg + "e" ) rex
1965 +      regemes.p2 regemes.p2 ( reg + "um" ) ( reg + "ibus" )
1966 +      g ;
1967 +
1968 +  -- i-declension
1969 +  noun3i : Str -> Str -> Gender -> Noun = \ars,artis,g ->
1970 +    let
1971 +      art : Str = Predef.tk 2 artis ;
1972 +      artemes : Str * Str = case g of {
1973 +       Masc | Fem => < art + "em" , art + "es" > ;
1974 +       Neutr => case art of {
1975 +         _ + #consonant + #consonant => < ars , art + "a" > ; -- maybe complete fiction but may be working
1976 +         _ => < ars , art + "ia" > -- Bayer-Lindauer 32 4
1977 +         }
1978 +       } ;
1979 +      arte : Str = case ars of {
1980 +       _ + ( "e" | "al" | "ar" ) => art + "i" ;
1981 +       _ => art + "e"
1982 +       };
1983 +    in
1984 +    mkNoun
1985 +      ars artemes.p1 artis ( art + "i" ) arte ars
1986 +      artemes.p2 artemes.p2 ( art + "ium" ) ( art + "ibus" )
1987 +      g ;
1988 +
1989 +  -- u-Declension
1990 +
1991 +  noun4us : Str -> Noun = \fructus ->
1992 +    let
1993 +      fructu = init fructus ;
1994 +      fruct  = init fructu
1995 +    in
1996 +    mkNoun
1997 +      fructus (fructu + "m") fructus (fructu + "i") fructu fructus
1998 +      fructus fructus (fructu + "um") (fruct + "ibus")
1999 +      Masc ;
2000 +
2001 +  noun4u : Str -> Noun = \cornu ->
2002 +    let
2003 +      corn = init cornu ;
2004 +      cornua = cornu + "a"
2005 +    in
2006 +    mkNoun
2007 +      cornu cornu (cornu + "s") cornu cornu cornu
2008 +      cornua cornua (cornu + "um") (corn + "ibus")
2009 +      Neutr ;
2010 +
2011 +  -- e-Declension
2012 +  noun5 : Str -> Noun = \res ->
2013 +    let
2014 +      re = init res ;
2015 +      rei = re + "i"
2016 +    in
2017 +    mkNoun
2018 +      res (re+ "m") rei rei re res
2019 +      res res (re + "rum") (re + "bus")
2020 +      Fem ;
2021 +
2022 +  -- smart paradigms
```

```
+
+ noun_ngg : Str -> Str -> Gender -> Noun = \verbum,verbi,g ->
+   let s : Noun = case <verbum,verbi> of {
+     <_ + "a", _ + "ae"> => noun1 verbum ;
+     <_ + "us", _ + "i"> => noun2us verbum ;
+     <_ + "um", _ + "i"> => noun2um verbum ;
+     <_ + ( "er" | "ir" ) , _ + "i"> => noun2er verbum verbi ;
+
+     <_ + "us", _ + "us"> => noun4us verbum ;
+     <_ + "u", _ + "us"> => noun4u verbum ;
+     <_ + "es", _ + "ei"> => noun5 verbum ;
+     _ => noun3 verbum verbi g
+     }
+   in
+   nounWithGen g s ;
+
+ noun : Str -> Noun = \verbum ->
+   case verbum of {
+     _ + "a" => noun1 verbum ;
+     _ + "us" => noun2us verbum ;
+     _ + "um" => noun2um verbum ;
+     _ + ( "er" | "ir" ) => noun2er verbum ( (Predef.tk 2 verbum) + "ri" ) ;
+     _ + "u" => noun4u verbum ;
+     _ + "es" => noun5 verbum ;
+     _ => Predef.error ("3rd declinsion cannot be applied to just one noun form" ++ verbum)
+     } ;
+
+
+ noun12 : Str -> Noun = \verbum ->
+   case verbum of {
+     _ + "a" => noun1 verbum ;
+     _ + "us" => noun2us verbum ;
+     _ + "um" => noun2um verbum ;
+     _ + ( "er" | "ir" ) =>
+       let
+         puer = verbum ;
+         pue = Predef.tk 1 puer ;
+         e = case puer of {
+           -- Exception of nouns where e is part of the word stem Bayer-Lindauer 27 4.2
+           "puer" | "socer" | "gener" | "vesper" => "e" ;
+           -- Exception of adjectives where e is part of the word stem 31 3.2
+           ("asper" | "miser" | "tener" | "frugifer") + _ => "e";
+           -- "liber" => ( "e" | "" ) ; conflicting with noun liber
+           _ => ""
+           } ;
+         pu = Predef.tk 1 pue ;
+       in noun2er verbum ( pu + e + "ri" );
+     _ => Predef.error ("noun12 does not apply to" ++ verbum)
+   } ;
+
+ 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
+     < ( "sedes" | "canis" | "iuvenis" | "mensis" | "sal" ) , _ > => noun3c rex regis g ;  -- Bayer-Lindauer 31 3 and Exercitia Latina 32 b)
+     < _ + ( "e" | "al" | "ar" ) , _ > => noun3i rex regis g ; -- Bayer-Lindauer 32 2.3
+     ( < _ + "ter" , _ + "tr" >
+     | < _ + "en" , _ + "in" >
+     | < _ + "s" , _ + "r" >
+     ) => noun3c rex regis g ; -- might not be right but seems fitting for Bayer-Lindauer 31 2.2
+     < _ , _ + #consonant + #consonant > => noun3i rex regis g ; -- Bayer-Lindauer 32 2.2
+     < _ + ( "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
+   } ;
```

```
2098 +
2099 +
2100  ----3 Proper names
2101 ----
2102 ---- Regular proper names are inflected with "'s" in the genitive.
2103 ----
2104 ----  nameReg : Str -> Gender -> {s : Case => Str} = \john,g ->
2105 ----    {s = table {Gen => john + "'s" ; _ => john} ; g = g} ;
2106 ----
2107 +
2108  ----2 Determiners
2109 ----
2110 ----  mkDeterminer : Number -> Str -> {s,sp : Str ; n : Number} = \n,s ->
2111 ----    {s,sp = s ; n = n} ;
2112 ----
2113 +
2114  ----2 Pronouns
2115 ----
2116 ---- Here we define personal pronouns.
2117 ----
2118 ---- We record the form "mine" and the gender for later use.
2119 ----
2120 ----  Pronoun : Type =
2121 ----    {s : Case => Str ; a : Agr} ;
2122 ----
2123 ----  mkPronoun : (_,_,_,_ : Str) -> Number -> Person -> Gender -> Pronoun =
2124 ----   \I,me,my,mine,n,p,g ->
2125 ----    {s = table {Nom => I ; Acc => me ; Gen => my} ;
2126 ----     a = toAgr n p g
2127 ----    } ;
2128 ----
2129 ----  human : Gender = Masc ; ---- doesn't matter
2130 ----
2131 ----  pronI = mkPronoun "I" "me" "my" "mine" Sg P1 human ;
2132 ----  pronYouSg = mkPronoun "you" "you" "your" "yours" Sg P2 human ; -- verb agr OK
2133 ----  pronHe = mkPronoun "he" "him" "his" "his" Sg P3 Masc ;
2134 ----  pronShe = mkPronoun "she" "her" "her" "hers" Sg P3 Fem ;
2135 ----  pronIt = mkPronoun "it" "it" "its" "it" Sg P3 Neutr ;
2136 ----
2137 ----  pronWe = mkPronoun "we" "us" "our" "ours" Pl P1 human ;
2138 ----  pronYouPl = mkPronoun "you" "you" "your" "yours" Pl P2 human ;
2139 ----  pronThey = mkPronoun "they" "them" "their" "theirs" Pl P3 human ; ----
2140 ----
2141 ----
2142 +
2143  ----2 Adjectives
2144 ----
2145 ---- To form the adjectival and the adverbial forms, two strings are needed
2146 ---- in the worst case. (First without degrees.)
2147 ----
2148 ----  Adjective = {s : AForm => Str} ;
2149 ----
2150 ---- However, most adjectives can be inflected using the final character.
2151 ---- N.B. this is not correct for "shy", but $mkAdjective$ has to be used.
2152 ----
2153 ----  regAdjective : Str -> Adjective = \free ->
2154 ----    let
2155 ----      e   = last free ;
2156 ----      fre = init free ;
2157 ----      freely = case e of {
2158 ----        "y" => fre + "ily" ;
2159 ----        _   => free + "ly"
2160 ----        } ;
2161 ----      fre = case e of {
2162 ----        "e" => fre ;
2163 ----        "y" => fre + "i" ;
2164 ----        _   => free
2165 ----        }
2166 ----    in
2167 ----    mkAdjective free (fre + "er") (fre + "est") freely ;
2168 ----
2169 ---- Many adjectives are 'inflected' by adding a comparison word.
2170 ----
2171 ----  adjDegrLong : Str -> Adjective = \ridiculous ->
2172 ----    mkAdjective
```

```
2173 ——       ridiculous
2174 ——       ("more" ++ ridiculous)
2175 ——       ("most" ++ ridiculous)
2176 ——       ((regAdjective ridiculous).s ! AAdv) ;
2177 ——
2178 ——
2179 +oper
2180 +  comp_super : Noun -> ( Agr => Str ) * ( Agr => Str ) =
2181 +    \bonus ->
2182 +    case bonus.s!Sg!Gen of {
2183 +      -- Exception Bayer-Lindauer 50 1
2184 +      "boni" => < comp "meli" , table { Ag g n c => table Gender [ (noun2us "optimus").s ! n ! c ; (noun1 "optima").s ! n ! c ; (noun2um "opti
2185 +      "mali" => < comp "pei" , super "pessus" > ;
2186 +      "magni" => < comp "mai" , table { Ag g n c => table Gender [ (noun2us "maximus").s ! n ! c ; (noun1 "maxima").s ! n ! c ; (noun2um "max
2187 +      "parvi" => < comp "mini" , table { Ag g n c => table Gender [ (noun2us "minimus").s ! n ! c ; (noun1 "minima").s ! n ! c ; (noun2um "mi
2188 +      --Exception Bayer-Lindauer 50.3
2189 +      "novi" => < comp "recenti" , super "recens" > ;
2190 +      "feri" => < comp "feroci" , super "ferox" > ;
2191 +      "sacris" => < comp "sancti" , super "sanctus" >;
2192 +      "frugiferi" => < comp "fertilis" , super "fertilis" > ;
2193 +      "veti" => < comp "vetusti" , super "vetustus" >;
2194 +      "inopis" => < comp "egentis" , super "egens" >;
2195 +      -- Default Case use Singular Genetive to determine comparative
2196 +      sggen => < comp sggen , super (bonus.s!Sg!Nom) >
2197 +    } ;
2198 +
2199 +  comp : Str -> ( Agr => Str ) = \boni -> -- Bayer-Lindauer 46 2
2200 +    case boni of {
2201 +      bon + ( "i" | "is" ) =>
2202 +       table
2203 +       {
2204 +         Ag ( Fem | Masc ) Sg c => table Case [ bon + "ior" ;
2205 +                                   bon + "iorem" ;
2206 +                                   bon + "ioris" ;
2207 +                                   bon + "iori" ;
2208 +                                   bon + "iore";
2209 +                                   bon + "ior" ] ! c ;
2210 +         Ag ( Fem | Masc ) Pl c => table Case [ bon + "iores" ;
2211 +                                   bon + "iores" ;
2212 +                                   bon + "iorum" ;
2213 +                                   bon + "ioribus" ;
2214 +                                   bon + "ioribus" ;
2215 +                                   bon + "iores" ] ! c ;
2216 +         Ag Neutr Sg c => table Case [ bon + "ius" ;
2217 +                                   bon + "ius" ;
2218 +                                   bon + "ioris" ;
2219 +                                   bon + "iori" ;
2220 +                                   bon + "iore" ;
2221 +                                   bon + "ius" ] ! c ;
2222 +         Ag Neutr Pl c => table Case [ bon + "iora" ;
2223 +                                   bon + "iora" ;
2224 +                                   bon + "iorum" ;
2225 +                                   bon + "ioribus" ;
2226 +                                   bon + "ioribus" ;
2227 +                                   bon + "iora" ] ! c
2228 +       }
2229 +    } ;
2230 +
2231 +  super : Str -> ( Agr => Str ) = \bonus ->
2232 +    let
2233 +      prefix : Str = case bonus of {
2234 +       ac + "er" => bonus ; -- Bayer-Lindauer 48 2
2235 +       faci + "lis" => faci + "l" ; -- Bayer-Lindauer 48 3
2236 +       feli + "x" => feli + "c" ; -- Bayer-Lindauer 48 1
2237 +       ege + "ns" => ege + "nt" ; -- Bayer-Lindauer 48 1
2238 +       bon + ( "us" | "is") => bon -- Bayer-Lindauer 48 1
2239 +       };
2240 +      suffix : Str = case bonus of {
2241 +       ac + "er" => "rim" ; -- Bayer-Lindauer 48 2
2242 +       faci + "lis" => "lim" ; -- Bayer-Lindauer 48 3
2243 +       _ => "issim" -- Bayer-Lindauer 48 1
2244 +       };
2245 +    in
2246 +    table {
2247 +      Ag Fem n c => (noun1 ( prefix + suffix + "a" )).s ! n ! c ;
```

```
2248 +      Ag Masc n c => (noun2us ( prefix + suffix + "us" )).s ! n ! c;
2249 +      Ag Neutr n c => (noun2um ( prefix + suffix + "um" )).s ! n ! c
2250 +    } ;
2251 +
2252 +  adj12 : Str -> Adjective = \bonus ->
2253 +    let
2254 +      bon : Str = case bonus of {
2255 +       -- Exceptions Bayer-Lindauer 41 3.2
2256 +       ("asper" | "liber" | "miser" | "tener" | "frugifer") => bonus ;
2257 +       -- Usual cases
2258 +       pulch + "er" => pulch + "r" ;
2259 +       bon + "us" => bon ;
2260 +       _ => Predef.error ("adj12 does not apply to" ++ bonus)
2261 +       } ;
2262 +      nbonus = (noun12 bonus) ;
2263 +      compsup : ( Agr => Str ) * ( Agr => Str ) =
2264 +       -- Bayer-Lindauer 50 4
2265 +       case bonus of {
2266 +         (_ + #vowel + "us" ) |
2267 +           (_ + "r" + "us" ) =>
2268 +           < table { Ag g n c => table Gender [ ( noun12 bonus ).s ! n ! c ; ( noun12 ( bon + "a" ) ).s ! n ! c ; ( noun12 ( bon + "um" ) ).s
2269 +           table { Ag g n c => table Gender [ ( noun12 bonus ).s ! n ! c ; ( noun12 ( bon + "a" ) ).s ! n ! c ; ( noun12 ( bon + "um" ) ).s !
2270 +         _ => comp_super nbonus
2271 +       };
2272 +      advs : Str * Str =
2273 +       case bonus of {
2274 +         -- Bayer-Lindauer 50 4
2275 +         idon + ( #vowel | "r" ) + "us" => < "magis" , "maxime" > ;
2276 +         _ => < "" , "" >
2277 +       }
2278 +    in
2279 +    mkAdjective
2280 +    nbonus
2281 +    (noun1 (bon + "a"))
2282 +    (noun2um (bon + "um"))
2283 +    < compsup.p1 , advs.p1 >
2284 +    < compsup.p2 , advs.p2 > ;
2285 +
2286 +  adj3x : (_,_ : Str) -> Adjective = \acer,acris ->
2287 +   let
2288 +     ac = Predef.tk 2 acer ;
2289 +     acrise : Str * Str = case acer of {
2290 +       _ + "er" => <ac + "ris", ac + "re"> ;
2291 +       _ + "is" => <acer      , ac + "e"> ;
2292 +       _        => <acer      , acer>
2293 +       } ;
2294 +     nacer = (noun3adj acer acris Masc) ;
2295 +     compsuper = comp_super nacer;
2296 +   in
2297 +   mkAdjective
2298 +    nacer
2299 +    (noun3adj acrise.p1 acris Fem)
2300 +    (noun3adj acrise.p2 acris Neutr)
2301 +    < compsuper.p1 , "" >
2302 +    < compsuper.p2 , "" >
2303 +    ;
2304 +
2305 +-- smart paradigms
2306 +
2307 +  adj123 : Str -> Str -> Adjective = \bonus,boni ->
2308 +    case <bonus,boni> of {
2309 +      <_ + ("us" | "er"), _ + "i" > => adj12 bonus ;
2310 +      <_ + ("us" | "er"), _ + "is"> => adj3x bonus boni ;
2311 +      <_                , _ + "is"> => adj3x bonus boni ;
2312 +      <_ + "is"         , _ + "e" > => adj3x bonus boni ;
2313 +      _ => Predef.error ("adj123: not applicable to" ++ bonus ++ boni)
2314 +    } ;
2315 +
2316 +  adj : Str -> Adjective = \bonus ->
2317 +    case bonus of {
2318 +      _ + ("us" | "er") => adj12 bonus ;
2319 +      facil + "is"      => adj3x bonus bonus ;
2320 +      feli  + "x"       => adj3x bonus (feli + "cis") ;
2321 +      _                 => adj3x bonus (bonus + "is") --- any example?
2322 +    } ;
```

```
+
+
 ----3 Verbs
------
------ The worst case needs five forms. (The verb "be" is treated separately.)
--
--  mkVerb4 : (_,_,_,_: Str) -> Verb = \go,goes,went,gone ->
--    let going = case last go of {
--      "e" => init go + "ing" ;
--      _ => go + "ing"
--      }
--    in
--    mkVerb go goes went gone going ;
--
------ This is what we use to derive the irregular forms in almost all cases
--
--  mkVerbIrreg : (_,_,_ : Str) -> Verb = \bite,bit,bitten ->
--    let bites = case last bite of {
--      "y" => y2ie bite "s" ;
--      "s" => init bite + "es" ;
--      _   => bite + "s"
--      }
--    in mkVerb4 bite bites bit bitten ;
--
------ This is used to derive regular forms.
--
--  mkVerbReg : Str -> Verb = \soak ->
--    let
--      soaks = case last soak of {
--        "y" => y2ie soak "s" ;
--        "s" => init soak + "es" ;
--        _   => soak + "s"
--        } ;
--      soaked = case last soak of {
--        "e" => init soak + "s" ;
--        _   => soak + "ed"
--        }
--    in
--    mkVerb4 soak soaks soaked soaked ;
--
--  verbGen : Str -> Verb = \kill -> case last kill of {
--    "y" => verbP3y (init kill) ;
--    "e" => verbP3e (init kill) ;
--    "s" => verbP3s (init kill) ;
--    _   => regVerbP3 kill
--    } ;
--
------ These are just auxiliary to $verbGen$.
--
--  regVerbP3 : Str -> Verb = \walk ->
--    mkVerbIrreg walk (walk + "ed") (walk + "ed") ;
--  verbP3s   : Str -> Verb = \kiss ->
--    mkVerb4 kiss (kiss + "es") (kiss + "ed") (kiss + "ed") ;
--  verbP3e   : Str -> Verb = \love ->
--    mkVerb4 love (love + "s") (love + "d") (love + "d") ;
--  verbP3y   : Str -> Verb = \cr ->
--    mkVerb4 (cr + "y") (cr + "ies") (cr + "ied") (cr + "ied") ;
--
------ The particle always appears right after the verb.
--
--  verbPart : Verb -> Str -> Verb = \v,p ->
--    {s = \\f => v.s ! f ++ p ; isRefl = v.isRefl} ;
--
--  verbNoPart : Verb -> Verb = \v -> verbPart v [] ;
--
--
--} ;
--
+
+-- 1./a-conjugation
+
+ verb1 : Str -> Verb = \laudare ->
+   let
+     lauda = Predef.tk 2 laudare ;
+     laud = init lauda ;
```

```
2398 +      laudav = lauda + "v" ;
2399 +      pres_stem = lauda ;
2400 +      pres_ind_base = lauda ;
2401 +      pres_conj_base = laud + "e" ;
2402 +      impf_ind_base = lauda + "ba" ;
2403 +      impf_conj_base = lauda + "re" ;
2404 +      fut_I_base = lauda + "bi" ;
2405 +      imp_base = lauda ;
2406 +      perf_stem = laudav ;
2407 +      perf_ind_base = laudav ;
2408 +      perf_conj_base = laudav + "eri" ;
2409 +      pqperf_ind_base = laudav + "era" ;
2410 +      pqperf_conj_base = laudav + "isse" ;
2411 +      fut_II_base = laudav + "eri" ;
2412 +      part_stem = lauda + "t" ;
2413 +    in
2414 +    mkVerb laudare pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base
2415 +    perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
2416 +
2417 +  -- 2./e-conjugation
2418 +
2419 +  verb2 : Str -> Verb = \monere ->
2420 +    let
2421 +      mone = Predef.tk 2 monere ;
2422 +      mon = init mone ;
2423 +      monu = mon + "u" ;
2424 +      pres_stem = mone ;
2425 +      pres_ind_base = mone ;
2426 +      pres_conj_base = mone + "a" ;
2427 +      impf_ind_base = mone + "ba" ;
2428 +      impf_conj_base = mone + "re" ;
2429 +      fut_I_base = mone + "bi" ;
2430 +      imp_base = mone ;
2431 +      perf_stem = monu ;
2432 +      perf_ind_base = monu ;
2433 +      perf_conj_base = monu + "eri" ;
2434 +      pqperf_ind_base = monu + "era" ;
2435 +      pqperf_conj_base = monu + "isse" ;
2436 +      fut_II_base = monu + "eri" ;
2437 +      part_stem = mon + "it" ;
2438 +    in
2439 +    mkVerb monere pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base
2440 +    perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
2441 +
2442 +-- 3./Consonant conjugation
2443 +
2444 +  verb3c : ( regere,rexi,rectus : Str ) -> Verb = \regere,rexi,rectus ->
2445 +    let
2446 +      rege = Predef.tk 2 regere ;
2447 +      reg = init rege ;
2448 +      rex = init rexi ;
2449 +      rect = Predef.tk 2 rectus ;
2450 +      pres_stem = reg ;
2451 +      pres_ind_base = reg ;
2452 +      pres_conj_base = reg + "a" ;
2453 +      impf_ind_base = reg + "eba" ;
2454 +      impf_conj_base = reg + "ere" ;
2455 +      fut_I_base = rege ;
2456 +      imp_base = reg ;
2457 +      perf_stem = rex ;
2458 +      perf_ind_base = rex ;
2459 +      perf_conj_base = rex + "eri" ;
2460 +      pqperf_ind_base = rex + "era" ;
2461 +      pqperf_conj_base = rex + "isse" ;
2462 +      fut_II_base = rex + "eri" ;
2463 +      part_stem = rect ;
2464 +    in
2465 +    mkVerb regere pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base
2466 +    perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
2467 +
2468 +-- 3./i-conjugation
2469 +
2470 +  verb3i : ( capere,cepi,captus : Str ) -> Verb = \capere,cepi,captus ->
2471 +    let
2472 +      cape = Predef.tk 2 capere ;
```

```
2473 +       cap = init cape ;
2474 +       capi = cap + "i" ;
2475 +       cep = init cepi ;
2476 +       capt = Predef.tk 2 captus ;
2477 +       pres_stem = capi ;
2478 +       pres_ind_base = capi ;
2479 +       pres_conj_base = capi + "a" ;
2480 +       impf_ind_base = capi + "eba" ;
2481 +       impf_conj_base = cape + "re" ;
2482 +       fut_I_base = capi + "e" ;
2483 +       imp_base = cap ;
2484 +       perf_stem = cep ;
2485 +       perf_ind_base = cep ;
2486 +       perf_conj_base = cep + "eri" ;
2487 +       pqperf_ind_base = cep + "era" ;
2488 +       pqperf_conj_base = cep + "isse" ;
2489 +       fut_II_base = cep + "eri" ;
2490 +       part_stem = capt ;
2491 +     in
2492 +     mkVerb capere pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base
2493 +     perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
2494 +
2495 +-- 4./i-conjugation
2496 +
2497 +  verb4 : Str -> Verb = \audire ->
2498 +     let
2499 +       audi = Predef.tk 2 audire ;
2500 +       audiv = audi + "v" ;
2501 +       pres_stem = audi ;
2502 +       pres_ind_base = audi ;
2503 +       pres_conj_base = audi + "a" ;
2504 +       impf_ind_base = audi + "eba" ;
2505 +       impf_conj_base = audi + "re" ;
2506 +       fut_I_base = audi +"e" ;
2507 +       imp_base = audi ;
2508 +       perf_stem = audiv ;
2509 +       perf_ind_base = audiv ;
2510 +       perf_conj_base = audiv + "eri" ;
2511 +       pqperf_ind_base = audiv + "era" ;
2512 +       pqperf_conj_base = audiv + "isse" ;
2513 +       fut_II_base = audiv + "eri" ;
2514 +       part_stem = audi + "t" ;
2515 +     in
2516 +     mkVerb audire pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base
2517 +     perf_stem perf_ind_base perf_conj_base pqperf_ind_base pqperf_conj_base fut_II_base part_stem ;
2518 +
2519 +-- deponent verb
2520 +
2521 +-- 1./a-conjugation
2522 +  deponent1 : Str -> Verb = \hortari ->
2523 +     let
2524 +       horta = Predef.tk 2 hortari ;
2525 +       hort = init horta ;
2526 +       pres_stem = horta ;
2527 +       pres_ind_base = horta ;
2528 +       pres_conj_base = hort + "e" ;
2529 +       impf_ind_base = horta + "ba" ;
2530 +       impf_conj_base = horta + "re" ;
2531 +       fut_I_base = horta + "bi" ;
2532 +       imp_base = horta ;
2533 +       part_stem = horta + "t" ;
2534 +     in
2535 +     mkDeponent hortari pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base part_stem ;
2536 +
2537 +-- 2./e-conjugation
2538 +  deponent2 : Str -> Verb = \vereri ->
2539 +     let
2540 +       vere = Predef.tk 2 vereri ;
2541 +       ver = init vere ;
2542 +       pres_stem = vere ;
2543 +       pres_ind_base = vere ;
2544 +       pres_conj_base = vere + "a" ;
2545 +       impf_ind_base = vere + "ba" ;
2546 +       impf_conj_base = vere + "re" ;
2547 +       fut_I_base = vere + "bi" ;
```

```
2548 +      imp_base = vere ;
2549 +      part_stem = ver + "it" ;
2550 +    in
2551 +    mkDeponent vereri pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base part_stem ;
2552 +
2553 +-- 3./Consonant conjugation
2554 +  deponent3c : ( sequi,sequor,secutus : Str ) -> Verb = \sequi,sequor,secutus ->
2555 +    let
2556 +      sequ = Predef.tk 2 sequor ;
2557 +      secu = Predef.tk 3 secutus ;
2558 +      pres_stem = sequ ;
2559 +      pres_ind_base = sequ ;
2560 +      pres_conj_base = sequ + "a" ;
2561 +      impf_ind_base = sequ + "eba" ;
2562 +      impf_conj_base = sequ + "ere" ;
2563 +      fut_I_base = sequ + "e" ;
2564 +      imp_base = sequi ;
2565 +      part_stem = secu + "t" ;
2566 +    in
2567 +    mkDeponent sequi pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base part_stem ;
2568 +
2569 +-- 3./i-conjugation
2570 +  deponent3i : ( pati,patior,passus : Str ) -> Verb = \pati,patior,passus ->
2571 +    let
2572 +      pat = init pati ;
2573 +      pass = Predef.tk 2 passus ;
2574 +      pres_stem = pati ;
2575 +      pres_ind_base = pati ;
2576 +      pres_conj_base = pati + "a" ;
2577 +      impf_ind_base = pati + "eba" ;
2578 +      impf_conj_base = pat + "ere" ;
2579 +      fut_I_base = pati + "e" ;
2580 +      imp_base = pati ;
2581 +      part_stem = pass ;
2582 +    in
2583 +    mkDeponent pati pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base part_stem ;
2584 +
2585 +-- 4./i-conjugation
2586 +  deponent4 : Str -> Verb = \largiri ->
2587 +    let
2588 +      largi = Predef.tk 2 largiri ;
2589 +      pres_stem = largi ;
2590 +      pres_ind_base = largi ;
2591 +      pres_conj_base = largi + "a" ;
2592 +      impf_ind_base = largi + "eba" ;
2593 +      impf_conj_base = largi + "re" ;
2594 +      fut_I_base = largi + "e" ;
2595 +      imp_base = largi ;
2596 +      part_stem = largi + "t" ;
2597 +    in
2598 +    mkDeponent largiri pres_stem pres_ind_base pres_conj_base impf_ind_base impf_conj_base fut_I_base imp_base part_stem ;
2599 +
2600 +-- smart paradigms
2601 +
2602 +  verb_ippp : (iacere,iacio,ieci,iactus : Str) -> Verb =
2603 +    \iacere,iacio,ieci,iactus ->
2604 +    case iacere of {
2605 +      _ + "ari" => deponent1 iacere ;
2606 +      _ + "eri" => deponent2 iacere ;
2607 +      _ + "iri" => deponent4 iacere ;
2608 +      _ + "i" => case iacio of {
2609 +        _ + "ior" => deponent3i iacere iacio iactus ;
2610 +        _ => deponent3c iacere iacio iactus
2611 +        } ;
2612 +      _ + "are" => verb1 iacere ;
2613 +      _ + "ire" => verb4 iacere ; -- ieci iactus ;
2614 +      _ + "ere" => case iacio of {
2615 +        _ + #consonant + "o" => verb3c iacere ieci iactus ; -- Bayer-Lindauer 74 1
2616 +        _ + "eo" => verb2 iacere ;
2617 +        _ + ( "i" | "u" ) + "o" => verb3i iacere ieci iactus ; -- Bayer-Linduaer 74 1
2618 +        _ => verb3c iacere ieci iactus
2619 +        } ;
2620 +      _ => Predef.error ("verb_ippp: illegal infinitive form" ++ iacere)
2621 +    } ;
2622 +
```

```
2623 +  verb : (iacere : Str) -> Verb =
2624 +    \iacere ->
2625 +    case iacere of {
2626 +      _ + "ari" => deponent1 iacere ;
2627 +      _ + "eri" => deponent2 iacere ;
2628 +      _ + "iri" => deponent4 iacere ;
2629 +      _ + "are" => verb1 iacere ;
2630 +      _ + "ire" => -- let iaci = Predef.tk 2 iacere in
2631 +        verb4 iacere ; -- (iaci + "vi") (iaci + "tus") ;
2632 +      _ + "ere" => verb2 iacere ;
2633 +      _ => Predef.error ("verb: illegal infinitive form" ++ iacere)
2634 +    } ;
2635 +}
2636 \ No newline at end of file
2637 diff --git a/lib/src/latin/NounLat.gf b/lib/src/latin/NounLat.gf
2638 index 01eb80a..ea60fbd 100644
2639 --- a/lib/src/latin/NounLat.gf
2640 +++ b/lib/src/latin/NounLat.gf
2641 @@ -1,132 +1,151 @@
2642 -concrete NounLat of Noun = CatLat ** open ResLat, Prelude in {
2643 +concrete NounLat of Noun = CatLat ** open ResLat, Prelude, ConjunctionLat in {
2644
2645    flags optimize=all_subs ;
2646
2647    lin
2648 -    DetCN det cn = {
2649 -      s = \\c => det.s ! cn.g ! c ++ cn.s ! det.n ! c ;
2650 -      n = det.n ; g = cn.g ; p = P3
2651 +    DetCN det cn = -- Det -> CN -> NP
2652 +      {
2653 +       s = \\c => det.s ! cn.g ! c ++ cn.preap.s ! (Ag cn.g det.n c) ++ cn.s ! det.n ! c ++ cn.postap.s ! (Ag cn.g det.n c) ;
2654 +       n = det.n ; g = cn.g ; p = P3 ;
2655        } ;
2656
2657 --    UsePN pn = pn ** {a = agrgP3 Sg pn.g} ;
2658 -    UsePron p = p ;
2659 +    UsePN pn = lin NP { s = pn.s ! Sg ; g = pn.g ; n = Sg ; p = P3 } ;
2660 +
2661 +    UsePron p = -- Pron -> Np
2662 +      {
2663 +       g = p.g ;
2664 +       n = p.n ;
2665 +       p = p.p ;
2666 +       s = \\c => case c of {
2667 +         Nom => p.pers ! PronDrop ! PronRefl ; -- Drop pronoun in nominative case
2668 +         _ => p.pers ! PronNonDrop ! PronRefl  -- but don't drop it otherwise
2669 +         } ! c ;
2670 +      } ;
2671
2672 --    PredetNP pred np = {
2673 --      s = \\c => pred.s ++ np.s ! c ;
2674 --      a = np.a
2675 --      } ;
2676 --
2677 --    PPartNP np v2 = {
2678 --      s = \\c => np.s ! c ++ v2.s ! VPPart ;
2679 --      a = np.a
2680 --      } ;
2681 --
2682 --    RelNP np rs = {
2683 --      s = \\c => np.s ! c ++ "," ++ rs.s ! np.a ;
2684 --      a = np.a
2685 --      } ;
2686 --
2687 --    AdvNP np adv = {
2688 --      s = \\c => np.s ! c ++ adv.s ;
2689 --      a = np.a
2690 --      } ;
2691 --
2692 --    DetQuantOrd quant num ord = {
2693 --      s  = quant.s ! num.hasCard ! num.n ++ num.s ++ ord.s ;
2694 --      sp = quant.sp ! num.hasCard ! num.n ++ num.s ++ ord.s ;
2695 --      n  = num.n
2696 --      } ;
2697 --
```

```
2698         DetQuant quant num = {
2699 -         s  = \\g,c => quant.s  ! num.n ! g ! c ++ num.s ! g ! c ;
2700 -         sp = \\g,c => quant.sp ! num.n ! g ! c ++ num.s ! g ! c ;
2701 +         s  = \\g,c => quant.s  ! Ag g num.n c ++ num.s ! g ! c ;
2702 +         sp = \\g,c => quant.sp ! Ag g num.n c ++ num.s ! g ! c ;
2703           n  = num.n
2704           } ;
2705
2706 -       DetNP det = {
2707 -         s = det.sp ! Neutr ;
2708 -         g = Neutr ; n = det.n ; p = P3
2709 -         } ;
2710 +       -- DetNP det = {
2711 +       --   s = det.sp ! Neutr ;
2712 +       --   g = Neutr ; n = det.n ; p = P3
2713 +       --   } ;
2714
2715 --      PossPron p = {
2716 --        s = \\_,_ => p.s ! Gen ;
2717 --        sp = \\_,_ => p.sp
2718 --        } ;
2719 --
2720         NumSg = {s = \\_,_ => [] ; n = Sg} ;
2721         NumPl = {s = \\_,_ => [] ; n = Pl} ;
2722
2723 --      NumCard n = n ** {hasCard = True} ;
2724 --
2725 --      NumDigits n = {s = n.s ! NCard ; n = n.n} ;
2726 --      OrdDigits n = {s = n.s ! NOrd} ;
2727 --
2728 --      NumNumeral numeral = {s = numeral.s ! NCard; n = numeral.n} ;
2729 --      OrdNumeral numeral = {s = numeral.s ! NOrd} ;
2730 --
2731 --      AdNum adn num = {s = adn.s ++ num.s ; n = num.n} ;
2732 --
2733 --      OrdSuperl a = {s = a.s ! AAdj Superl} ;
2734
2735         DefArt = {
2736 -         s = \\_,_,_ => [] ;
2737 -         sp = \\n,g => (personalPronoun g n P3).s
2738 +         s = \\_ => [] ;
2739 +         sp = \\_ => [] ;
2740           } ;
2741
2742 --      IndefArt = {
2743 --        s = \\c,n => case <n,c> of {
2744 --          <Sg, False> => artIndef ;
2745 --          _ => []
2746 --          } ;
2747 --        sp = \\c,n => case <n,c> of {
2748 --          <Sg, False> => "one" ;
2749 --          <Pl, False> => "ones" ;
2750 --          _ => []
2751 --          }
2752 --        } ;
2753 --
2754 --      MassNP cn = {
2755 --        s = cn.s ! Sg ;
2756 --        a = agrP3 Sg
2757 --        } ;
2758 --
2759 -       UseN n = n ;
2760 --      UseN2 n = n ;
2761 ------b     UseN3 n = n ;
2762 +       IndefArt = {
2763 +         s = \\_ => [] ;
2764 +         sp = \\_ => [] ;
2765 +         } ;
2766 +
2767 +       -- MassNP cn = {
2768 +       --   s = cn.s ! Sg ;
2769 +       --   a = Ag cn.g Sg
2770 +       --   } ;
2771 +
2772 +--2 Common Nouns
```

```
2773 +     UseN n = -- N -> CN
2774 +   lin CN ( n ** {preap, postap = {s = \\_ => "" } } ) ;
2775 +
2776 +       UseN2 n2 = -- N2 -> CN
2777 +   lin CN ( n2 ** {preap, postap = {s = \\_ => "" } } ) ;
2778 + ----b      UseN3 n = n ;
2779  --
2780  --     Use2N3 f = {
2781  --       s = \\n,c => f.s ! n ! Nom ;
2782  --       g = f.g ;
2783  --       c2 = f.c2
2784  --       } ;
2785  --
2786  --     Use3N3 f = {
2787  --       s = \\n,c => f.s ! n ! Nom ;
2788  --       g = f.g ;
2789  --       c2 = f.c3
2790  --       } ;
2791  --
2792  --     ComplN2 f x = {s = \\n,c => f.s ! n ! Nom ++ f.c2 ++ x.s ! c ; g = f.g} ;
2793  --     ComplN3 f x = {
2794  --       s = \\n,c => f.s ! n ! Nom ++ f.c2 ++ x.s ! c ;
2795  --       g = f.g ;
2796  --       c2 = f.c3
2797  --       } ;
2798
2799 -     AdjCN ap cn = {
2800 -       s = \\n,c => preOrPost ap.isPre (ap.s ! cn.g ! n ! c) (cn.s ! n ! c) ;
2801 -       g = cn.g
2802 +   param
2803 +     AdjPos = Pre | Post ;
2804 +   lin
2805 +     AdjCN ap cn =  -- AP -> CN -> CN
2806 +       let pos = variants { Post ; Pre }
2807 +       in
2808 +       {
2809 +        -- s = \\n,c => preOrPost ap.isPre (ap.s ! cn.g ! n ! c) (cn.s ! n ! c) ;
2810 +        -- s = \\n,c => ( cn.s ! n ! c ) ++ ( ap.s ! AdjPhr cn.g n c) ; -- always add adjectives after noun?
2811 +         s = cn.s ;
2812 +         postap = case pos of { Pre => cn.postap ; Post => { s = \\a => ap.s ! a ++ cn.postap.s ! a } } ;
2813 +         preap = case pos of { Pre => { s = \\a => ap.s ! a ++ cn.preap.s ! a } ; Post => cn.preap } ;
2814 +        -- variants { postap = ConsAP postap ap ; preap = ConsAP preap ap } ; -- Nice if that would work
2815 +         g = cn.g
2816          } ;
2817
2818  --     RelCN cn rs = {
2819  --       s = \\n,c => cn.s ! n ! c ++ rs.s ! agrgP3 n cn.g ;
2820  --       g = cn.g
2821  --       } ;
2822
2823 -     AdvCN cn ad = {s = \\n,c => cn.s ! n ! c ++ ad.s ; g = cn.g} ;
2824 +--     AdvCN cn ad = {s = \\n,c => cn.s ! n ! c ++ ad.s ; g = cn.g} ;
2825
2826  --     SentCN cn sc = {s = \\n,c => cn.s ! n ! c ++ sc.s ; g = cn.g} ;
2827  --
2828  --     ApposCN cn np = {s = \\n,c => cn.s ! n ! Nom ++ np.s ! c ; g = cn.g} ;
2829  --
2830  }
2831 diff --git a/lib/src/latin/ParadigmsLat.gf b/lib/src/latin/ParadigmsLat.gf
2832 index 808d46b..a45cd4a 100644
2833 --- a/lib/src/latin/ParadigmsLat.gf
2834 +++ b/lib/src/latin/ParadigmsLat.gf
2835 @@ -1,65 +1,102 @@
2836 --#-path=.:../abstract:../../prelude:../common
2837 +--#-path=.:../abstract:../prelude:../common
2838
2839  --1 Latin Lexical Paradigms
2840  --
2841 -- Aarne Ranta 2008
2842 +-- Aarne Ranta 2008, Extended Herbert Lange 2013
2843  --
2844  -- This is an API for the user of the resource grammar
2845  -- for adding lexical items. It gives functions for forming
2846  -- expressions of open categories: nouns, adjectives, verbs.
2847  --
```

```
2848  -- Closed categories (determiners, pronouns, conjunctions) are
2849  -- accessed through the resource syntax API, $Structural.gf$.
2850
2851  resource ParadigmsLat = open
2852    (Predef=Predef),
2853    Prelude,
2854    ResLat,
2855 +  MorphoLat,
2856    CatLat
2857    in {
2858
2859  --2 Parameters
2860  --
2861  -- To abstract over gender names, we define the following identifiers.
2862
2863  oper
2864    masculine : Gender ;
2865    feminine  : Gender ;
2866    neuter    : Gender ;
2867
2868    mkN = overload {
2869      mkN : (verbum : Str) -> N
2870 -      = \n -> noun n ** {lock_N = <>} ;
2871 +      = \n -> lin N ( noun n ) ;
2872      mkN : (verbum, verbi : Str) -> Gender -> N
2873 -      = \x,y,z -> noun_ngg x y z ** {lock_N = <>} ;
2874 +      = \x,y,z -> lin N ( noun_ngg x y z ) ;
2875    } ;
2876
2877    mkA = overload {
2878      mkA : (verbum : Str) -> A
2879 -      = \n -> adj n ** {isPre = False ; lock_A = <>} ;
2880 +      = \n -> lin A ( adj n ** {isPre = False } ) ;
2881      mkA : (verbum, verbi : Str) -> A
2882 -      = \x,y -> adj123 x y ** {isPre = False ; lock_A = <>} ;
2883 +      = \x,y -> lin A ( adj123 x y ** {isPre = False } ) ;
2884      mkA : (bonus,bona,bonum : N) -> A
2885 -      = \x,y,z -> mkAdjective x y z ** {isPre = False ; lock_A = <>} ;
2886 +      = \x,y,z ->
2887 +      let compsup = comp_super x ;
2888 +          advs : Str * Str =
2889 +            case x.s!Sg!Nom of {
2890 +              -- Bayer-Lindauer 50 4
2891 +              idon + #vowel + "us" => < "magis" , "maxime" > ;
2892 +              _ => < "" , "" >
2893 +            };
2894 +      in
2895 +      lin A ( mkAdjective x y z < compsup.p1 , advs.p2 > < compsup.p2 , advs.p2> ** {isPre = False } ) ;
2896    } ;
2897
2898
2899    mkV = overload {
2900      mkV : (tacere : Str) -> V
2901 -      = \v -> verb v ** {lock_V = <>} ;
2902 -    mkV : (iacio,ieci,iactus,iacere : Str) -> V
2903 -      = \v,x,y,z -> verb_pppi v x y z ** {lock_V = <>} ;
2904 +      = \v -> lin V ( verb v ) ;
2905 +    mkV : (iacere,iacio,ieci,iactus : Str) -> V
2906 +      = \v,x,y,z -> lin V ( verb_ippp v x y z ) ;
2907 +    mkV : (iacere,iacio,ieci : Str) -> V
2908 +      = \v,x,y -> lin V ( verb_ippp v x y "######" ) ;
2909    } ;
2910
2911 +  V0 : Type = V ;
2912 +  mkV0 : V -> V0 = \v -> lin V0 v ; -- Same as in english, don't know if it's working
2913 +
2914    mkV2 = overload {
2915      mkV2 : (amare : Str) -> V2
2916 -      = \v -> verb v ** {c = {s = [] ; c = Acc} ; lock_V2 = <>} ;
2917 +      = \v -> lin V2 ( verb v ** { c = lin Prep ( mkPrep "" Acc ) } ) ;
2918      mkV2 : (facere : V) -> V2
2919 -      = \v -> v ** {c = {s = [] ; c = Acc} ; lock_V2 = <>} ;
2920 +      = \v -> lin V2 ( v ** { c = lin Prep ( mkPrep "" Acc ) } ) ;
2921 +    mkV2 : V -> Prep -> V2
2922 +      = \v,p -> lin V2 ( v ** { c = p } ) ;
```

```
2923        } ;
2924 ---.
2925 +
2926     masculine = Masc ;
2927     feminine = Fem ;
2928     neuter = Neutr ;
2929
2930 +-- To be implemented, just place holders
2931 +  mkPN : N -> PN = \n -> lin PN n ;
2932 +  mkN2 : N -> Prep -> N2 = \n,p -> lin N2 ( n ** { c = p } );
2933 +  mkN3 : N -> Prep -> Prep -> N3 = \n,p1,p2 -> lin N3 ( n **{ c = p1 ; c2 = p2 } ) ;
2934 +  mkV2S : V -> Prep -> V2S = \v,p -> lin V2S ( v ** { c = p } ) ;
2935 +  mkV2Q : V -> Prep -> V2Q = \v,p -> lin V2Q ( v ** { c = p } ) ;
2936 +  mkV2V : V -> Str -> Bool -> V2V = \v,s,b -> lin V2V ( v ** { c2 = s ; isAux = b } ) ;
2937 +  mkVV : V -> Bool -> VV = \v,b -> lin VV ( v ** { isAux = b } ) ;
2938 +  mkVA : V -> VA = \v -> lin VA v ;
2939 +  mkV3 : V -> Prep -> Prep -> V3 = \v,p1,p2 -> lin V3 ( v ** { c2 = p1; c3 = p2 } ) ;
2940 +  mkVQ : V -> VQ = \v -> lin VQ v ;
2941 +  mkVS : V -> VS = \v -> lin VS v ;
2942 +  mkV2A : V -> Prep -> V2A = \v,p -> lin V2A (v ** { c = p } ) ;
2943 +  AS : Type = A ;
2944 +  mkAS : A -> AS = \a -> lin AS a ;
2945 +  mkA2 : A -> Prep -> A2 = \a,p -> lin A2 ( a ** { c = p } ) ;
2946 +  A2V : Type = A2 ;
2947 +  mkA2V : A -> Prep -> A2V = \a,p -> lin A2V ( lin A2 ( a ** { c = p } ) ) ;
2948 +  AV : Type = A ;
2949 +  mkAV : A -> AV = \a -> lin AV a ;
2950  }
2951 diff --git a/lib/src/latin/PhraseLat.gf b/lib/src/latin/PhraseLat.gf
2952 index 1c92d44..1c699a9 100644
2953 --- a/lib/src/latin/PhraseLat.gf
2954 +++ b/lib/src/latin/PhraseLat.gf
2955 @@ -1,24 +1,24 @@
2956 ---concrete PhraseLat of Phrase = CatLat ** open Prelude, ResLat in {
2957 ---
2958 ---  lin
2959 ---    PhrUtt pconj utt voc = {s = pconj.s ++ utt.s ++ voc.s} ;
2960 ---
2961 ---    UttS s = s ;
2962 ---    UttQS qs = {s = qs.s ! QDir} ;
2963 +concrete PhraseLat of Phrase = CatLat ** open Prelude, ResLat in {
2964 +
2965 +  lin
2966 +    PhrUtt pconj utt voc = {s = pconj.s ++ utt.s ++ voc.s} ;
2967 +    --
2968 +    UttS s = lin Utt s ; -- S -> Utt
2969 +    UttQS qs = {s = qs.s ! QDir} ;
2970  --    UttImpSg pol imp = {s = pol.s ++ imp.s ! contrNeg True pol.p ! ImpF Sg False} ;
2971  --    UttImpPl pol imp = {s = pol.s ++ imp.s ! contrNeg True pol.p ! ImpF Pl False} ;
2972  --    UttImpPol pol imp = {s = pol.s ++ imp.s ! contrNeg True pol.p ! ImpF Sg True} ;
2973  --
2974  --    UttIP ip = {s = ip.s ! Nom} ; --- Acc also
2975  --    UttIAdv iadv = iadv ;
2976  --    UttNP np = {s = np.s ! Nom} ;
2977  --    UttVP vp = {s = infVP False vp (agrP3 Sg)} ;
2978  --    UttAdv adv = adv ;
2979  --
2980 ---    NoPConj = {s = []} ;
2981 ---    PConjConj conj = {s = conj.s2} ; ---
2982 +    NoPConj = {s = []} ;
2983 +    PConjConj conj = {s = conj.s2} ; ---
2984  --
2985 ---    NoVoc = {s = []} ;
2986 ---    VocNP np = {s = "," ++ np.s ! Nom} ;
2987 +    NoVoc = {s = []} ;
2988 +    VocNP np = {s = "," ++ np.s ! Voc} ;
2989  --
2990 ---}
2991 +}
2992 diff --git a/lib/src/latin/ResLat.gf b/lib/src/latin/ResLat.gf
2993 index 767c599..e039329 100644
2994 --- a/lib/src/latin/ResLat.gf
2995 +++ b/lib/src/latin/ResLat.gf
2996 @@ -1,573 +1,959 @@
2997 ---#-path=.:../abstract:../common:../../prelude
```

```
2998 +--#-path=.:../abstract:../common:../prelude
2999
3000 ---1 Latlish auxiliary operations.
3001 +--1 Latin auxiliary operations.
3002
3003 -resource ResLat = ParamX ** open Prelude in {
3004 +resource ResLat = ParamX ** open Prelude,TenseX in {
3005
3006  param
3007 -  Gender = Masc | Fem | Neutr ;
3008    Case = Nom | Acc | Gen | Dat | Abl | Voc ;
3009 +  Gender = Masc | Fem | Neutr ;
3010  ---  Degree = DPos | DComp | DSup ;
3011
3012 -oper
3013 -  Noun : Type = {s : Number => Case => Str ; g : Gender} ;
3014 -  Adjective : Type = {s : Gender => Number => Case => Str} ;
3015 -
3016 -  -- worst case
3017 -
3018 -  mkNoun : (n1,_,_,_,_,_,_,_,_,n10 : Str) -> Gender -> Noun =
3019 -    \sn,sa,sg,sd,sab,sv,pn,pa,pg,pd, g -> {
3020 -    s = table {
3021 -      Sg => table {
3022 -        Nom => sn ;
3023 -        Acc => sa ;
3024 -        Gen => sg ;
3025 -        Dat => sd ;
3026 -        Abl => sab ;
3027 -        Voc => sv
3028 -        } ;
3029 -      Pl => table {
3030 -        Nom | Voc => pn ;
3031 -        Acc => pa ;
3032 -        Gen => pg ;
3033 -        Dat | Abl => pd
3034 -        }
3035 -      } ;
3036 -    g = g
3037 -    } ;
3038 -
3039 -  -- declensions
3040 -
3041 -  noun1 : Str -> Noun = \mensa ->
3042 -    let
3043 -      mensae = mensa + "e" ;
3044 -      mensis = init mensa + "is" ;
3045 -    in
3046 -    mkNoun
3047 -      mensa (mensa +"m") mensae mensae mensa mensa
3048 -      mensae (mensa + "s") (mensa + "rum") mensis
3049 -      Fem ;
3050 -
3051 -  noun2us : Str -> Noun = \servus ->
3052 -    let
3053 -      serv = Predef.tk 2 servus ;
3054 -      servum = serv + "um" ;
3055 -      servi = serv + "i" ;
3056 -      servo = serv + "o" ;
3057 -    in
3058 -    mkNoun
3059 -      servus servum servi servo servo (serv + "e")
3060 -      servi (serv + "os") (serv + "orum") (serv + "is")
3061 -      Masc ;
3062 -
3063 -  noun2er : Str -> Noun = \puer ->
3064 -    let
3065 -      puerum = puer + "um" ;
3066 -      pueri = puer + "i" ;
3067 -      puero = puer + "o" ;
3068 -    in
3069 -    mkNoun
3070 -      puer puerum pueri puero puero (puer + "e")
3071 -      pueri (puer + "os") (puer + "orum") (puer + "is")
3072 -      Masc ;
```

```
3073 -
3074 -  noun2um : Str -> Noun = \bellum ->
3075 -    let
3076 -      bell = Predef.tk 2 bellum ;
3077 -      belli = bell + "i" ;
3078 -      bello = bell + "o" ;
3079 -      bella = bell + "a" ;
3080 -    in
3081 -    mkNoun
3082 -      bellum bellum belli bello bello (bell + "e")
3083 -      bella bella (bell + "orum") (bell + "is")
3084 -      Neutr ;
3085 -
3086 --- smart paradigm for declensions 1&2
3087 -
3088 -  noun12 : Str -> Noun = \verbum ->
3089 -    case verbum of {
3090 -      _ + "a"  => noun1 verbum ;
3091 -      _ + "us" => noun2us verbum ;
3092 -      _ + "um" => noun2um verbum ;
3093 -      _ + "er" => noun2er verbum ;
3094 -      _  => Predef.error ("noun12 does not apply to" ++ verbum)
3095 + oper
3096 +    consonant : pattern Str = #( "p" | "b" | "f" | "v" | "m" | "t" | "d" | "s" | "z" | "n" | "r" | "c" | "g" | "l" | "q" | "qu" | "h" );
3097 +
3098 +    Noun : Type = {s : Number => Case => Str ; g : Gender} ;
3099 +    NounPhrase : Type =
3100 +      {
3101 +       s : Case => Str ;
3102 +       g : Gender ;
3103 +       n : Number ;
3104 +       p : Person ;
3105        } ;
3106 -
3107 -  noun3c : Str -> Str -> Gender -> Noun = \rex,regis,g ->
3108 -    let
3109 -      reg = Predef.tk 2 regis ;
3110 -      rege : Str = case rex of {
3111 -        _ + "e" => reg + "i" ;
3112 -        _ + ("al" | "ar") => rex + "i" ;
3113 -        _ => reg + "e"
3114 -        } ;
3115 -      regemes : Str * Str = case g of {
3116 -        Neutr => <rex,reg + "a"> ;
3117 -        _     => <reg + "em", reg + "es">
3118 -        } ;
3119 -    in
3120 -    mkNoun
3121 -      rex regemes.p1 (reg + "is") (reg + "i") rege rex
3122 -      regemes.p2 regemes.p2 (reg + "um") (reg + "ibus")
3123 -      g ;
3124 -
3125 -
3126 -  noun3 : Str -> Noun = \labor ->
3127 -    case labor of {
3128 -      _    + "r"  => noun3c labor (labor + "is")    Masc ;
3129 -      fl   + "os" => noun3c labor (fl    + "oris")  Masc ;
3130 -      lim  + "es" => noun3c labor (lim   + "itis")  Masc ;
3131 -      cod  + "ex" => noun3c labor (cod   + "icis")  Masc ;
3132 -      poem + "a"  => noun3c labor (poem  + "atis")  Neutr ;
3133 -      calc + "ar" => noun3c labor (calc  + "aris")  Neutr ;
3134 -      mar  + "e"  => noun3c labor (mar   + "is")    Neutr ;
3135 -      car  + "men"=> noun3c labor (car   + "minis") Neutr ;
3136 -      rob  + "ur" => noun3c labor (rob   + "oris")  Neutr ;
3137 -      temp + "us" => noun3c labor (temp  + "oris")  Neutr ;
3138 -      vers + "io" => noun3c labor (vers  + "ionis") Fem ;
3139 -      imag + "o"  => noun3c labor (imag  + "inis")  Fem ;
3140 -      ae   + "tas"=> noun3c labor (ae    + "tatis") Fem ;
3141 -      vo   + "x"  => noun3c labor (vo    + "cis")   Fem ;
3142 -      pa   + "rs" => noun3c labor (pa    + "rtis")  Fem ;
3143 -      cut  + "is" => noun3c labor (cut   + "is")    Fem ;
3144 -      urb  + "s"  => noun3c labor (urb   + "is")    Fem ;
3145 -      _  => Predef.error ("noun3 does not apply to" ++ labor)
3146 + param
3147 +    Order = SVO | VSO | VOS | OSV | OVS | SOV ;
```

```
3148 +  param
3149 +    Agr = Ag Gender Number Case ; -- Agreement for NP et al.
3150 +  oper
3151 +    Adjective : Type = {
3152 +      s : Degree => Agr => Str ;
3153 +--      comp_adv : Str ;
3154 +--      super_adv : Str
3155        } ;
3156 +    CommonNoun : Type =
3157 +    {
3158 +      s : Number => Case => Str ;
3159 +      g : Gender ;
3160 +      preap : {s : Agr => Str } ;
3161 +      postap : {s : Agr => Str } ;
3162 +    } ;
3163 +-- nouns
3164 +  useCNasN : CommonNoun -> Noun = \cn ->
3165 +    {
3166 +      s = cn.s ;
3167 +      g = cn.g ;
3168 +    } ;
3169
3170 -  noun4us : Str -> Noun = \fructus ->
3171 -    let
3172 -      fructu = init fructus ;
3173 -      fruct  = init fructu
3174 -    in
3175 -    mkNoun
3176 -      fructus (fructu + "m") fructus (fructu + "i") fructu fructus
3177 -      fructus fructus (fructu + "um") (fruct + "ibus")
3178 -      Masc ;
3179 -
3180 -  noun4u : Str -> Noun = \cornu ->
3181 -    let
3182 -      corn = init cornu ;
3183 -      cornua = cornu + "a"
3184 -    in
3185 -    mkNoun
3186 -      cornu cornu (cornu + "s") (cornu + "i") cornu cornu
3187 -      cornua cornua (cornu + "um") (corn + "ibus")
3188 -      Neutr ;
3189 -
3190 -  noun5 : Str -> Noun = \res ->
3191 -    let
3192 -      re = init res ;
3193 -      rei = re + "i"
3194 -    in
3195 -    mkNoun
3196 -      res (re+ "m") rei rei re res
3197 -      res res (re + "rum") (re + "bus")
3198 -      Fem ;
3199 -
3200 +  pluralN : Noun -> Noun = \n ->
3201 +    {
3202 +      s = table {
3203 +        Pl => n.s ! Pl ;
3204 +        Sg => \\_ => "#######" -- no singular forms
3205 +        };
3206 +      g = n.g ;
3207 +      preap = n.preap ;
3208 +      postap = n.postap ;
3209 +    };
3210 +
3211 +  mkNoun : (n1,_,_,_,_,_,_,_,_,n10 : Str) -> Gender -> Noun =
3212 +    \sn,sa,sg,sd,sab,sv,pn,pa,pg,pd,g -> {
3213 +      s = table {
3214 +        Sg => table {
3215 +          Nom => sn ;
3216 +          Acc => sa ;
3217 +          Gen => sg ;
3218 +          Dat => sd ;
3219 +          Abl => sab ;
3220 +          Voc => sv
3221 +          } ;
3222 +        Pl => table {
```

```
3223 +            Nom | Voc => pn ;
3224 +            Acc => pa ;
3225 +            Gen => pg ;
3226 +            Dat | Abl => pd
3227 +            }
3228 +        } ;
3229 +      g = g
3230 +    } ;
3231 +
3232 +  -- to change the default gender
3233
3234 -    nounWithGen : Gender -> Noun -> Noun = \g,n ->
3235 -      {s = n.s ; g = g} ;
3236 -
3237 -- smart paradigms
3238 -
3239 -  noun_ngg : Str -> Str -> Gender -> Noun = \verbum,verbi,g ->
3240 -    let s : Noun = case <verbum,verbi> of {
3241 -      <_ + "a", _ + "ae"> => noun1 verbum ;
3242 -      <_ + "us", _ + "i"> => noun2us verbum ;
3243 -      <_ + "um", _ + "i"> => noun2um verbum ;
3244 -      <_ + "er", _ + "i"> => noun2er verbum ;
3245 -      <_ + "us", _ + "us"> => noun4us verbum ;
3246 -      <_ + "u", _ + "us"> => noun4u verbum ;
3247 -      <_ + "es", _ + "ei"> => noun5 verbum ;
3248 -      _ => noun3c verbum verbi g
3249 -      }
3250 -    in
3251 -    nounWithGen g s ;
3252 -
3253 -  noun : Str -> Noun = \verbum ->
3254 -    case verbum of {
3255 -      _ + "a" => noun1 verbum ;
3256 -      _ + "us" => noun2us verbum ;
3257 -      _ + "um" => noun2um verbum ;
3258 -      _ + "er" => noun2er verbum ;
3259 -      _ + "u" => noun4u verbum ;
3260 -      _ + "es" => noun5 verbum ;
3261 -      _ => noun3 verbum
3262 -      } ;
3263 -
3264 +  nounWithGen : Gender -> Noun -> Noun = \g,n ->
3265 +    {s = n.s ; g = g} ;
3266
3267 +  regNP : (_,_,_,_,_,_ : Str) -> Gender -> Number -> NounPhrase =
3268 +    \nom,acc,gen,dat,abl,voc,g,n ->
3269 +    {
3270 +      s = table Case [ nom ; acc ; gen ; dat ; abl ; voc ] ;
3271 +      g = g ;
3272 +      n = n ;
3273 +      p = P3
3274 +    } ;
3275 +-- also used for adjectives and so on
3276
3277  -- adjectives
3278
3279 -  mkAdjective : (_,_,_ : Noun) -> Adjective = \bonus,bona,bonum -> {
3280 -    s = table {
3281 -      Masc  => bonus.s ;
3282 -      Fem   => bona.s ;
3283 -      Neutr => bonum.s
3284 -      }
3285 +  mkAdjective : (_,_,_ : Noun) ->
3286 +    ( (Agr => Str) * Str ) ->
3287 +    ( (Agr => Str) * Str ) -> Adjective =
3288 +    \bonus,bona,bonum,melior,optimus ->
3289 +    {
3290 +      s = table {
3291 +       Posit => table {
3292 +         Ag Masc  n c => bonus.s ! n ! c ;
3293 +         Ag Fem   n c => bona.s  ! n ! c ;
3294 +         Ag Neutr n c => bonum.s ! n ! c
3295 +         } ;
3296 +       Compar => melior.p1 ;
3297 +       Superl => optimus.p1
```

```
3298 +        } ;
3299 +      comp_adv = melior.p2 ;
3300 +      super_adv = optimus.p2
3301       } ;
3302 -
3303 -  adj12 : Str -> Adjective = \bonus ->
3304 -    let
3305 -      bon : Str = case bonus of {
3306 -       pulch + "er" => pulch + "r" ;
3307 -       bon + "us" => bon ;
3308 -       _ => Predef.error ("adj12 does not apply to" ++ bonus)
3309 -       }
3310 -    in
3311 -    mkAdjective (noun12 bonus) (noun1 (bon + "a")) (noun2um (bon + "um")) ;
3312 -
3313 -  adj3x : (_,_ : Str) -> Adjective = \acer,acris ->
3314 -   let
3315 -     ac = Predef.tk 2 acer ;
3316 -     acrise : Str * Str = case acer of {
3317 -       _ + "er" => <ac + "ris", ac + "re"> ;
3318 -       _ + "is" => <acer      , ac + "e"> ;
3319 -       _        => <acer      , acer>
3320 -       }
3321 -   in
3322 -   mkAdjective
3323 -     (noun3adj acer acris Masc)
3324 -     (noun3adj acrise.p1 acris Fem)
3325 -     (noun3adj acrise.p2 acris Neutr) ;
3326 +
3327
3328     noun3adj : Str -> Str -> Gender -> Noun = \audax,audacis,g ->
3329       let
3330         audac   = Predef.tk 2 audacis ;
3331         audacem = case g of {Neutr => audax ; _ => audac + "em"} ;
3332         audaces = case g of {Neutr => audac +"ia" ; _ => audac + "es"} ;
3333         audaci  = audac + "i" ;
3334       in
3335       mkNoun
3336         audax audacem (audac + "is") audaci audaci audax
3337         audaces audaces (audac + "ium") (audac + "ibus")
3338         g ;
3339
3340
3341 --- smart paradigm
3342 -
3343 -  adj123 : Str -> Str -> Adjective = \bonus,boni ->
3344 -    case <bonus,boni> of {
3345 -      <_ + ("us" | "er"), _ + "i">  => adj12 bonus ;
3346 -      <_ + ("us" | "er"), _ + "is"> => adj3x bonus boni ;
3347 -      <_                , _ + "is"> => adj3x bonus boni ;
3348 -      _ => Predef.error ("adj123: not applicable to" ++ bonus ++ boni)
3349 -    } ;
3350 -
3351 -  adj : Str -> Adjective = \bonus ->
3352 -    case bonus of {
3353 -      _ + ("us" | "er") => adj12 bonus ;
3354 -      facil + "is"      => adj3x bonus bonus ;
3355 -      feli  + "x"       => adj3x bonus (feli + "cis") ;
3356 -      _                 => adj3x bonus (bonus + "is") --- any example?
3357 -    } ;
3358 -
3359 +  emptyAdj : Adjective =
3360 +    { s = \\_,_ => "" ; comp_adv = "" ; super_adv = "" } ;
3361
3362  --- verbs
3363
3364 -  param
3365 +param
3366     VActForm  = VAct VAnter VTense Number Person ;
3367 -  VPassForm = VPass VTense Number Person ;
3368 -  VInfForm  = VInfActPres | VInfActPerf ;
3369 -  VImpForm  = VImpPres Number | VImpFut2 Number | VImpFut3 Number ;
3370 +  VPassForm = VPass VTense Number Person ; --- No anteriority because perfect forms are built using participle
3371 +  VInfForm  = VInfActPres | VInfActPerf Gender | VInfActFut Gender | VInfPassPres | VInfPassPerf Gender | VinfPassFut ;
3372 +  VImpForm  = VImp1 Number | VImp2 Number Person ;
```

```
3373      VGerund    = VGenAcc | VGenGen |VGenDat | VGenAbl ;
3374      VSupine    = VSupAcc | VSupAbl ;
3375  +   VPartForm = VActPres | VActFut | VPassPerf ;
3376
3377  -   VAnter = VSim | VAnt ;
3378  +   VAnter = VAnt | VSim ;
3379      VTense = VPres VMood | VImpf VMood | VFut ;
3380      VMood  = VInd | VConj ;
3381
3382      oper
3383  +   VerbPhrase : Type = {
3384  +     fin : VActForm => Str ;
3385  +     inf : VInfForm => Str ;
3386  +     obj : Str ;
3387  +     adj : Agr => Str
3388  +   } ;
3389  +
3390      Verb : Type = {
3391  -     act  : VActForm => Str ;
3392  ---     pass : VPassForm => Str ;
3393  -     inf  : VAnter => Str ;
3394  ---     imp  : VImpForm => Str ;
3395  ---     ger  : VGerund => Str ;
3396  ---     sup  : VSupine => Str ;
3397  ---     partActPres  : Adjective ;
3398  ---     partActFut   : Adjective ;
3399  ---     partPassPerf : Adjective ;
3400  ---     partPassFut  : Adjective ;
3401  +     act    : VActForm => Str ;
3402  +     pass   : VPassForm => Str ;
3403  +     inf    : VInfForm => Str ;
3404  +     imp    : VImpForm => Str ;
3405  +     ger    : VGerund => Str ;
3406  +     geriv  : Agr => Str ;
3407  +     sup    : VSupine => Str ;
3408  +     part   : VPartForm =>Agr => Str ;
3409  +     } ;
3410  +
3411  +   VV : Type = Verb ** { isAux : Bool } ;
3412  +
3413  +   tenseToVTense : Tense -> VTense =
3414  +     \t ->
3415  +     case t of
3416  +     {
3417  +       Pres => VPres VInd ;
3418  +       Past => VImpf VInd ;
3419  +       Fut => VFut ;
3420  +       Cond => VPres VConj --- don't know what to do
3421  +     } ;
3422  +
3423  +   anteriorityToVAnter : Anteriority -> VAnter =
3424  +     \a ->
3425  +     case a of
3426  +     {
3427  +       Simul => VSim ;
3428  +       Anter => VAnt
3429  +     } ;
3430  +
3431  +   useVV : VV -> Verb = \vv ->
3432  +     {
3433  +       act = vv.act ;
3434  +       pass = vv.pass ;
3435  +       inf = vv.inf ;
3436  +       imp = vv.imp ;
3437  +       ger = vv.ger ;
3438  +       geriv = vv.geriv ;
3439  +       sup = vv.sup ;
3440  +       part = vv.part ;
3441  +     } ;
3442  +
3443  +   useVPasV : VerbPhrase -> Verb = \vp ->
3444  +     {
3445  +       act = \\a => vp.obj ++ vp.fin ! a ;
3446  +       pass = \\_ => "#######" ;
3447  +       inf = \\a => vp.obj ++ vp.inf ! a ;
```

```
3448 +       imp = \\_ => "######" ;
3449 +       ger = \\_ => "######" ;
3450 +       geriv = \\_ => "######" ;
3451 +       sup = \\_ => "######" ;
3452 +       part = \\_,_ => "######" ;
3453       } ;
3454
3455     mkVerb :
3456 -     (cela,cele,celab,celo,celant,celare,celavi,celatus,celabo,celabunt,celabi : Str)
3457 -       -> Verb =
3458 -     \cela,cele,celab,celo,celant,celare,celavi,celatus,celabo,celabunt,celabi ->
3459 +     (regere,reg,regi,rega,regeba,regere,rege,regi,rex,rex,rexeri,rexera,rexisse,rexeri,rect : Str)
3460 +     -> Verb =
3461 +     \inf_act_pres,pres_stem,pres_ind_base,pres_conj_base,impf_ind_base,impf_conj_base,fut_I_base,imp_base,
3462 +     perf_stem,perf_ind_base,perf_conj_base,pqperf_ind_base,pqperf_conj_base,fut_II_base,part_stem ->
3463       let
3464 -       celav = init celavi
3465 -     in {
3466 -       act = table {
3467 -         VAct VSim (VPres VInd)  Sg P1 => celo ;
3468 -         VAct VSim (VPres VInd)  Pl P3 => celant ;
3469 -         VAct VSim (VPres VInd)  n  p  => cela + actPresEnding n p ;
3470 -         VAct VSim (VPres VConj) n  p  => cele + actPresEnding n p ;
3471 -         VAct VSim (VImpf VInd)  n  p  => celab + "ba" + actPresEnding n p ;
3472 -         VAct VSim (VImpf VConj) n  p  => celare + actPresEnding n p ;
3473 -         VAct VSim VFut          Sg P1 => celabo ;
3474 -         VAct VSim VFut          Pl P3 => celabunt ;
3475 -         VAct VSim VFut          n  p  => celabi + actPresEnding n p ;
3476 -         VAct VAnt (VPres VInd)  Pl P3 => celav + "erunt" ;
3477 -         VAct VAnt (VPres VInd)  n  p  => celavi + actPerfEnding n p ;
3478 -         VAct VAnt (VPres VConj) n  p  => celav + "eri" + actPresEnding n p ;
3479 -         VAct VAnt (VImpf VInd)  n  p  => celav + "era" + actPresEnding n p ;
3480 -         VAct VAnt (VImpf VConj) n  p  => celav + "isse" + actPresEnding n p ;
3481 -         VAct VAnt VFut          Sg P1 => celav + "ero" ;
3482 -         VAct VAnt VFut          n  p  => celav + "eri" + actPresEnding n p
3483 +       fill : Str * Str * Str = case pres_stem of {
3484 +       _ + ( "a" | "e" ) => < "" , "" , "" > ;
3485 +       _ + #consonant => < "e" , "u" , "i" > ;
3486 +       _ => < "e" , "u" , "" >
3487 +       } ;
3488 +     in
3489 +     {
3490 +       act =
3491 +        table {
3492 +           VAct VSim (VPres VInd)  Sg P1 => -- Present Indicative
3493 +            ( case pres_ind_base of {
3494 +                _ + "a" =>  ( init pres_ind_base ) ;
3495 +                _ => pres_ind_base
3496 +                }
3497 +            ) + "o" ; --actPresEnding Sg P1 ;
3498 +          VAct VSim (VPres VInd)  Pl P3 => -- Present Indicative
3499 +            pres_ind_base + fill.p2 + actPresEnding Pl P3 ;
3500 +           VAct VSim (VPres VInd)  n  p  => -- Present Indicative
3501 +            pres_ind_base + fill.p3 + actPresEnding n p ;
3502 +           VAct VSim (VPres VConj) n  p  => -- Present Conjunctive
3503 +            pres_conj_base + actPresEnding n p ;
3504 +           VAct VSim (VImpf VInd)  n  p  => -- Imperfect Indicative
3505 +            impf_ind_base + actPresEnding n p ;
3506 +           VAct VSim (VImpf VConj) n  p  => -- Imperfect Conjunctive
3507 +            impf_conj_base + actPresEnding n p ;
3508 +           VAct VSim VFut          Sg P1 => -- Future I
3509 +            case fut_I_base of {
3510 +              _ + "bi" => ( init fut_I_base ) + "o" ;
3511 +              _  => ( init fut_I_base ) + "a" + actPresEnding Sg P1
3512 +            } ;
3513 +          VAct VSim VFut          Pl P3 => -- Future I
3514 +            ( case fut_I_base of {
3515 +                _ + "bi" => ( init fut_I_base ) + "u";
3516 +                _ => fut_I_base
3517 +                }
3518 +            ) + actPresEnding Pl P3 ;
3519 +           VAct VSim VFut          n  p  => -- Future I
3520 +            fut_I_base + actPresEnding n p ;
3521 +           VAct VAnt (VPres VInd)  n  p  => -- Prefect Indicative
3522 +            perf_ind_base + actPerfEnding n p ;
```

```
3523 +          VAct VAnt (VPres VConj) n  p  => -- Prefect Conjunctive
3524 +           perf_conj_base + actPresEnding n p ;
3525 +          VAct VAnt (VImpf VInd)  n  p  => -- Plusperfect Indicative
3526 +           pqperf_ind_base + actPresEnding n p ;
3527 +          VAct VAnt (VImpf VConj) n  p  => -- Plusperfect Conjunctive
3528 +           pqperf_conj_base + actPresEnding n p ;
3529 +          VAct VAnt VFut          Sg P1 => -- Future II
3530 +           ( init fut_II_base ) + "o" ;
3531 +          VAct VAnt VFut          n  p  => -- Future II
3532 +           fut_II_base + actPresEnding n p
3533          } ;
3534 -      inf = table {
3535 -        VSim => celare ;
3536 -        VAnt => celav + "isse"
3537 -        }
3538 -      } ;
3539 +      pass =
3540 +       table {
3541 +         VPass (VPres VInd)  Sg P1 => -- Present Indicative
3542 +           ( case pres_ind_base of
3543 +               {
3544 +                 _ + "a" => (init pres_ind_base ) ;
3545 +                 _ => pres_ind_base
3546 +               }
3547 +           ) + "o" + passPresEnding Sg P1 ;
3548 +         VPass (VPres VInd)  Sg P2 => -- Present Indicative
3549 +           ( case imp_base of {
3550 +               _ + #consonant =>
3551 +                 ( case pres_ind_base of {
3552 +                     _ + "i" => ( init pres_ind_base ) ;
3553 +                     _ => pres_ind_base
3554 +                     }
3555 +                 ) + "e" ;
3556 +               _ => pres_ind_base
3557 +               }
3558 +           ) + passPresEnding Sg P2 ;
3559 +         VPass (VPres VInd)  Pl P3 => -- Present Indicative
3560 +           pres_ind_base + fill.p2 + passPresEnding Pl P3 ;
3561 +         VPass (VPres VInd)  n  p  => -- Present Indicative
3562 +           pres_ind_base + fill.p3 + passPresEnding n p ;
3563 +         VPass (VPres VConj) n  p  => -- Present Conjunctive
3564 +           pres_conj_base + passPresEnding n p ;
3565 +         VPass (VImpf VInd)  n  p  => -- Imperfect Indicative
3566 +           impf_ind_base + passPresEnding n p ;
3567 +         VPass (VImpf VConj) n  p  => -- Imperfect Conjunctive
3568 +           impf_conj_base + passPresEnding n p ;
3569 +         VPass VFut          Sg P1 => -- Future I
3570 +           ( case fut_I_base of {
3571 +               _ + "bi" => ( init fut_I_base ) + "o" ;
3572 +               _ => ( init fut_I_base ) + "a"
3573 +               }
3574 +           ) + passPresEnding Sg P1 ;
3575 +         VPass VFut          Sg P2 => -- Future I
3576 +           ( init fut_I_base ) + "e" + passPresEnding Sg P2 ;
3577 +         VPass VFut          Pl P3 => -- Future I
3578 +           ( case fut_I_base of {
3579 +               _ + "bi" => ( init fut_I_base ) + "u" ;
3580 +               _ => fut_I_base
3581 +               }
3582 +           ) + passPresEnding Pl P3 ;
3583 +         VPass VFut          n  p  => -- Future I
3584 +           fut_I_base + passPresEnding n p
3585 +       } ;
3586 +      inf =
3587 +       table {
3588 +          VInfActPres       => -- Infinitive Active Present
3589 +           inf_act_pres ;
3590 +          VInfActPerf _     => -- Infinitive Active Perfect
3591 +           perf_stem + "isse" ;
3592 +         VInfActFut Masc  => -- Infinitive Active Future
3593 +           part_stem + "urum" ;
3594 +         VInfActFut Fem   => -- Infinitive Active Future
3595 +           part_stem + "uram" ;
3596 +         VInfActFut Neutr => -- Infinitive Active Future
3597 +           part_stem + "urum" ;
```

```
3598 +         VInfPassPres        => -- Infinitive Present Passive
3599 +           ( init inf_act_pres ) + "i" ;
3600 +         VInfPassPerf Masc   => -- Infinitive Perfect Passive
3601 +           part_stem + "um" ;
3602 +         VInfPassPerf Fem    => -- Infinitive Perfect Passive
3603 +           part_stem + "am" ;
3604 +         VInfPassPerf Neutr  => -- Infinitive Perfect Passive
3605 +           part_stem + "um" ;
3606 +         VInfPassFut         => -- Infinitive Future Passive
3607 +           part_stem + "um"
3608 +       } ;
3609 +     imp =
3610 +       let
3611 +         imp_fill : Str * Str =
3612 +           case imp_base of {
3613 +             _ + #consonant => < "e" , "i" > ;
3614 +             _ => < "" , "" >
3615 +           };
3616 +         in
3617 +       table {
3618 +         VImp1 Sg            => -- Imperative I
3619 +           imp_base + imp_fill.p1 ;
3620 +         VImp1 Pl            => -- Imperative I
3621 +           imp_base + imp_fill.p2 + "te" ;
3622 +         VImp2 Sg ( P2 | P3 ) => -- Imperative II
3623 +           imp_base + imp_fill.p2 + "to" ;
3624 +         VImp2 Pl P2         => -- Imperative II
3625 +           imp_base + fill.p3 + "tote" ;
3626 +         VImp2 Pl P3         => -- Imperative II
3627 +           pres_stem + fill.p2 + "nto" ;
3628 +         _ => "######" -- No imperative form
3629 +       } ;
3630 +     ger =
3631 +       table {
3632 +         VGenAcc => -- Gerund
3633 +           pres_stem + fill.p1 + "ndum" ;
3634 +         VGenGen => -- Gerund
3635 +           pres_stem + fill.p1 + "ndi" ;
3636 +         VGenDat => -- Gerund
3637 +           pres_stem + fill.p1 + "ndo" ;
3638 +         VGenAbl => -- Gerund
3639 +           pres_stem + fill.p1 + "ndo"
3640 +       } ;
3641 +     geriv =
3642 +       ( mkAdjective
3643 +           ( mkNoun ( pres_stem + fill.p1 + "ndus" ) ( pres_stem + fill.p1 + "ndum" ) ( pres_stem + fill.p1 + "ndi" )
3644 +               ( pres_stem + fill.p1 + "ndo" ) ( pres_stem + fill.p1 + "ndo" ) ( pres_stem + fill.p1 + "nde" )
3645 +               ( pres_stem + fill.p1 + "ndi" ) ( pres_stem + fill.p1 + "ndos" ) ( pres_stem + fill.p1 + "ndorum" )
3646 +               ( pres_stem + fill.p1 + "ndis" )
3647 +               Masc )
3648 +           ( mkNoun ( pres_stem + fill.p1 + "nda" ) ( pres_stem + fill.p1 + "ndam" ) ( pres_stem + fill.p1 + "ndae" )
3649 +               ( pres_stem + fill.p1 + "ndae" ) ( pres_stem + fill.p1 + "nda" ) ( pres_stem + fill.p1 + "nda" )
3650 +               ( pres_stem + fill.p1 + "ndae" ) ( pres_stem + fill.p1 + "ndas" ) (pres_stem + fill.p1 +"ndarum" )
3651 +               ( pres_stem + fill.p1 + "ndis" )
3652 +               Fem )
3653 +           ( mkNoun ( pres_stem + fill.p1 + "ndum" ) ( pres_stem + fill.p1 + "ndum" ) ( pres_stem + fill.p1 + "ndi" )
3654 +               ( pres_stem + fill.p1 + "ndo" ) ( pres_stem + fill.p1 + "ndo" ) ( pres_stem + fill.p1 + "ndum" )
3655 +               ( pres_stem + fill.p1 + "nda" ) ( pres_stem + fill.p1 + "nda" ) ( pres_stem + fill.p1 + "ndorum" )
3656 +               ( pres_stem + fill.p1 + "ndis" )
3657 +               Neutr )
3658 +           < \\_ => "" , "" >
3659 +           < \\_ => "" , "" >
3660 +       ).s!Posit ;
3661 +     sup =
3662 +       table {
3663 +         VSupAcc => -- Supin
3664 +           part_stem + "um" ;
3665 +         VSupAbl => -- Supin
3666 +           part_stem + "u"
3667 +       } ;
3668 +     part= table {
3669 +       VActPres => table {
3670 +         Ag ( Fem | Masc ) n c =>
3671 +           ( mkNoun ( pres_stem + fill.p1 + "ns" ) ( pres_stem + fill.p1 + "ntem" ) ( pres_stem + fill.p1 + "ntis" )
3672 +               ( pres_stem + fill.p1 + "nti" ) ( pres_stem + fill.p1 + "nte" ) ( pres_stem + fill.p1 + "ns" )
```

```
3673 +               ( pres_stem + fill.p1 + "ntes" ) ( pres_stem + fill.p1 + "ntes" ) ( pres_stem + fill.p1 + "ntium" )
3674 +               ( pres_stem + fill.p1 + "ntibus" )
3675 +               Masc ).s ! n ! c ;
3676 +         Ag Neutr n c =>
3677 +           ( mkNoun ( pres_stem + fill.p1 + "ns" ) ( pres_stem + fill.p1 + "ns" ) ( pres_stem + fill.p1 + "ntis" )
3678 +               ( pres_stem + fill.p1 + "nti" ) ( pres_stem + fill.p1 + "nte" ) ( pres_stem + fill.p1 + "ns" )
3679 +               ( pres_stem + fill.p1 + "ntia" ) ( pres_stem + fill.p1 + "ntia" ) ( pres_stem + fill.p1 + "ntium" )
3680 +               ( pres_stem + fill.p1 + "ntibus" )
3681 +               Masc ).s ! n ! c
3682 +       } ;
3683 +
3684 +       VActFut =>
3685 +         ( mkAdjective
3686 +             ( mkNoun ( part_stem + "urus" ) ( part_stem + "urum" ) ( part_stem + "uri" )
3687 +                 ( part_stem + "uro" ) ( part_stem + "uro" ) ( part_stem + "ure" ) ( part_stem + "uri" )
3688 +                 ( part_stem + "uros" ) ( part_stem + "urorum" ) ( part_stem + "uris" )
3689 +                 Masc )
3690 +             ( mkNoun ( part_stem + "ura" ) ( part_stem + "uram" ) ( part_stem + "urae" )
3691 +                 ( part_stem + "urae" ) ( part_stem + "ura" ) ( part_stem + "ura" )( part_stem + "urae" )
3692 +                 ( part_stem + "uras" ) ( part_stem +"urarum" ) ( part_stem + "uris" )
3693 +                 Fem )
3694 +             ( mkNoun ( part_stem + "urum" ) ( part_stem + "urum" ) ( part_stem + "uri" )
3695 +                 ( part_stem + "uro" ) ( part_stem + "uro" ) ( part_stem + "urum" ) ( part_stem + "ura" )
3696 +                 ( part_stem + "ura" ) ( part_stem + "urorum" ) ( part_stem + "uris" )
3697 +                 Neutr )
3698 +           < \\_ => "" , "" >
3699 +           < \\_ => "" , "" >
3700 +         ).s!Posit ;
3701 +       VPassPerf =>
3702 +         ( mkAdjective
3703 +             ( mkNoun ( part_stem + "us" ) ( part_stem + "um" ) ( part_stem + "i" ) ( part_stem + "o" )
3704 +                 ( part_stem + "o" ) ( part_stem + "e" ) ( part_stem + "i" ) ( part_stem + "os" )
3705 +                 ( part_stem + "orum" ) ( part_stem + "is" )
3706 +                 Masc )
3707 +             ( mkNoun ( part_stem + "a" ) ( part_stem + "am" ) ( part_stem + "ae" ) ( part_stem + "ae" )
3708 +                 ( part_stem + "a" ) ( part_stem + "a" ) ( part_stem + "ae" ) ( part_stem + "as" )
3709 +                 ( part_stem + "arum" ) ( part_stem + "is" )
3710 +                 Fem )
3711 +             ( mkNoun ( part_stem + "um" ) ( part_stem + "um" ) ( part_stem + "i" ) ( part_stem + "o" )
3712 +               ( part_stem + "o" ) ( part_stem + "um" ) ( part_stem + "a" ) ( part_stem + "a" )
3713 +                 ( part_stem + "orum" ) ( part_stem + "is" )
3714 +                 Neutr )
3715 +           < \\_ => "" , "" >
3716 +           < \\_ => "" , "" >
3717 +         ).s!Posit
3718 +       }
3719 +   } ;
3720 +
3721 +
3722 + mkDeponent : ( sequi,sequ,sequi,sequa,sequeba,sequere,seque,sequi,secut : Str ) -> Verb =
3723 +   \inf_pres,pres_stem,pres_ind_base,pres_conj_base,impf_ind_base,impf_conj_base,fut_I_base,imp_base,part_stem ->
3724 +   let fill : Str * Str =
3725 +         case pres_ind_base of {
3726 +           _ + ( "a" | "e" ) => < "" , "" >;
3727 +           _ => < "u" , "e" >
3728 +         }
3729 +   in
3730 +   {
3731 +     act =
3732 +       table {
3733 +          VAct VSim (VPres VInd)  Sg P1 => -- Present Indicative
3734 +           ( case pres_ind_base of {
3735 +               _ + "a" =>  ( init pres_ind_base ) ;
3736 +               _ => pres_ind_base
3737 +               }
3738 +           ) + "o" + passPresEnding Sg P1 ;
3739 +         VAct VSim (VPres VInd)  Sg P2 => -- Present Indicative
3740 +           ( case inf_pres of {
3741 +               _ + "ri" => pres_ind_base  ;
3742 +               _ => ( case pres_ind_base of {
3743 +                        _ + "i" => init pres_ind_base ;
3744 +                        _ => pres_ind_base
3745 +                        }
3746 +                 ) + "e"
3747 +               }
```

```
3748 +          ) + passPresEnding Sg P2 ;
3749 +         VAct VSim (VPres VInd)  Pl P3 => -- Present Indicative
3750 +          pres_ind_base + fill.p1 + passPresEnding Pl P3 ;
3751 +         VAct VSim (VPres VInd)  n  p  => -- Present Indicative
3752 +          pres_ind_base +
3753 +          ( case pres_ind_base of {
3754 +              _ + #consonant => "i" ;
3755 +              _ => ""
3756 +              }
3757 +          ) + passPresEnding n p ;
3758 +         VAct VSim (VPres VConj) n  p  => -- Present Conjunctive
3759 +          pres_conj_base + passPresEnding n p ;
3760 +         VAct VSim (VImpf VInd)  n  p  => -- Imperfect Indicative
3761 +          impf_ind_base + passPresEnding n p ;
3762 +         VAct VSim (VImpf VConj) n  p  => -- Imperfect Conjunctive
3763 +          impf_conj_base + passPresEnding n p ;
3764 +         VAct VSim VFut          Sg P1 => -- Future I
3765 +          (init fut_I_base ) +
3766 +          ( case fut_I_base of {
3767 +              _ + "bi" => "o" ;
3768 +              _ => "a"
3769 +              }
3770 +          ) + passPresEnding Sg P1 ;
3771 +         VAct VSim VFut          Sg P2 => -- Future I
3772 +          ( case fut_I_base of {
3773 +              _ + "bi" => ( init fut_I_base ) + "e" ;
3774 +              _ => fut_I_base
3775 +              }
3776 +          ) + passPresEnding Sg P2 ;
3777 +         VAct VSim VFut          Pl P3 => -- Future I
3778 +          (init fut_I_base ) +
3779 +          ( case fut_I_base of {
3780 +              _ + "bi" => "u" ;
3781 +              _ => "e"
3782 +              }
3783 +          ) + passPresEnding Pl P3 ;
3784 +
3785 +         VAct VSim VFut          n  p  => -- Future I
3786 +          fut_I_base + passPresEnding n p ;
3787 +         VAct VAnt (VPres VInd)  n  p  => -- Prefect Indicative
3788 +          "######" ; -- Use participle
3789 +         VAct VAnt (VPres VConj) n  p  => -- Prefect Conjunctive
3790 +          "######" ; -- Use participle
3791 +         VAct VAnt (VImpf VInd)  n  p  => -- Plusperfect Indicative
3792 +          "######" ; -- Use participle
3793 +         VAct VAnt (VImpf VConj) n  p  => -- Plusperfect Conjunctive
3794 +          "######" ; -- Use participle
3795 +         VAct VAnt VFut          n  p  => -- Future II
3796 +          "######" -- Use participle
3797 +       } ;
3798 +     pass =
3799 +      \\_ => "######" ; -- no passive forms
3800 +     inf =
3801 +      table {
3802 +          VInfActPres         => -- Infinitive Present Active
3803 +           inf_pres ;
3804 +          VInfActPerf Masc    => -- Infinitive Perfect Active
3805 +           part_stem + "um" ;
3806 +         VInfActPerf Fem     => -- Infinitive Perfect Active
3807 +           part_stem + "am" ;
3808 +         VInfActPerf Neutr   => -- Infinitive Perfect Active
3809 +           part_stem + "um" ;
3810 +         VInfActFut Masc     => -- Infinitive Future Active
3811 +           part_stem + "urum" ;
3812 +         VInfActFut Fem      => -- Infinitive Perfect Active
3813 +           part_stem + "uram" ;
3814 +         VInfActFut Neutr    => -- Infinitive Perfect Active
3815 +           part_stem + "urum" ;
3816 +         VInfPassPres        => -- Infinitive Present Passive
3817 +           "######" ; -- no passive form
3818 +         VInfPassPerf _      => -- Infinitive Perfect Passive
3819 +           "######" ; -- no passive form
3820 +         VInfPassFut         => -- Infinitive Future Passive
3821 +           "######" -- no passive form
3822 +       } ;
```

```
3823 +      imp =
3824 +       table {
3825 +         VImpl1 Sg              => -- Imperative I
3826 +           ( case inf_pres of {
3827 +               _ + "ri" => imp_base ;
3828 +               _ => (init imp_base ) + "e"
3829 +               }
3830 +           ) + "re" ;
3831 +         VImpl1 Pl              => -- Imperative I
3832 +           imp_base + "mini" ;
3833 +         VImp2 Sg ( P2 | P3 ) => -- Imperative II
3834 +           imp_base + "tor" ;
3835 +         VImp2 Pl P2            => -- Imperative II
3836 +           "######" ; -- really no such form?
3837 +         VImp2 Pl P3            => -- Imperative II
3838 +           pres_ind_base + fill.p1 + "ntor" ;
3839 +         _ => "######" -- No imperative form
3840 +       } ;
3841 +      ger =
3842 +       table {
3843 +         VGenAcc => -- Gerund
3844 +           pres_stem + fill.p2 + "ndum" ;
3845 +         VGenGen => -- Gerund
3846 +           pres_stem + fill.p2 + "ndi" ;
3847 +         VGenDat => -- Gerund
3848 +           pres_stem + fill.p2 + "ndo" ;
3849 +         VGenAbl => -- Gerund
3850 +           pres_stem + fill.p2 + "ndo"
3851 +       } ;
3852 +      geriv =
3853 +       ( mkAdjective
3854 +           ( mkNoun ( pres_stem + fill.p2 + "ndus" ) ( pres_stem + fill.p2 + "ndum" )
3855 +               ( pres_stem + fill.p2 + "ndi" ) ( pres_stem + fill.p2 + "ndo" ) ( pres_stem + fill.p2 + "ndo" )
3856 +               ( pres_stem + fill.p2 + "nde" ) ( pres_stem + fill.p2 + "ndi" ) ( pres_stem + fill.p2 + "ndos" )
3857 +               ( pres_stem + fill.p2 + "ndorum" ) ( pres_stem + fill.p2 + "ndis" )
3858 +               Masc )
3859 +           ( mkNoun ( pres_stem + fill.p2 + "nda" ) ( pres_stem + fill.p2 + "ndam" )
3860 +               ( pres_stem + fill.p2 + "ndae" ) ( pres_stem + fill.p2 + "ndae" ) ( pres_stem + fill.p2 + "nda" )
3861 +                     ( pres_stem + fill.p2 + "nda" ) ( pres_stem + fill.p2 + "ndae" ) ( pres_stem + fill.p2 + "ndas" )
3862 +               (pres_stem + fill.p2 +"ndarum" ) ( pres_stem + fill.p2 + "ndis" )
3863 +               Fem )
3864 +           ( mkNoun ( pres_stem + fill.p2 + "ndum" ) ( pres_stem + fill.p2 + "ndum" )
3865 +               ( pres_stem + fill.p2 + "ndi" ) ( pres_stem + fill.p2 + "ndo" ) ( pres_stem + fill.p2 + "ndo" )
3866 +               ( pres_stem + fill.p2 + "ndum" ) ( pres_stem + fill.p2 + "nda" ) ( pres_stem + fill.p2 + "nda" )
3867 +               ( pres_stem + fill.p2 + "ndorum" ) ( pres_stem + fill.p2 + "ndis" )
3868 +                     Neutr )
3869 +           < \\_ => "" , "" >
3870 +           < \\_ => "" , "" >
3871 +       ).s!Posit ;
3872 +      sup =
3873 +       table {
3874 +         VSupAcc => -- Supin
3875 +           part_stem + "um" ;
3876 +         VSupAbl => -- Supin
3877 +           part_stem + "u"
3878 +       } ;
3879 +      -- Bayer-Lindauer 44 1
3880 +      part = table {
3881 +       VActPres =>
3882 +         table {
3883 +           Ag ( Fem | Masc ) n c =>
3884 +             ( mkNoun ( pres_stem + fill.p2 + "ns" ) ( pres_stem + fill.p2 + "ntem" )
3885 +                 ( pres_stem + fill.p2 + "ntis" ) ( pres_stem + fill.p2 + "nti" ) ( pres_stem + fill.p2 + "nte" )
3886 +                 ( pres_stem + fill.p2 + "ns" ) ( pres_stem + fill.p2 + "ntes" ) ( pres_stem + fill.p2 + "ntes" )
3887 +                 ( pres_stem + fill.p2 + "ntium" ) ( pres_stem + fill.p2 + "ntibus" )
3888 +                 Masc ).s ! n ! c ;
3889 +           Ag Neutr n c =>
3890 +             ( mkNoun ( pres_stem + fill.p2 + "ns" ) ( pres_stem + fill.p2 + "ns" )
3891 +               ( pres_stem + fill.p2 + "ntis" ) ( pres_stem + fill.p2 + "nti" ) ( pres_stem + fill.p2 + "nte" )
3892 +                 ( pres_stem + fill.p2 + "ns" ) ( pres_stem + fill.p2 + "ntia" ) ( pres_stem + fill.p2 + "ntia" )
3893 +                 ( pres_stem + fill.p2 + "ntium" ) ( pres_stem + fill.p2 + "ntibus" )
3894 +                 Masc ).s ! n ! c
3895 +         } ;
3896 +       VActFut =>
3897 +         ( mkAdjective
```

```
3898 +              ( mkNoun ( part_stem + "urus" ) ( part_stem + "urum" ) ( part_stem + "uri" )
3899 +                  ( part_stem + "uro" ) ( part_stem + "uro" ) ( part_stem + "ure" ) ( part_stem + "uri" )
3900 +                  ( part_stem + "uros" ) ( part_stem + "urorum" ) ( part_stem + "uris" )
3901 +                  Masc )
3902 +              ( mkNoun ( part_stem + "ura" ) ( part_stem + "uram" ) ( part_stem + "urae" )
3903 +                  ( part_stem + "urae" ) ( part_stem + "ura" ) ( part_stem + "ura" )( part_stem + "urae" )
3904 +                  ( part_stem + "uras" ) ( part_stem +"urarum" ) ( part_stem + "uris" )
3905 +                  Fem )
3906 +              ( mkNoun ( part_stem + "urum" ) ( part_stem + "urum" ) ( part_stem + "uri" )
3907 +                  ( part_stem + "uro" ) ( part_stem + "uro" ) ( part_stem + "urum" ) ( part_stem + "ura" )
3908 +                  ( part_stem + "ura" ) ( part_stem + "urorum" ) ( part_stem + "uris" )
3909 +                  Neutr )
3910 +              < \\_ => "" , "" >
3911 +              < \\_ => "" , "" >
3912 +          ).s!Posit ;
3913 +        VPassPerf =>
3914 +          ( mkAdjective
3915 +              ( mkNoun ( part_stem + "us" ) ( part_stem + "um" ) ( part_stem + "i" )
3916 +                  ( part_stem + "o" ) ( part_stem + "o" ) ( part_stem + "e" )
3917 +                  ( part_stem + "i" ) ( part_stem + "os" ) ( part_stem + "orum" )
3918 +                  ( part_stem + "is" )
3919 +                  Masc )
3920 +              ( mkNoun ( part_stem + "a" ) ( part_stem + "am" ) ( part_stem + "ae" )
3921 +                  ( part_stem + "ae" ) ( part_stem + "a" ) ( part_stem + "a" )
3922 +                  ( part_stem + "ae" ) ( part_stem + "as" ) ( part_stem + "arum" )
3923 +                  ( part_stem + "is" )
3924 +                  Fem )
3925 +              ( mkNoun ( part_stem + "um" ) ( part_stem + "um" ) ( part_stem + "i" )
3926 +                  ( part_stem + "o" ) ( part_stem + "o" ) ( part_stem + "um" )
3927 +                  ( part_stem + "a" ) ( part_stem + "a" ) ( part_stem + "orum" )
3928 +                  ( part_stem + "is" )
3929 +                  Neutr )
3930 +              < \\_ => "" , "" >
3931 +              < \\_ => "" , "" >
3932 +          ).s!Posit
3933 +        }
3934 +    } ;
3935
3936      actPresEnding : Number -> Person -> Str =
3937        useEndingTable <"m", "s", "t", "mus", "tis", "nt"> ;
3938
3939      actPerfEnding : Number -> Person -> Str =
3940 -      useEndingTable <"", "sti", "t", "mus", "stis", "erunt"> ;
3941 -
3942 +      useEndingTable <"i", "isti", "it", "imus", "istis", "erunt"> ;
3943 +
3944 +  passPresEnding : Number -> Person -> Str =
3945 +    useEndingTable <"r", "ris", "tur", "mur", "mini", "ntur"> ;
3946 +
3947 +  passFutEnding : Str -> Number -> Person -> Str =
3948 +    \lauda,n,p ->
3949 +    let endings : Str * Str * Str * Str * Str * Str = case lauda of {
3950 +          ( _ + "a" ) |
3951 +            ( _ + "e" ) => < "bo" , "be" , "bi" , "bi" , "bi" , "bu" > ;
3952 +          _             => < "a"  , "e"  , "e"  , "e"  , "e"  , "e" >
3953 +          }
3954 +    in
3955 +    (useEndingTable endings n p) + passPresEnding n p ;
3956 +
3957      useEndingTable : (Str*Str*Str*Str*Str*Str) -> Number -> Person -> Str =
3958        \es,n,p -> case n of {
3959          Sg => case p of {
3960            P1 => es.p1 ;
3961            P2 => es.p2 ;
3962            P3 => es.p3
3963            } ;
3964          Pl => case p of {
3965            P1 => es.p4 ;
3966            P2 => es.p5 ;
3967            P3 => es.p6
3968            }
3969          } ;
3970
3971 -  esse_V : Verb =
3972 -    let
```

```
3973 -       esse = mkVerb "es" "si" "era" "sum" "sunt" "esse" "fui" "*futus"
3974 -                     "ero" "erunt" "eri" ;
3975 -     in {
3976 -       act = table {
3977 -         VAct VSim (VPres VInd)   Sg P2 => "es" ;
3978 -         VAct VSim (VPres VInd)   Pl P1 => "sumus" ;
3979 -         v => esse.act ! v
3980 -         } ;
3981 -       inf = esse.inf
3982 -       } ;
3983 -
3984 -   verb1 : Str -> Verb = \celare ->
3985 -     let
3986 -       cela = Predef.tk 2 celare ;
3987 -       cel  = init cela ;
3988 -       celo = cel + "o" ;
3989 -       cele = cel + "e" ;
3990 -       celavi = cela + "vi" ;
3991 -       celatus = cela + "tus" ;
3992 -     in mkVerb cela cele cela celo (cela + "nt") celare celavi celatus
3993 -               (cela + "bo") (cela + "bunt") (cela + "bi") ;
3994 -
3995 -   verb2 : Str -> Verb = \habere ->
3996 -     let
3997 -       habe = Predef.tk 2 habere ;
3998 -       hab  = init habe ;
3999 -       habeo = habe + "o" ;
4000 -       habea = habe + "a" ;
4001 -       habui = hab + "ui" ;
4002 -       habitus = hab + "itus" ;
4003 -     in mkVerb habe habea habe habeo (habe + "nt") habere habui habitus
4004 -               (habe + "bo") (habe + "bunt") (habe + "bi") ;
4005 -
4006 -   verb3 : (_,_,_ : Str) -> Verb = \gerere,gessi,gestus ->
4007 -     let
4008 -       gere = Predef.tk 2 gerere ;
4009 -       ger  = init gere ;
4010 -       gero = ger + "o" ;
4011 -       geri = ger + "i" ;
4012 -       gera = ger + "a" ;
4013 -     in mkVerb geri gera gere gero (ger + "unt") gerere gessi gestus
4014 -               (ger + "am") (ger + "ent") gere ;
4015 -
4016 -   verb3i : (_,_,_ : Str) -> Verb = \iacere,ieci,iactus ->
4017 -     let
4018 -       iac   = Predef.tk 3 iacere ;
4019 -       iaco  = iac + "io" ;
4020 -       iaci  = iac + "i" ;
4021 -       iacie = iac + "ie" ;
4022 -       iacia = iac + "ia" ;
4023 -     in mkVerb iaci iacia iacie iaco (iaci + "unt") iacere ieci iactus
4024 -               (iac + "iam") (iac + "ient") iacie ;
4025 -
4026 -   verb4 : (_,_,_ : Str) -> Verb = \sentire,sensi,sensus ->
4027 -     let
4028 -       senti  = Predef.tk 2 sentire ;
4029 -       sentio = senti + "o" ;
4030 -       sentia = senti + "a" ;
4031 -       sentie = senti + "e" ;
4032 -     in mkVerb senti sentia sentie sentio (senti + "unt") sentire sensi sensus
4033 -               (senti + "am") (senti + "ent") sentie ;
4034 -
4035 -
4036 --- smart paradigms
4037 -
4038 -   verb_pppi : (iacio,ieci,iactus,iacere : Str) -> Verb =
4039 -     \iacio,ieci,iactus,iacere ->
4040 -     case iacere of {
4041 -     _ + "are" => verb1 iacere ;
4042 -     _ + "ire" => verb4 iacere ieci iactus ;
4043 -     _ + "ere" => case iacio of {
4044 -       _ + "eo" => verb2 iacere ;
4045 -       _ + "io" => verb3i iacere ieci iactus ;
4046 -       _ => verb3 iacere ieci iactus
4047 -       } ;
```

```
4048 -     _ => Predef.error ("verb_pppi: illegal infinitive form" ++ iacere)
4049 -     } ;
4050 -
4051 -   verb : (iacere : Str) -> Verb =
4052 -     \iacere ->
4053 -     case iacere of {
4054 -     _ + "are" => verb1 iacere ;
4055 -     _ + "ire" => let iaci = Predef.tk 2 iacere
4056 -                  in verb4 iacere (iaci + "vi") (iaci + "tus") ;
4057 -     _ + "ere" => verb2 iacere ;
4058 -     _ => Predef.error ("verb: illegal infinitive form" ++ iacere)
4059 -     } ;
4060 -
4061   -- pronouns
4062
4063 +param
4064 +   PronReflForm = PronRefl | PronNonRefl ;
4065 +   PronDropForm = PronDrop | PronNonDrop;
4066 +--   PronIndefUsage = PronSubst | PronAdj ;
4067 +--   PronIndefPol = PronPos | PronNeg ;
4068 +--   PronIndefMeaning = PronSomeone | PronCertainOne | PronEvery ;
4069 +--   PronType = PronPers PronReflForm | PronPoss PronReflForm | PronDemo | PronRelat | PronInterrog |
4070 +--     PronIndef PronIndefUsage PronIndefPol PronIndefMeaning ;
4071 +
4072 +oper
4073 +
4074     Pronoun : Type = {
4075 -     s : Case => Str ;
4076 +     pers : PronDropForm => PronReflForm => Case => Str ;
4077 +     poss : PronReflForm => Agr => Str ;
4078       g : Gender ;
4079       n : Number ;
4080       p : Person ;
4081       } ;
4082
4083 -   mkPronoun : (_,_,_,_,_ : Str) -> Gender -> Number -> Person -> Pronoun =
4084 -     \ego,me,mei,mihi,mee,g,n,p -> {
4085 -       s = pronForms ego me mei mihi mee ;
4086 -       g = g ;
4087 -       n = n ;
4088 -       p = p
4089 -       } ;
4090 +   pronForms = overload {
4091 +     pronForms : (_,_,_,_,_ : Str) -> Case => Str =
4092 +       \ego,me,mei,mihi,mee -> table Case [ego ; me ; mei ; mihi ; mee ; ego] ;
4093 +     pronForms : (_,_,_,_,_ : Str) -> Case => Str =
4094 +       \meus,meum,mei,meo,meoo,mi -> table Case [meus ; meum ; mei ; meo ; meoo ; mi] ;
4095 +     };
4096 +
4097 +   createPronouns : Gender -> Number -> Person -> ( ( PronDropForm => PronReflForm => Case => Str ) * ( PronReflForm => Agr => Str ) ) = \g,n,
4098 +     case <n,p> of {
4099 +       <Sg,P1> =>
4100 +         <
4101 +         table {
4102 +           PronDrop    => \\_,_ => "" ;
4103 +           PronNonDrop => \\_ => pronForms "ego" "me" "mei" "mihi" "me" "me"
4104 +         },
4105 +         \\_ => table {
4106 +           Ag Masc  Sg c => ( pronForms "meus" "meum" "mei" "meo" "meo" "mi" ) ! c ;
4107 +           Ag Masc  Pl c => ( pronForms "mei" "meos" "meorum" "meis" "meis" "mei" ) ! c ;
4108 +           Ag Fem   Sg c => ( pronForms "mea" "meam" "meae" "meae" "mea" "mea" ) ! c ;
4109 +           Ag Fem   Pl c => ( pronForms "meae" "meas" "mearum" "meis" "meis" "meae" ) ! c ;
4110 +           Ag Neutr Sg c => ( pronForms "meum" "meum" "mei" "meo" "meo" "meum" ) ! c ;
4111 +           Ag Neutr Pl c => ( pronForms "mea" "mea" "meorum" "meis" "meis" "mea" ) ! c
4112 +         }
4113 +         > ;
4114 +       <Sg,P2> =>
4115 +         <
4116 +         table {
4117 +           PronDrop => \\_,_ => "" ;
4118 +           PronNonDrop => \\_ => pronForms "tu"  "te" "tui" "tibi" "te" "te"
4119 +         } ,
4120 +         \\_ => table {
4121 +           Ag Masc  Sg c => ( pronForms "tuus" "tuum" "tui" "tuo" "tu" "tue" ) ! c ;
4122 +           Ag Masc  Pl c => ( pronForms "tui" "tuos" "tuorum" "tuis" "tuis" "tui" ) ! c ;
```

```
4123 +          Ag Fem    Sg  c => ( pronForms "tua" "tuam" "tuae" "tuae" "tua" "tua" ) ! c ;
4124 +          Ag Fem    Pl  c => ( pronForms "tuae" "tuas" "tuarum" "tuis" "tuis" "tuae" ) ! c ;
4125 +          Ag Neutr  Sg  c => ( pronForms "tuum" "tuum" "tui" "tuo" "tuo" "tuum" ) ! c ;
4126 +          Ag Neutr  Pl  c => ( pronForms "tua" "tua" "tuorum" "tuis" "tuis" "tua" ) ! c
4127 +        }
4128 +        > ;
4129 +      <Pl,P1> =>
4130 +        <
4131 +        table {
4132 +          PronDrop => \\_,_ => "" ;
4133 +          PronNonDrop => \\_ => pronForms "nos" "nos" "nostri" "nobis" "nobis" -- nostrum
4134 +        } ,
4135 +        \\_ => table {
4136 +          Ag Masc   Sg  c => ( pronForms "noster" "nostrum" "nostri" "nostro" "nostro" "noster" ) ! c ;
4137 +          Ag Masc   Pl  c => ( pronForms "nostri" "nostros" "nostrorum" "nostris" "nostris" "nostri" ) ! c ;
4138 +          Ag Fem    Sg  c => ( pronForms "nostra" "nostram" "nostrae" "nostrae" "nostra" "nostra" ) ! c ;
4139 +          Ag Fem    Pl  c => ( pronForms "nostrae" "nostras" "nostrarum" "nostris" "nostris" "nostrae" ) ! c ;
4140 +          Ag Neutr  Sg  c => ( pronForms "nostrum" "nostrum" "nostri" "nostro" "nostro" "nostrum" ) ! c ;
4141 +          Ag Neutr  Pl  c => ( pronForms "nostra" "nostra" "nostrorum" "nostris" "nostris" "nostra" ) ! c
4142 +        }
4143 +        > ;
4144 +      <Pl,P2> =>
4145 +        <
4146 +        table {
4147 +          PronDrop => \\_,_ => "" ;
4148 +          PronNonDrop => \\_ => pronForms "vos" "vos" "vestri" "vobis" "vobis" -- vestrum
4149 +        } ,
4150 +        \\_ => table {
4151 +          Ag Masc   Sg  c => ( pronForms "vester" "vestrum" "vestri" "vestro" "vestro" "vester" ) ! c ;
4152 +          Ag Masc   Pl  c => ( pronForms "vestri" "vestros" "vestrorum" "vestris" "vestris" "vestri" ) ! c ;
4153 +          Ag Fem    Sg  c => ( pronForms "vestra" "vestram" "vestrae" "vestrae" "vestra" "vestra" ) ! c ;
4154 +          Ag Fem    Pl  c => ( pronForms "vestrae" "vestras" "vestrarum" "vestris" "vestris" "vestrae" ) ! c ;
4155 +          Ag Neutr  Sg  c => ( pronForms "vestrum" "vestrum" "vestri" "vestro" "vestro" "vestrum" ) ! c ;
4156 +          Ag Neutr  Pl  c => ( pronForms "vestra" "vestra" "vestrorum" "vestris" "vestris" "vestra" ) ! c
4157 +        }
4158 +        >;
4159 +      <_ ,P3> =>
4160 +        <
4161 +        table {
4162 +          PronDrop => \\_,_ => "" ;
4163 +          PronNonDrop =>
4164 +            table {
4165 +              PronNonRefl =>
4166 +                case <g,n> of {
4167 +                  <Masc ,Sg> => pronForms "is"  "eum" "eius"  "ei" "eo" ;
4168 +                  <Fem  ,Sg> => pronForms "ea"  "eam" "eius"  "ei" "ea" ;
4169 +                  <Neutr,Sg> => pronForms "id"  "id"  "eius"  "ei" "eo" ;
4170 +                  <Masc ,Pl> => pronForms "ei"  "eos" "eorum" "eis" "eis" ;
4171 +                  <Fem  ,Pl> => pronForms "eae" "eas" "earum" "eis" "eis" ;
4172 +                  <Neutr,Pl> => pronForms "ea"  "ea"  "eorum" "eis" "eis"
4173 +                } ;
4174 +              PronRefl => pronForms "######" "se" "sui" "sibi" "se"
4175 +            }
4176 +        } ,
4177 +        table {
4178 +          PronNonRefl =>
4179 +            \\_ =>
4180 +            case <g,n> of {
4181 +              <_,Sg>      => "eius" ;
4182 +              <Masc,Pl>   => "eorum" ;
4183 +              <Fem, Pl>   => "earum" ;
4184 +              <Neutr,Pl>  => "eorum"
4185 +            };
4186 +          PronRefl =>
4187 +            table {
4188 +              Ag Masc   Sg  c => ( pronForms "suus" "suum" "sui" "suo" "suo" ) ! c ;
4189 +              Ag Masc   Pl  c => ( pronForms "sui" "suos" "suorum" "suis" "suis" ) ! c ;
4190 +              Ag Fem    Sg  c => ( pronForms "sua" "suam" "suae" "suae" "sua" ) ! c ;
4191 +              Ag Fem    Pl  c => ( pronForms "suae" "suas" "suarum" "suis" "suis" ) ! c ;
4192 +              Ag Neutr  Sg  c => ( pronForms "suum" "suum" "sui" "suo" "suo" ) ! c ;
4193 +              Ag Neutr  Pl  c => ( pronForms "sua" "sua" "suorum" "suis" "suis" ) ! c
4194 +            }
4195 +        }
4196 +        >
4197 +--     ;
```

```
4198 +--       _ =>
4199 +--     < \\_,_,_ => "######!" , \\_,_ => "######!" > -- should never be reached
4200 +    } ;
4201
4202 -  pronForms : (_,_,_,_,_ : Str) -> Case => Str =
4203 -    \ego,me,mei,mihi,mee -> table Case [ego ; me ; mei ; mihi ; mee ; ego] ;
4204 -
4205 -  personalPronoun : Gender -> Number -> Person -> Pronoun = \g,n,p -> {
4206 -    s = case <g,n,p> of {
4207 -      <_,Sg,P1> => pronForms "ego" "me" "mei" "mihi" "me" ;
4208 -      <_,Sg,P2> => pronForms "tu"  "te" "tui" "tibi" "te" ;
4209 -      <_,Pl,P1> => pronForms "nos" "nos" "nostri" "nobis" "nobis" ; -- nostrum
4210 -      <_,Pl,P2> => pronForms "vos" "vos" "vestri" "vobis" "vobis" ; -- vestrum
4211 -      <Masc, Sg,P3> => pronForms "is" "eum" "eius" "ei" "eo" ;
4212 -      <Fem,  Sg,P3> => pronForms "ea" "eam" "eius" "ei" "ea" ;
4213 -      <Neutr,Sg,P3> => pronForms "id" "id"  "eius" "ei" "eo" ;
4214 -      <Masc, Pl,P3> => pronForms "ii" "eos" "eorum" "iis" "iis" ;
4215 -      <Fem,  Pl,P3> => pronForms "ii" "eas" "earum" "iis" "iis" ;
4216 -      <Neutr,Pl,P3> => pronForms "ea" "ea"  "eorum" "iis" "iis"
4217 -      } ;
4218 -    g = g ;
4219 -    n = n ;
4220 -    p = p
4221 +  mkPronoun : Gender -> Number -> Person -> Pronoun = \g,n,p ->
4222 +    let
4223 +      -- Personal_Form * Possesive_Form
4224 +      prons : ( PronDropForm => PronReflForm => Case => Str ) * ( PronReflForm => Agr => Str ) =
4225 +      createPronouns g n p ;
4226 +    in
4227 +    {
4228 +     pers = prons.p1 ;
4229 +     poss = prons.p2 ;
4230 +     g = g ;
4231 +     n = n ;
4232 +     p = p
4233      } ;
4234
4235 +-- prepositions
4236 +
4237    Preposition : Type = {s : Str ; c : Case} ;
4238
4239 -  VP : Type = {
4240 -    fin : VActForm => Str ;
4241 -    inf : VAnter => Str ;
4242 -    obj : Str ;
4243 -    adj : Gender => Number => Str
4244 -  } ;
4245 +-- Bayer-Lindauer $149ff.
4246 +  about_P = lin Prep (mkPrep "de" Gen ) ; -- L...
4247 +  at_P = lin Prep (mkPrep "ad" Acc ) ; -- L...
4248 +  on_P = lin Prep ( mkPrep "ad" Gen ) ; -- L...
4249 +  to_P = lin Prep ( mkPrep "ad" Acc ) ; -- L...
4250 +  Gen_Prep = lin Prep ( mkPrep "" Gen ) ;
4251 +  Acc_Prep = lin Prep ( mkPrep "" Acc ) ;
4252 +  Dat_Prep = lin Prep ( mkPrep "" Dat ) ;
4253 +  Abl_Prep = lin Prep ( mkPrep "" Abl ) ;
4254
4255 -  VPSlash = VP ** {c2 : Preposition} ;
4256 +  VPSlash = VerbPhrase ** {c2 : Preposition} ;
4257
4258 -  predV : Verb -> VP = \v -> {
4259 +  predV : Verb -> VerbPhrase = \v -> {
4260      fin = v.act ;
4261      inf = v.inf ;
4262      obj = [] ;
4263 -    adj = \\_,_ => []
4264 +    adj = \\a => []
4265    } ;
4266
4267    predV2 : (Verb ** {c : Preposition}) -> VPSlash = \v -> predV v ** {c2 = v.c} ;
4268
4269    appPrep : Preposition -> (Case => Str) -> Str = \c,s -> c.s ++ s ! c.c ;
4270
4271 -  insertObj : Str -> VP -> VP = \obj,vp -> {
4272 +  insertObj : Str -> VerbPhrase -> VerbPhrase = \obj,vp -> {
```

```
4273        fin = vp.fin ;
4274        inf = vp.inf ;
4275        obj = obj ++ vp.obj ;
4276        adj = vp.adj
4277      } ;
4278
4279 -    insertAdj : (Gender => Number => Case => Str) -> VP -> VP = \adj,vp -> {
4280 +    insertAdj : (Agr => Str) -> VerbPhrase -> VerbPhrase = \adj,vp -> {
4281        fin = vp.fin ;
4282        inf = vp.inf ;
4283        obj = vp.obj ;
4284 -      adj = \\g,n => adj ! g ! n ! Nom ++ vp.adj ! g ! n
4285 +      adj = \\a => adj ! a ++ vp.adj ! a
4286      } ;
4287
4288 -    Clause = {s : VAnter => VTense => Polarity => Str} ;
4289 +--    Clause = {s : VAnter => VTense => Polarity => Str} ;
4290
4291 -    mkClause : Pronoun -> VP -> Clause = \np,vp -> {
4292 -      s = \\a,t,p => np.s ! Nom ++ vp.obj ++ vp.adj ! np.g ! np.n ++ negation p ++
4293 -          vp.fin ! VAct a t np.n np.p
4294 -    } ;
4295 +--  mkClause : Pronoun -> VP -> Clause = \np,vp -> {
4296 +--    s = \\a,t,p => np.s ! Nom ++ vp.obj ++ vp.adj ! np.g ! np.n ++ negation p ++
4297 +--        vp.fin ! VAct a t np.n np.p
4298 +--    } ;
4299
4300      negation : Polarity -> Str = \p -> case p of {
4301        Pos => [] ;
4302        Neg => "non"
4303      } ;
4304
4305   -- determiners
4306
4307      Determiner : Type = {
4308 -      s,sp : Gender => Case => Str ;
4309 +      s : Gender => Case => Str ; -- s,sp : Gender => Case => Str ; Don't know what sp is for
4310        n : Number
4311        } ;
4312
4313 +    mkDeterminer : Adjective -> Number -> Determiner = \a,n ->
4314 +      {
4315 +        n = n ;
4316 +        s = \\g,c => a.s ! Posit ! Ag g n c ;
4317 +      } ;
4318 +
4319      Quantifier : Type = {
4320 -      s,sp : Number => Gender => Case => Str ;
4321 +      s,sp : Agr => Str ;
4322        } ;
4323
4324      mkQuantifG : (_,_,_,_,_ : Str) -> (_,_,_,_ : Str) -> (_,_,_ : Str) ->
4325        Gender => Case => Str =
4326        \mn,ma,mg,md,mab,  fno,fa,fg,fab,  nn,ng,nab -> table {
4327          Masc  => pronForms mn  ma mg md mab ;
4328          Fem   => pronForms fno fa fg md fab ;
4329          Neutr => pronForms nn  nn ng md nab
4330        } ;
4331
4332      mkQuantifier : (sg,pl : Gender => Case => Str) -> Quantifier = \sg,pl ->
4333 -      let ssp = table {Sg => sg ; Pl => pl}
4334 -      in {
4335 +      let
4336 +        ssp =
4337 +         table {
4338 +           Ag g Sg c => sg ! g ! c ;
4339 +           Ag g Pl c => pl ! g ! c
4340 +         }
4341 +      in
4342 +      {
4343          s  = ssp ;
4344          sp = ssp
4345 -        } ;
4346 +      } ;
4347
```

```
4348      hic_Quantifier = mkQuantifier
4349        (mkQuantifG
4350           "hic" "hunc" "huius" "huic" "hoc"  "haec" "hanc" "huius" "hac"  "hoc" "huius" "hoc")
4351        (mkQuantifG
4352           "hi" "hos" "horum" "his" "his"  "hae" "has" "harum" "his"  "haec" "horum" "his")
4353        ;
4354
4355      ille_Quantifier = mkQuantifier
4356        (mkQuantifG
4357           "ille" "illum" "illius" "illi" "illo"
4358           "illa" "illam" "illius" "illa"
4359           "illud" "illius" "illo")
4360        (mkQuantifG
4361           "illi" "illos" "illorum" "illis" "illis"
4362           "illae" "illas" "illarum" "illis"
4363           "illa" "illorum" "illis")
4364        ;
4365
4366 -    mkPrep : Str -> Case -> {s : Str ; c : Case} = \s,c -> {s = s ; c = c} ;
4367 +    mkPrep : Str -> Case -> Preposition  = \s,c -> lin Preposition {s = s ; c = c} ;
4368 +
4369 +    mkAdv : Str -> { s: Str } = \adv -> { s = adv } ;
4370
4371    param
4372      Unit = one | ten | hundred | thousand | ten_thousand | hundred_thousand ;
4373
4374    }
4375 -
4376  diff --git a/lib/src/latin/SentenceLat.gf b/lib/src/latin/SentenceLat.gf
4377  index 78676ee..381d991 100644
4378  --- a/lib/src/latin/SentenceLat.gf
4379  +++ b/lib/src/latin/SentenceLat.gf
4380  @@ -1,67 +1,79 @@
4381  -concrete SentenceLat of Sentence = CatLat ** open Prelude, ResLat in {
4382  +concrete SentenceLat of Sentence = CatLat,TenseX ** open Prelude, ResLat in {
4383
4384      flags optimize=all_subs ;
4385
4386      lin
4387
4388 -      PredVP = mkClause ;
4389 +      PredVP np vp = -- NP -> VP -> Cl
4390 +        {
4391 +         s = \\tense,anter,pol,order =>
4392 +           case order of {
4393 +             SVO => np.s ! Nom ++ negation pol ++ vp.adj ! Ag np.g Sg Nom ++ vp.fin ! VAct ( anteriorityToVAnter anter ) ( tenseToVTense tense
4394 +             VSO => negation pol ++ vp.adj ! Ag np.g Sg Nom ++ vp.fin ! VAct ( anteriorityToVAnter anter ) ( tenseToVTense tense ) np.n np.p ++
4395 +             VOS => negation pol ++ vp.adj ! Ag np.g Sg Nom ++ vp.fin ! VAct ( anteriorityToVAnter anter ) ( tenseToVTense tense ) np.n np.p ++
4396 +             OSV => vp.obj ++ np.s ! Nom ++ negation pol ++ vp.adj ! Ag np.g Sg Nom ++ vp.fin ! VAct ( anteriorityToVAnter anter ) ( tenseToVTe
4397 +             OVS => vp.obj ++ negation pol ++ vp.adj ! Ag np.g Sg Nom ++ vp.fin ! VAct ( anteriorityToVAnter anter ) ( tenseToVTense tense ) np
4398 +             SOV => np.s ! Nom ++ vp.obj ++ negation pol ++ vp.adj ! Ag np.g Sg Nom ++ vp.fin ! VAct ( anteriorityToVAnter anter ) ( tenseToVTe
4399 +           }
4400 +        } ;
4401  --
4402  --    PredSCVP sc vp = mkClause sc.s (agrP3 Sg) vp ;
4403  --
4404  --    ImpVP vp = {
4405  --      s = \\pol,n =>
4406  --        let
4407  --          agr    = AgP2 (numImp n) ;
4408  --          verb   = infVP True vp agr ;
4409  --          dont   = case pol of {
4410  --            CNeg True => "don't" ;
4411  --            CNeg False => "do" ++ "not" ;
4412  --            _ => []
4413  --            }
4414  --        in
4415  --        dont ++ verb
4416  --    } ;
4417  --
4418  --    SlashVP np vp =
4419  --      mkClause (np.s ! Nom) np.a vp ** {c2 = vp.c2} ;
4420  --
4421  --    AdvSlash slash adv = {
4422  --      s  = \\t,a,b,o => slash.s ! t ! a ! b ! o ++ adv.s ;
```

```
4423  --        c2 = slash.c2
4424  --    } ;
4425  --
4426  --    SlashPrep cl prep = cl ** {c2 = prep.s} ;
4427  --
4428  --    SlashVS np vs slash =
4429  --      mkClause (np.s ! Nom) np.a
4430  --        (insertObj (\\_ => conjThat ++ slash.s) (predV vs))  **
4431  --        {c2 = slash.c2} ;
4432  --
4433  --    EmbedS  s  = {s = conjThat ++ s.s} ;
4434  --    EmbedQS qs = {s = qs.s ! QIndir} ;
4435  --    EmbedVP vp = {s = infVP False vp (agrP3 Sg)} ; --- agr
4436  --
4437 ---    UseCl  t p cl = {
4438 ---      s = t.s ++ p.s ++ cl.s ! t.t ! t.a ! ctr p.p ! ODir
4439 ---    } ;
4440 +    UseCl  t p cl = -- Temp -> Pol -> Cl -> S
4441 +      {
4442 +       s = t.s ++ p.s ++ cl.s ! t.t ! t.a ! p.p ! SOV
4443 +    } ;
4444  --    UseQCl t p cl = {
4445  --      s = \\q => t.s ++ p.s ++ cl.s ! t.t ! t.a ! ctr p.p ! q
4446  --    } ;
4447  --    UseRCl t p cl = {
4448  --      s = \\r => t.s ++ p.s ++ cl.s ! t.t ! t.a ! ctr p.p ! r ;
4449  --      c = cl.c
4450  --    } ;
4451  --    UseSlash t p cl = {
4452  --      s = t.s ++ p.s ++ cl.s ! t.t ! t.a ! ctr p.p  ! ODir ;
4453  --      c2 = cl.c2
4454  --    } ;
4455  --
4456  --    AdvS a s = {s = a.s ++ "," ++ s.s} ;
4457  --
4458  --    RelS s r = {s = s.s ++ "," ++ r.s ! agrP3 Sg} ;
4459  --
4460  --  oper
4461  --    ctr = contrNeg True ;  -- contracted negations
4462  --
4463  }
4464
4465 diff --git a/lib/src/latin/StructuralLat.gf b/lib/src/latin/StructuralLat.gf
4466 index 589a91b..373d72e 100644
4467 --- a/lib/src/latin/StructuralLat.gf
4468 +++ b/lib/src/latin/StructuralLat.gf
4469 @@ -1,128 +1,143 @@
4470 +
4471 +--1 Basic Latin Lexicon for Structural Categories.
4472 +-- Aarne Ranta pre 2013, Herbert Lange 2013
4473 +
4474 +-- This lexicon implements all the words in the abstract Lexicon.
4475 +-- For each entry a source is given, either a printed dictionary, a
4476 +-- printed grammar book or a link to an online source. The used printed
4477 +-- dictionaries are Langescheidts Schulwörterbuch Lateinisch 17. Edition
4478 +-- 1984 (shorter: Langenscheidts), PONS Schulwörterbuch Latein 1. Edition
4479 +-- 2012 (Shorter: Pons) and Der kleine Stowasser 3. Edition 1991 (shorter:
4480 +-- Stowasser). The Grammar book is Bayer–Lindauer: Lateinische Schulgrammatik
4481 +-- 2. Edition 1994.
4482 +
4483  concrete StructuralLat of Structural = CatLat **
4484 -  open ResLat, (P = ParadigmsLat), Prelude in
4485 +  open ResLat, ParadigmsLat, Prelude, IrregLat, ConstructX in
4486    {
4487 -
4488 +
4489    flags optimize=all ;
4490 -
4491 +
4492    lin
4493 -  above_Prep = mkPrep "super" Acc ;
4494 -  after_Prep = mkPrep "post" Acc ;
4495 ---  all_Predet = ss "all" ;
4496 -  almost_AdA, almost_AdN = ss "quasi" ;
4497 ---  although_Subj = ss "although" ;
```

```
4498 -   always_AdV = ss "semper" ;
4499 ---   and_Conj = sd2 [] "and" ** {n = Pl} ;
4500 +   above_Prep = mkPrep "super" Abl ; -- abl. (Langenscheidts)
4501 +   after_Prep = mkPrep "post" Acc ; -- acc. (Langenscheidts)
4502 +   all_Predet = ss "cuncti" ; -- (Langenscheidts)
4503 +   almost_AdA, almost_AdN = ss "quasi" ; -- (Langenscheidts)
4504 +   although_Subj = ss "quamquam" ; -- (Langenscheidts)
4505 +   always_AdV = ss "semper" ; -- (Langenscheidts)
4506 +   and_Conj = sd2 [] "et" ** {n = Pl} ; -- (Langenscheidts)
4507 ------b   and_Conj = ss "and" ** {n = Pl} ;
4508 ---   because_Subj = ss "because" ;
4509 -   before_Prep = mkPrep "ante" Acc ;
4510 ---   behind_Prep = ss "behind" ;
4511 -   between_Prep = mkPrep "inter" Acc ;
4512 ---   both7and_DConj = sd2 "both" "and" ** {n = Pl} ;
4513 -   but_PConj = ss "sed" ;
4514 -   by8agent_Prep = mkPrep "a" Abl ;
4515 -   by8means_Prep = mkPrep "per" Acc ;
4516 ---   can8know_VV, can_VV = {
4517 ---     s = table {
4518 ---       VVF VInf => ["be able to"] ;
4519 ---       VVF VPres => "can" ;
4520 ---       VVF VPPart => ["been able to"] ;
4521 ---       VVF VPresPart => ["being able to"] ;
4522 ---       VVF VPast => "could" ;       --# notpresent
4523 ---       VVPastNeg => "couldn't" ;    --# notpresent
4524 ---       VVPresNeg => "can't"
4525 ---       } ;
4526 ---     isAux = True
4527 ---     } ;
4528 ---   during_Prep = ss "during" ;
4529 ---   either7or_DConj = sd2 "either" "or" ** {n = Sg} ;
4530 ---   everybody_NP = regNP "everybody" Sg ;
4531 ---   every_Det = mkDeterminer Sg "every" ;
4532 ---   everything_NP = regNP "everything" Sg ;
4533 ---   everywhere_Adv = ss "everywhere" ;
4534 ---   few_Det = mkDeterminer Pl "few" ;
4535 +   because_Subj = ss "cum" ; -- (Langenscheidts)
4536 +   before_Prep = mkPrep "ante" Acc ; -- acc. (Langenscheidts)
4537 +   behind_Prep = mkPrep "a tergo" Acc ; -- acc. (Langenscheidts)
4538 +   between_Prep = mkPrep "inter" Acc ; -- acc. (Langenscheidts)
4539 +   both7and_DConj = sd2 "et" "et" ** {n = Pl} ; --(Langenscheidts)
4540 +   but_PConj = ss "sed" ; -- (Langenscheidts)
4541 +   by8agent_Prep = mkPrep "per" Abl ; -- (Langenscheidts)
4542 +   by8means_Prep = Abl_Prep ; -- (Langenscheidts)
4543 +   can8know_VV, can_VV = IrregLat.can_VV ; --(Langenscheidts)
4544 +   during_Prep = mkPrep "inter" Acc ; -- (Langenscheidts)
4545 +   either7or_DConj = sd2 "aut" "aut" ** {n = Sg} ; -- (Langenscheidts)
4546 +   everybody_NP = regNP  "quisque" "quemque" "cuiusque" "cuique" "quoque" "quisque" ( Masc | Fem ) Sg ;-- regNP "quisquae" Sg ; -- (Langensche
4547 +   every_Det = mkDeterminer ( mkA "omnis" ) Pl ; -- (Pons)
4548 +   everything_NP = regNP "omnia" "omnia" "omnium" "omnis" "omnia" Neutr Pl ; --regNP "omnia" Pl ; -- (Langenscheidts)
4549 +   everywhere_Adv = ss "ubique" ; -- (Langenscheidts)
4550 +   few_Det = mkDeterminer ( mkA "paulus" ) Pl ; -- (Langenscheidts)
4551 ------   first_Ord = ss "first" ; DEPRECATED
4552 -   for_Prep = mkPrep "pro" Abl ;
4553 -   from_Prep = mkPrep "de" Abl ;
4554 -   he_Pron = personalPronoun Masc Sg P3 ;
4555 -   here_Adv = ss "hic" ;
4556 ---   here7to_Adv = ss ["to here"] ;
4557 ---   here7from_Adv = ss ["from here"] ;
4558 ---   how_IAdv = ss "how" ;
4559 ---   how8many_IDet = mkDeterminer Pl ["how many"] ;
4560 ---   if_Subj = ss "if" ;
4561 -   in8front_Prep = mkPrep "coram" Abl ;
4562 -   i_Pron = personalPronoun Masc Sg P1 ;
4563 -   in_Prep = mkPrep "in" Abl ;
4564 -   it_Pron = personalPronoun Neutr Sg P3 ;
4565 ---   less_CAdv = ss "less" ;
4566 ---   many_Det = mkDeterminer Pl "many" ;
4567 ---   more_CAdv = ss "more" ;
4568 ---   most_Predet = ss "most" ;
4569 ---   much_Det = mkDeterminer Sg "much" ;
4570 ---   must_VV = {
4571 ---     s = table {
4572 ---       VVF VInf => ["have to"] ;
```

```
4573 ——        VVF VPres => "must" ;
4574 ——        VVF VPPart => ["had to"] ;
4575 ——        VVF VPresPart => ["having to"] ;
4576 ——        VVF VPast => ["had to"] ;       --# notpresent
4577 ——        VVPastNeg => ["hadn't to"] ;       --# notpresent
4578 ——        VVPresNeg => "mustn't"
4579 ——        } ;
4580 ——      isAux = True
4581 ——      } ;
4582 ------b   no_Phr = ss "no" ;
4583 -   no_Utt = ss "non" ;
4584 ——    on_Prep = ss "on" ;
4585 +   for_Prep = mkPrep "pro" Abl ; -- abl. (Langenscheidts)
4586 +   from_Prep = mkPrep "de" Abl ; -- abl. (Langenscheidts)
4587 +   he_Pron = mkPronoun Masc Sg P3 ;
4588 +   here_Adv = ss "hic" ; -- (Langenscheidts)
4589 +   here7to_Adv = ss "huc" ; -- (Langenscheidts)
4590 +   here7from_Adv = ss "hinc" ; -- (Langenscheidts)
4591 +   how_IAdv = ss "qui" ; -- modale (Langenscheidts)
4592 +   how8many_IDet = mkDeterminer (mkA "quantus" ) Pl ; -- (Pons)
4593 +   how8much_IAdv = ss "quantum" ; -- (Langenscheidts)
4594 +   if_Subj = ss "si" ; -- (Langenscheidts)
4595 +   in8front_Prep = mkPrep "ante" Acc ; -- acc. (Langenscheidts)
4596 +   i_Pron = mkPronoun Masc Sg P1 ;
4597 +   in_Prep = mkPrep "in" ( variants { Abl ; Acc } ) ; -- (Langenscheidts)
4598 +   it_Pron = mkPronoun Neutr Sg P3 ;
4599 +   less_CAdv = mkCAdv "minus" "quam" ; -- (Langenscheidts)
4600 +   many_Det = mkDeterminer ( mkA "multus" ) Pl ; -- (Langenscheidts)
4601 +   more_CAdv = mkCAdv "magis" "quam" ; -- (Langenscheidts)
4602 +   most_Predet = ss "plurimi" ; -- (Langenscheidts)
4603 +   much_Det = mkDeterminer ( mkA "multus" ) Sg ; -- (Langenscheidts)
4604 +   must_VV = mkVV ( mkV "debere" ) True ; -- (Langenscheidts)
4605 +------b   no_Phr = ss "immo" ;
4606 +   no_Utt = ss "non␣est" ; -- should be expressed by a short negated sentence (Langenscheidts)
4607 +   on_Prep = mkPrep "in" ( Acc | Abl ) ; -- (Langenscheidts)
4608  --------  one_Quant = mkDeterminer Sg "one" ; -- DEPRECATED
4609 -   only_Predet = ss "tantum" ;
4610 ——    or_Conj = sd2 [] "or" ** {n = Sg} ;
4611 ——    otherwise_PConj = ss "otherwise" ;
4612 -   part_Prep = mkPrep [] Gen ;
4613 ——    please_Voc = ss "please" ;
4614 -   possess_Prep = mkPrep [] Gen ;
4615 ——    quite_Adv = ss "quite" ;
4616 -   she_Pron = personalPronoun Fem Sg P3 ;
4617 -   so_AdA = ss "sic" ;
4618 ——    somebody_NP = regNP "somebody" Sg ;
4619 ——    someSg_Det = mkDeterminer Sg "some" ;
4620 ——    somePl_Det = mkDeterminer Pl "some" ;
4621 ——    something_NP = regNP "something" Sg ;
4622 ——    somewhere_Adv = ss "somewhere" ;
4623 +   only_Predet = ss "solum" ; -- (Langenscheidts)
4624 +   or_Conj = sd2 [] "aut" ** {n = Sg} ; -- (Langenscheidts)
4625 +   otherwise_PConj = ss "praeterea" ; -- (Pons)
4626 +   part_Prep = mkPrep [] Gen ; -- (Bayer–Lindauer $127)
4627 +   please_Voc = ss "queso" ; -- (Langenscheidts)
4628 +   possess_Prep = mkPrep [] Gen ; -- (Bayer–Lindauer $125.2)
4629 +   quite_Adv = ss "admodum" ; -- or by comparation (Langenscheidts)
4630 +   she_Pron = mkPronoun Fem Sg P3 ;
4631 +   so_AdA = ss "sic" ; -- (Langenscheidts)
4632 +   somebody_NP = regNP "aliquis" "aliquem" "alicuius" "clicui" "aliquo" "aliquis" ( Masc | Fem ) Sg ; -- (Bayer–Lindauer $60.1)
4633 +   someSg_Det = mkDeterminer ( mkA "aliquis" ) Sg ; -- (Langenscheidts)
4634 +   somePl_Det = mkDeterminer ( mkA "nonnullus" ) Pl ; -- (Langenscheidts)
4635 +   something_NP = regNP "aliquid" "aliquid" "alicuius␣rei" "alicui␣rei" "aliqua␣re" "aliquid" Masc Sg ; -- (Bayer–Lindauer $60.1)
4636 +   somewhere_Adv = ss "usquam" ; -- (Langenscheidts)
4637     that_Quant = ille_Quantifier ;
4638 ——    there_Adv = ss "there" ;
4639 ——    there7to_Adv = ss "there" ;
4640 ——    there7from_Adv = ss ["from there"] ;
4641 ——    therefore_PConj = ss "therefore" ;
4642 -   they_Pron = personalPronoun Masc Pl P3 ;
4643 +   that_Subj = ss "ut" ; -- (Langenscheidts)
4644 +   there_Adv = ss "ibi" ; -- loc. (Langenscheidts)
4645 +   there7to_Adv = ss "eo" ; -- (Pons)
4646 +   there7from_Adv = ss "inde" ; -- (Pons)
4647 +   therefore_PConj = ss "ergo" ; -- (Langenscheidts)
```

```
4648 +  they_Pron = mkPronoun Masc Pl P3 ;
4649    this_Quant = hic_Quantifier ;
4650 ---  through_Prep = ss "through" ;
4651 ---  too_AdA = ss "too" ;
4652 ---  to_Prep = ss "to" ;
4653 -  under_Prep = mkPrep "sub" Acc ;
4654 -  very_AdA = ss "valde" ;
4655 ---  want_VV = P.mkVV (P.regV "want") ;
4656 -  we_Pron = personalPronoun Masc Pl P1 ;
4657 ---  whatPl_IP = mkIP "what" "what" "what's" Pl ;
4658 ---  whatSg_IP = mkIP "what" "what" "what's" Sg ;
4659 ---  when_IAdv = ss "when" ;
4660 ---  when_Subj = ss "when" ;
4661 ---  where_IAdv = ss "where" ;
4662 ---  which_IQuant = {s = \\_ => "which"} ;
4663 +  through_Prep = mkPrep "per" Acc ; -- (Langenscheidts)
4664 +  too_AdA = ss "quoque" ; -- (Langenscheidts)
4665 +  to_Prep = mkPrep "ad" Acc; -- (Langenscheidts)
4666 +  under_Prep = mkPrep "sub" Acc ; -- (Langenscheidts)
4667 +  very_AdA = ss "valde" ; -- (Langenscheidts)
4668 +  want_VV = mkVV IrregLat.want_V True ; -- (Langenscheidts)
4669 +  we_Pron = mkPronoun Masc Pl P1 ;
4670 +  whatSg_IP = { s =pronForms "quid" "quid" "cuius" "cui" "quo" ; n = Sg } ; -- only feminine or masculine (Bayer–Lindauer §59.1)
4671 +  whatPl_IP = { s = \\_ => "" ; n = Pl } ; -- no plural forms (Bayer–Lindauer §59.1)
4672 +  when_IAdv = ss "quando" ; -- (Langenscheidts)
4673 +  when_Subj = ss "si" ; -- (Langenscheidts)
4674 +  where_IAdv = ss "ubi" ; -- (Langenscheidts)
4675 +  which_IQuant = -- (Bayer–Lindauer §59.1.2 and §58.1)
4676 +    { s = table {
4677 +        Ag Masc  Sg c => ( pronForms "qui" "quem" "cuius" "cui" "quo" ) ! c ;
4678 +        Ag Masc  Pl c => ( pronForms "qui" "quos" "quorum" "quibus" "quibus" ) ! c ;
4679 +        Ag Fem   Sg c => ( pronForms "quae" "quam" "cuius" "cui" "qua" ) ! c ;
4680 +        Ag Fem   Pl c => ( pronForms "quae" "quas" "quarum" "quibus" "quibus" ) ! c ;
4681 +        Ag Neutr Sg c => ( pronForms "quod" "quod" "cuius" "cui" "quo" ) ! c ;
4682 +        Ag Neutr Pl c => ( pronForms "quae" "quae" "quorum" "quibus" "quibus" ) ! c
4683 +        }
4684 +    } ;
4685  ----b  whichPl_IDet = mkDeterminer Pl ["which"] ;
4686  ----b  whichSg_IDet = mkDeterminer Sg ["which"] ;
4687 ---  whoPl_IP = mkIP "who" "whom" "whose" Pl ;
4688 ---  whoSg_IP = mkIP "who" "whom" "whose" Sg ;
4689 ---  why_IAdv = ss "why" ;
4690 -  without_Prep = mkPrep "sine" Abl ;
4691 -  with_Prep = mkPrep "cum" Abl ;
4692 -  yes_Utt = ss "sic" ;
4693 -  youSg_Pron = personalPronoun Masc Sg P2 ;
4694 -  youPl_Pron = personalPronoun Masc Pl P2 ;
4695 -  youPol_Pron = personalPronoun Masc Sg P2 ;
4696 +  whoSg_IP = { s =pronForms "quis" "quem" "cuius" "cui" "quo" ; n = Sg } ; -- only feminine or masculine (Bayer–Lindauer §59.1)
4697 +  whoPl_IP = { s = \\_ => "" ; n = Pl } ; -- no plural forms (Bayer–Lindauer §59.1)
4698 +
4699 +  why_IAdv = ss "cur" ; -- (Langenscheidts)
4700 +  without_Prep = mkPrep "sine" Abl ; -- abl. L..
4701 +  with_Prep = mkPrep "cum" Abl ; -- abl. L..
4702 +  yes_Utt = ss "sane" ; -- (Langenscheidts)
4703 +  youSg_Pron = mkPronoun Masc Sg P2 ;
4704 +  youPl_Pron = mkPronoun Masc Pl P2 ;
4705 +  youPol_Pron = youSg_Pron | youPl_Pron ;
4706 +
4707 +  no_Quant = { s , sp = ( mkA "nullus" ).s ! Posit } ; -- nullus (Langenscheidts)
4708 +  not_Predet = ss "non" ; -- (Langenscheidts)
4709 +  if_then_Conj = {s1 = "si" ; s2 = "######" ; n = Sg } ; -- no word in s2 field (Langenscheidts)
4710 +  at_least_AdN = ss "saltem" ; -- (Langenscheidts)
4711 +  at_most_AdN = ss "summum" ; -- (Pons)
4712 +  nobody_NP = regNP "nemo" "neminem" "neminis" "nemini" "nemine" "nemo" ( Masc | Fem ) Sg ; -- (Bayer Lindauer §60.4)
4713 +  nothing_NP = regNP "nihil" "nihil" "nullius␣rei" "nulli␣rei" "nulla␣re" "nihil" Neutr Sg ; -- (Bayer–Lindauer §60.4)
4714 +  except_Prep = mkPrep "praeter" Acc ; -- (Langenscheidts)
4715 +
4716 +  as_CAdv = mkCAdv "ita" "ut" ; -- (Langenscheidts)
4717
4718 -  lin language_title_Utt = ss "latina" ;
4719 +  have_V2 = mkV2 (mkV "habere") ; -- habeo, -ui, -itum 2 (Langenscheidts)
4720
4721 +  lin language_title_Utt = ss "lingua␣latina" ;
4722
```

```
4723   }
4724
4725  diff --git a/lib/src/latin/VerbLat.gf b/lib/src/latin/VerbLat.gf
4726  index 29f970a..5fffc4a 100644
4727  --- a/lib/src/latin/VerbLat.gf
4728  +++ b/lib/src/latin/VerbLat.gf
4729  @@ -1,50 +1,52 @@
4730  -concrete VerbLat of Verb = CatLat ** open ResLat in {
4731  +concrete VerbLat of Verb = CatLat ** open ResLat,IrregLat in {
4732
4733     flags optimize=all_subs ;
4734
4735     lin
4736       UseV = predV ;
4737
4738       SlashV2a v = predV2 v ;
4739  --     Slash2V3 v np =
4740  --       insertObjc (\\_ => v.c2 ++ np.s ! Acc) (predV v ** {c2 = v.c3}) ;
4741  --     Slash3V3 v np =
4742  --       insertObjc (\\_ => v.c3 ++ np.s ! Acc) (predVc v) ; ----
4743  --
4744  --     ComplVV v vp = insertObj (\\a => infVP v.isAux vp a) (predVV v) ;
4745  --     ComplVS v s  = insertObj (\\_ => conjThat ++ s.s) (predV v) ;
4746  --     ComplVQ v q  = insertObj (\\_ => q.s ! QIndir) (predV v) ;
4747  --     ComplVA v ap = insertObj (ap.s) (predV v) ;
4748  --
4749  --     SlashV2V v vp = insertObjc (\\a => infVP v.isAux vp a) (predVc v) ;
4750  --     SlashV2S v s  = insertObjc (\\_ => conjThat ++ s.s) (predVc v) ;
4751  --     SlashV2Q v q  = insertObjc (\\_ => q.s ! QIndir) (predVc v) ;
4752  --     SlashV2A v ap = insertObjc (\\a => ap.s ! a) (predVc v) ; ----
4753  --
4754  -    ComplSlash vp np = insertObj (appPrep vp.c2 np.s) vp ;
4755  +    ComplSlash vp np = -- VPSlash -> NP -> VP
4756  +      insertObj (appPrep vp.c2 np.s) vp ;
4757  --
4758  --     SlashVV vv vp =
4759  --       insertObj (\\a => infVP vv.isAux vp a) (predVV vv) **
4760  --         {c2 = vp.c2} ;
4761  --     SlashV2VNP vv np vp =
4762  --       insertObjPre (\\_ => vv.c2 ++ np.s ! Acc)
4763  --         (insertObjc (\\a => infVP vv.isAux vp a) (predVc vv)) **
4764  --           {c2 = vp.c2} ;
4765  --
4766  -    UseComp comp = insertAdj comp.s (predV esse_V) ;
4767  +    UseComp comp = -- Comp -> VP
4768  +      insertAdj comp.s (predV be_V) ;
4769
4770  -    AdvVP vp adv = insertObj adv.s vp ;
4771  +--   AdvVP vp adv = insertObj adv.s vp ;
4772
4773  -    AdVVP adv vp = insertObj adv.s vp ;
4774  +--   AdVVP adv vp = insertObj adv.s vp ;
4775
4776  --     ReflVP v = insertObjPre (\\a => v.c2 ++ reflPron ! a) v ;
4777  --
4778  --     PassV2 v = insertObj (\\_ => v.s ! VPPart) (predAux auxBe) ;
4779  --
4780  ----b     UseVS, UseVQ = \vv -> {s = vv.s ; c2 = [] ; isRefl = vv.isRefl} ; -- no "to"
4781  --
4782       CompAP ap = ap ;
4783  --     CompNP np = {s = \\_ => np.s ! Acc} ;
4784  --     CompAdv a = {s = \\_ => a.s} ;
4785  --
4786   }
```

Listing 4.1: Unterschiede zwischen der Anfangs- und Endversion der Lateingrammatik