1 Einleitung

Ziel dieser Arbeit ist es, ein Program zu entwickeln, das Lautgesetze auf Wörter anwendet, und die Lautgesetze vom Urindogermanischen in die altgriechischen Dialekte zu sammeln und zu sortieren.

Das junggrammatische Postulat der Ausnahmslosigkeit der Lautgesetze, das fordert, dass Lautgesetze wie mechanisch ablaufen, wirkt sehr einladend, tatsächlich einen Mechanismus zu entwickeln, der die Anwendung von Lautgesetzen simuliert. Im 19. Jh. war dies natürlich unmöglich, aber trotz der Computerrevolution des 20. Jh. hat sich offenbar noch niemand daran gemacht, diese eigentlich sehr naheliegenden Idee zu verwirklichen. (TODO: doch, Amir Zeldes ansatzweise)

Das hier besprochene Programm `lga` ist der Nachfolger eines Prototyps (komplett in `sed` geschrieben), der primitiver war und einige Mängel aufzeigte. Ebenso ist aber `lga` nur als Prototyp für ein elaborierteres Programm zu verstehen. Im Laufe der Arbeit wird sich herausstellen, welche Probleme, Mängel und weiteren Anforderungen es gibt, die man in einer nächsten Version angehen müsste.

Im ersten Teil dieser Arbeit wird `lga` in seinen Designideen und in seiner Funktionsweise vorgestellt. Im zweiten Teil wird `lga` benutzt, um die Lautentwicklung vom Urindogermanischen ins Altgriechische zu modellieren.

2 lga

2.1 Vorüberlegungen

Wenn ein Lautgesetzt wirkt, wird ein Laut (oder mehrere) durch einen anderen ersetzt. Da man Laute phonetisch notieren kann, kann man auch ein Lautgesetz durch Ersetzung von einem Zeichen durch ein anderes beschreiben.

Da Lautwandel meist durch die Lautumgebung bedingt ist und sich oft nicht nur ein Laut sondern Lautgruppen verändern (bspw. Stimmloswerdung von stimmhaften Lauten), ist es sinnvoll, eine Sprache zu entwicklen, die Lautgesetze kompakt beschreiben und von einem Computerprogramm verstanden werden kann. Eine naheliegende Wahl ist daher eine Form von *regulären Ausdrücken*, die in vielen Programmen implementiert und weithin bekannt sind. Reguläre Ausdrücke sind zwar für diesen Zweck nicht perfekt geeignet, aber für einen Prototyp ausreichend. Wenn sich gezeigt hat, welche Defizite reguläre Ausdrücke für diesen Anwendungsfall haben, wird man eine besser geeignete Sprache entwickeln und implementieren können.

Als Programmiersprache wird CHICKEN – eine Implementation von Scheme – verwendet (`http://www.call-cc.org/`).

2.2 Reguläre Ausdrücke

Hier eine (informelle) Beschreibung der hier verwendeten Untermenge regulärer Ausdrücke. Die vollständige Dokumentation für die `irregex`-Engine für CHICKEN findet sich auf `http://wiki.call-cc.org/man/4/Unit%20irregex`

Die grundlegende Funktion von regulären Ausdrücken ist das Finden (engl. *match*) von Zeichenketten (String) anhand eines Musters. Der gematchte Text kann dann durch einen anderen ersetzt werden, der Teile des Matches wieder aufnehmen kann.

Ein regulärer Ausdruck matcht einen String aus einem Alphabet; hier die Menge der Unicode *Codepoints* (nicht der *Grapheme*, die aus mehreren Codepoints bestehen können).

Ein Zeichen des Alphabets, das kein Metazeichen ist (dazu unten), matcht sich selbst. Metazeichen verlieren ihre spezielle Funktion, wenn ihnen ein Backslash `\` vorangeht. Der Ausdruck `foo` matcht also den String `foo` (und auch das `foo` in `foobar`).

Der Punkt `.` matcht ein beliebiges Zeichen. Der Ausdruck `...` matcht also alle Strings mit drei Zeichen, `\.\.\.` matcht drei Punkte.

Eine in eckigen Klammern stehende Menge an Zeichen matcht eins dieser Zeichen, oder alle außer dieser Zeichen, wenn das erste Zeichen ein Zirkumflex `^` ist. Dabei können auch Zeichenbereiche mit Bindestrich angegeben werden. `[abc]` matcht also `a`, `b` oder `c`, `[^abc]` alle Zeichen außer `a`, `b` und `c`. Alternativ hätte man auch `[a-c]` schreiben können.

Der Zirkumflex `^` und das Dollarzeichen `$` matchen den Anfang bzw. das Ende einer Zeile. `^foo` matcht also `foo` am Anfang der Zeile, `bar$` matcht `bar` am Ende der Zeile und `^quux$` die Zeile, die nur `quux` enthält. Da in `lga` jedes Wort in einer eigenen Zeile steht, kann man die Zeichen für den Anfang bzw. das Ende des Wortes benutzen.

`?`, `*` und `+` sind Quantoren und matchen, was ihnen vorangeht, null oder einmal, null bis unendlich mal und ein bis unendlich mal. Der Ausdruck `.?[a-d]*x+` z.B. matcht optional ein beliebiges Zeichen, dann eins der Zeichen `a-d` null bis unendlich mal und schließlich mindestens ein `x`.

Mit Klammern `()` werden analog zur mathematischen Notation Matches enger gruppiert. So würde `[ab]c+` `a` oder `b` und dann mindestens ein `c` matchen (also z.B. `accc` oder `bc`). `([ab]c)+` dagegen matcht `[ab]c` mindestens einmal (also z.B. `acbcbc`).

Neben ihrer Gruppierungsfunktion werden die Submatches innerhalb der Klammern für die Ersetzung gespeichert. `([0-9]+)foo([0-9]+)` würde also zwei Zahlen und `foo` dazwischen matchen. Die Zahlen wären dann in den submatchtes `1` und `2` gespeichert und können im Ersetzungstext wieder aufgegriffen werden.

Mit dem senkrechten Strich `e0|e1` wird `e0` oder `e1` gematcht (der Senkrechtstrich hat die niedrigste Präzedenz). `foo|bar+` matcht also `foo` oder mindestens ein `bar`.

`(foo|bar)+` dagegen matcht mindestens ein `foo` oder `bar`.

2.3 Funktionsweise

Im Kern arbeitet `lga` mit einer Liste von Wörtern und einer Liste von Lautgesetzen (in der Form von Textersetzung durch reguläre Ausdrücke) und wendet der Reihe nach alle Lautgesetze auf die Liste der Wörter an. Das Resultat sind die Wörter nach Anwendung aller Lautgesetze.

Die Liste der Lautgesetze aber hängt davon ab, von welcher in welche Sprache die Wörter transformiert werden sollen und wird aus einem *Baum* von Lautgesetzen und einem zugehörigen Sprachstammbaum generiert.

Die Liste der Wörter wird vorher durch eine weitere Liste von Regeln, die von der Ausgangssprache abhängt, von ihrer graphematischen Form in eine interne phonetische gebracht (was natürlich eine eindeutige Graphie voraussetzt) und am Ende durch noch eine Liste von Regeln, die von der Zielsprache abhängig ist, wieder in eine graphematische Darstellung gebracht.

Die Funktion `(run-list from to words)` generiert die drei Listen von Regeln anhand der Sprachen `from` und `to`, macht aus der Liste von Wörtern `words` aus Effizienzgründen einen einzigen String, in dem jedes Element von `words` in einer eigenen Zeile steht, wendet darauf die drei Regellisten an, und gibt das Ergebnis aus.

Zunächst wird mit der Funktion `(make-path tree start end)` anhand eines Sprachstammbaums `tree` eine Liste aller Sprachen von der Ausgangs- zur Zielsprache generiert. Der Aufruf `(make-path lang-tree 'uridg 'lesb)` würde bspw. die Liste `(uridg urgr uraiol lesb)` erzeugen.

Der Sprachstammbaum (gespeichert in der globalen Variable `lang-tree`) definiert das Verwandtschaftsverhältnis von Sprachen. Wenn eine Sprache keine Untersprachen hat, ist sie (in Scheme-Terminologie) ein Symbol (also hat z.B. `myk` keine Abkömmlinge). Hat eine Sprache Untersprachen, ist sie eine Liste, dessen erstes Element ein Symbol für die Sprache selber ist; die weiteren Elemente sind Untersprachen (wiederum Symbole oder Listen). Ein vereinfachter Stammbaum könnte also bspw. folgendermaßen aussehen:

```
(uridg (urgr (urark myk ark kypr)
             (uraiol thess boiot lesb)
             (urwgr nwgr dor)
             (urion (aion ion) (aatt att))))
```

Es sei angemerkt, dass die so beschriebenen Verhältnisse nicht streng als Verwandtschaft interpretiert werden sollten. Urion. mag zwar eine Ursprache gewesen sein, aber

auch nach der Aufspaltung von att. und ion. teilen sie noch gewisse Entwicklungen. Ebenso gibt es gemeingriechische Entwicklung die nach-urgriechisch sind. Eine präzisere Trennung von Verwandtschaft und "Sprachgruppen" würde aber das Modell möglicherweise zu sehr verkomplizieren. Die beste Möglichkeit ist es womöglich, auf die Vorsilbe *ur-* zu verzichten und bspw. `gr` gleichermaßen für urgriechisch und gemeingriechisch zu verwenden.

Der Lautgesetzbaum ist eine Liste von vier möglichen Elementen, nämlich Funktionen, die einen String als Argument nehmen und einen String zurückgeben (also Funktionen, die Lautgesetze anwenden), Symbole, die eine Sprachstufe markieren, und zwei Arten von Listen des Typs `(br/sub lang-lst rule-tree)`, deren zweites Element eine Liste von Sprachen ist, für die sie gelten (bzw. nicht gelten, wenn das erste Element `not` ist), und deren drittes Element ein weiterer Lautgesetzbaum ist. `br` (für *branch*) leitet einen neuen Sprachzweig ein, der durch `rule-tree` definiert ist (d.h. dass alle Regeln nach einem erfolgreichen `br` ignoriert werden). `sub` (für *subrules*) funktioniert ähnlich, macht aber nach Abarbeitung von `rule-tree` nach der Regel weiter. Dies impliziert, dass es sich hier bei `rule-tree` nicht um einen echten Baum handelt. `sub` ist nur dazu gedacht eine Liste von Lautgesetzen bedingt anzuwenden. Im Griechischen ist dies besonders nützlich, da zwar alle Dialekte ähnliche Entwicklungen machen, diese sich aber im einzelnen unterscheiden. Mit `sub` kann man einfach Regeln für bestimme Dialekte oder Dialektgruppen definieren ohne für jeden Dialekt einen eigenen Sprachzweig haben zu müssen.

Die Funktion `(make-rules tree path)` gibt anhand des Lautgesetz*baums* `tree` eine *Liste* von Lautgesetzen zurück, die die Sprachentwicklung in `path` (von `make-path` erzeugt, s.o.) reflektiert. Von `(apply-rules rules words)` wird dann diese Liste `rules` auf die Wörter `words` angewandt.

Die Lautgesetzfunktionen werden von der Funktion `(s pattern . subst)` erzeugt, die aus einem String `pattern` einen regulären Ausdruck macht und eine Funktion zurückgibt, die `irregex-replace/all` auf diesen regulären Ausdruck und die Ersetzungen in der Liste `subst` anwendet. Matcht der reguläre Ausdruck einen Teil der Eingabe, wird diese durch `subst` ersetzt, dessen Elemente Strings, Zahlen als Indizes der Submatches, sowie Funktionen, die anhand eines Matches Strings zurückgeben, sind. `(irregex-replace/all "(foo)(bar)" "foobar" 2 (lambda (m) (string-reverse (irregex-match-substring m 1))) "quux")` z.B. würde den String `"foobar"` durch `"baroofquux"` ersetzen.

Die oben beschriebenen regulären Ausdrücke sind für die Modellierung von Laugesetzen nicht ausreichend. Neben anderen Unzulänglichkeiten, die sich im Laufe der Arbeit herausstellen werden, gibt es eindeutig Probleme, mit Lautklassen umzugehen,

und noch allgemeiner, Laute durch Zeichen zu kodieren.

Wenn jeder Laut durch genau ein Zeichen bzw. einen Unicode Codepoint kodiert wird, gibt es der Lautkodierung keine Probleme. Da es allerdings sinnvoll ist, einige Laute mit mehreren Codepoints zu kodieren, kann es zu Problemen kommen. Will man bspw. ein Lautgesetz `(s "b" "p")` formulieren, so hätte dies die Nebenwirkung, dass auch `bʰ` zu `pʰ` würde, was im allgemeinen Fall unerwünscht ist. Wenn man Laute wie b^b oder g^w mit einem einzigen Codepoint repräsentieren würde, hätte man dieses Problem natürlich zwar nicht, jedoch bietet Unicode für solche Vorhaben keine definierten Codepoints. Dafür müsste man die Private Use Area benutzen, wodurch die Kodierung jedoch von der Schriftart abhängig wird: eine unschöne Lösung. In der hier entwickelten Version von `lga` wird das Problem nicht gelöst. Stattdessen müssen die Lautgesetze vorläufig besonders vorsichtig formuliert werden, der obere Fall also als `(s "b([^ʰ])" "p" 1)`.

Bei der Formulierung von Lautgesetzen ist es ferner sinnvoll, Laute in Lautklassen zusammenzufassen um eine konzise und leicht abstrahierende Darstellung zu haben. Der POSIX Standard definiert zwar Zeichenklassen wie `[:digit:]` für Ziffern oder `[:lower:]` für Kleinbuchstaben, aber die Bedürfnisse, die man als Linguist hat, sind damit nicht abgedeckt, da sich zur Laufzeit Zeichenklassen weder neu definieren noch verändern lassen. Dieses Problem lässt sich vorläufig lösen, indem die regulären Ausdrücke, bevor sie von `string->irregex` in eine interne Form kompiliert werden, noch verändert werden, so dass die tatsächlich im Quelltext vorkommenden regulären Ausdrücke erst zu echten regulären Ausdrücken verarbeitet werden, die von der `irregex`-Engine verstanden werden. Konkret funktioniert dies so, dass in einer assoziativen Liste Lautklassennamen (als Symbole) mit Strings assoziiert werden und bei der Vorbearbeitung der regulären Ausdrücke Strings der Form `<lautklassenname>` durch den mit dem Symbol `lautklassenname` assoziierten String ersetzt werden. Die Funktion `(set-class key value)` assoziiert den Namen `key` mit dem String `value` und `(l key)` (kurz für *lookup*) findet den zu `key` passenden String in der assoziativen Liste.

So kann man bspw. mit `(set-class 'kurz-vok "a|e|i|o|u")` und `(set-class 'lang-vok "ā|ē|ī|ō|ū")` Kurz- und Langvokale definieren, mit `(set-class 'vok (s+ (l 'kurz-vok) "|" (l 'lang-vok)))` dann die Menge aller Vokale (`s+` ist eine Abkürzung für `string-append` zum Verketten von Strings) und diese in einem Lautgesetz wie `(s "<vok>" "a")` verwenden.

Die Lautklassen sind jedoch zur Laufzeit nicht vernünftig veränderbar, da sie zur Zeit der Auswertung von `s` in den regulären Ausdruck eingefügt werden, man die Definition aber wohl am ehesten aus dem Lautgesetzbaum heraus verändern wollen würde. Hier

wird nur eine neuentwickelte Lautgesetzengine wirklich Abhilfe schaffen können.

Es gibt jedoch noch ein weiteres Problem. So kann man nicht ohne weiteres z.B. einen Langvokal durch seinen entsprechenden Kurzvokal oder einen silbischen durch sein unsilbisches Pendant ersetzen. Beispiele für solche Vorgänge gibt es genug und es ist eine Methode notwendig, mit der man diese Fälle unkompliziert ausdrücken kann. In einer späteren Version wäre vielleicht eine Ersetzung wie `(s "<lang-vok>" "<kurz-vok>")` wünschenswert, in der aktuellen Version können jedoch auch Funktionen, die auf Submatches angewandt werden, denselben Effekt erzielen. Als Ersetzungsargumente kann man an `irregex-replace/all` neben Strings und Zahlen, die das entsprechende Submatch bezeichnen, auch Funktionen übergeben, die auf das aktuelle Match angewandt werden und einen String zurückgeben. Um also z.B. einen Langvokal in ein Kurzvokal zu verwandeln, braucht es nur eine Funktion, die auf ein Submatch eine Reihe von regulären Ausdrücken anwendet und das Ergebnis zurückgibt. Eine Funktion, die reguläre Ausdrücke auf einen String anwendet, gibt es ja schon: `apply-rules`. Diese wird von der Funktion `(match-rulelist rules)` benutzt, die eine Liste von Ersetzungsfunktionen `rules` auf ein Submatch anwendet. Dazu gibt sie eine Funktion zurück, die einen Submatchindex `i` bindet und eine Funktion zurückgibt, auf die `irregex-replace/all` angewandt werden kann. Die Vokalkürzung könnte somit folgendermaßen definiert werden:

```
(define kuerzung
  (match-rulelist
    (list (s "ā" "a")
          (s "ē" "e")
          (s "ō" "o")
          (s "ī" "i")
          (s "ū" "u"))))
```

Ein Lautgesetz, das alle Langvokale kürzt, könnte dann als `(s "(<lang-vok>)" (kuerzung 1))` formuliert werden.

Die eingangs erwähnten Regellisten zur Umwandlung von einer graphematisch/phonologischen in eine phonetische und von einer phonetischen in eine graphematisch/phonologische Darstellung sind unter ihrem Sprachnamen ebenfalls in assoziativen Listen abgelegt.

Hiermit ist die Funktionalität von `lga` weitestgehend erläutert. Eine genauere Beschreibung kann natürlich nur der Quelltext selber liefern.

3 Modellierung der Lautentwicklung des Altgriechischen

3.1 Vorbemerkungen

Vor der Beschreibung der Lautentwicklung noch einige Anmerkungen, die nicht speziell das Griechische betreffen. Aufgrund von Fehlern in der Implementation von `irregex` können in regulären Ausdrücken keine höheren Unicodezeichen in `[]`-Sets benutzt werden. Da diese aber notwendig sind, um gewisse Zeichen auszuschließen – insbesondere ʰ zur Unterscheidung von Aspiraten – müssen diese aus dem ASCII-Vorrat stammen. Aus dem Grund wird für die Aspiration `!` statt ʰ verwendet. Eigentlich wäre wie schon oben beschrieben für Aspiraten oder sonstige Laute, die mit mehreren Codepoints ausgedrückt werden, ein einziger Codepoint oder eine Möglichkeit, mehrere Codepoints als ein Zeichen zu behandeln, praktischer und angemessen, aber da ersterer Fall, wie ja schon angemerkt, nicht von Unicode abgedeckt wird, wäre man dann von der Schriftart abhängig, und der zweite Fall wird von `irregex` nicht unterstützt. Für die Labiovelare wurden im Gegensatz zu den Velaren Großbuchstaben verwendet, um dieses Problem zu vermeiden und auch die silbischen Resonanten *r̥ l̥ m̥ n̥* werden mit den Großbuchstaben `RLMN` ausgedrückt. Die Halbvokale *i̯ u̯* werden mit `yw` und die Laryngale *h*$_{[123]}$ mit `H[123]` bezeichnet. Akzente werden in der graphematischen Schreibweise als combining diacritics geschrieben und intern mit `'` und `~` bezeichnet.

Die Umwandlung dieser graphematischen Schreibungen in die interne (mehr oder weniger phonetische) Darstellung geschieht in der Liste der phonologischen Regeln für `uridg` (in `rules.scm`).

Als weitere (echte) phonologische Regeln für das Uridg. sind Laryngalumfärbung (**e* neben **h*$_3$ jedoch noch von **o* verschieden, was allerdings für das Gr. belanglos ist) und Stimmhaftwerdung von **s* neben stimmhaften Konsonanten beschrieben. Weitere phonologische Regeln müssen ggf. ergänzt werden. Lautgesetze wie Stangs Gesetz, die vor dem Erreichen des Sprachzustandes, der mit `uridg` bezeichnet wird, durchlaufen wurden, wurden nicht berücksichtigt und die Rekonstrukte bzw. Transponate, die als Eingabe dienen, sollten einen entsprechenden Lautstand vorweisen (um bei Stangs Gesetz zu beleiben wäre also **di̯ēm* nicht **di̯eu̯m* `uridg`). Insbesondere enthält `uridg` den *þ*-Laut, dessen Herkunft und Phonetik hier ebenfalls nicht weiter hinterfragt wird.

Im Folgenden werden die Lautgesetze vom Urindogermanischen ins Griechische in drei Etappen aufgestellt, wobei Rix als Grundlage dient. Zunächst wird die Lautentwicklung bis ins Urgriechische modelliert, dann bis und in mykenischer Zeit und schließlich in die Dialekte der alphabetischen Zeit.

Eine (relative) Chronologie aller Lautgesetze lässt sich nicht immer mit Sicherheit

aufstellen, da viele Lautgesetze kaum mit anderen Lautgesetzen interagieren oder mehrere Entwicklungen denkbar sind, so dass man gerade in der frühesten Zeit, in der das Griechische noch nicht belegt ist, in diesen Fällen wenig mehr als nur raten kann. Hier werden dann die Lautgesetze entweder mit ähnlichen anderen Gesetzen oder nach Gefühl einsortiert.

3.2 Urindogermanisch bis Urgriechisch

Die Lautveränderung, die ins Urgr. führen, sind grob gesprochen die Kentumvertretung der Tektale, **sT* als Reflex von **TT*, die Entwicklung von **þ*, jegliche Laryngalentwicklungen, die Entwicklung der silbischen Resonanten (teilweise aber auch erst dialektal), die Stimmloswerdung der Mediae aspiratae, einige Okklusivassimilationen, die Entwicklung von **s* und die Anfänge diverser Palatalisierungen.

1 Kentum Das Griechische ist eine Kentumsprache, es fallen also die uridg. Palatale mit den Velaren zusammen. (Rix §92-94)

2 *TT > sT In der Verbindung zweier Dentale wird der erste zu urgr. **s*. (Rix §106c)

3 *MA > TA Mediae aspiratae werden zu Tenues aspiratae (Rix §94). Aufgrund der Nähe zum Makedonischen und Phrygischen ist dieser Wandel vielleicht später anzusetzen.

4 *(H)i̯- Da es im Gr. scheinbar zwei Vertretungen von **i̯-* gibt, versucht man mit der Laryngaltheorie die beiden Reflexe als **Hi̯-* und **i̯-* zu unterscheiden. Welcher Anlaut zu urgr. **dz-* verschärft wurde, ist unklar – nach Rix §68,80e ist es **Hi̯-* – und nach dem Datenmaterial kann für beides argumentiert werden. Eine lautlich plausible Lösung ist frühe Verschärfung von uridg. **i̯-* zu urgr. **dz-* (vielleicht über Zusammenfall mit ererbtem **di̯-*, das ebenfalls zu **dz-* wird), danach Wegfall aller Laryngale (also keine Vokalisierung wie vor **u̯*), wodurch **Hi̯-* als **i̯-* stehenbleibt, wie es im Frühmyk. belegt ist.

5 Laryngale Die Entwicklung der Laryngale folgt Rix §79-85. Teilweise sind die Lautgesetze weniger explizit als bei Rix formuliert, um eine lautlich plausiblere Entwicklung zu modellieren. So wird z.B. intervokalischer Laryngal nach **i* oder **u* nicht zum Gleitlaut sondern dieser entsteht erst als Hiattilger nach dem Laryngalschwund.

6 *u̯eu̯ > u̯ei̯ *u̯eu̯ wird zu *u̯ei̯ dissimiliert, wie εἶπον < **e-u̯e-u̯kʷ-om* (vgl. ai. *avocam*) lehrt. Dies muss ferner vor 7 passieren, wie εἶπον ebenfalls zeigt.

7 Labiovelar > Velar Neben *ŭ̯, *u̯ und vor *i̯ fallen die Labiovelare mit den Velaren zusammen (Rix §97).

8 Silbische Resonanten Die Entwicklung der silbischen Resonanten folgt Rix §75 und §76. Für das Urgr. wird ein Sprossvokal *ə angesetzt, welcher sich dialektal zu *a* oder *o* entwickelt. Genauere Datierung unklar.

9 Nasal vor Okklusiv Vor Okklusiven wird der homorgane Nasal realisiert (Rix §78). Gegen Rix jedoch nicht **ms* > **ns* wegen ἔνειμα < **enemsa*. Hier kann *m* zwar analog wieder eingeführt worden sein, aber ohne ein Beispiel, das bei Rix fehlt, ist die Regel zunächst unnötig. Die Nasalassimilation passiert auch später noch und dürfte tatsächlich wohl über lange Zeit eine synchrone phonolgische Regel sein.

10 *m > *n \ *i̯ Rix §77. Datierung unklar. Vielleicht nach *m̥ > **am*, aber *m̥ > *n̥ \ *i̯ ist ebenfalls als zusätzliche Regel denkbar.

11 Benachbarte Vokale nach Laryngalschwund (1. Kontraktion) Kontraktion von **e*, **a*, **o*, zwischen denen Laryngal geschwunden ist (Rix §81). Datierung unklar.

Nach Vokal werden **i*, **u* zu Halbvokalen (es entstehen also Diphthonge). Im umgekehrten Fall entsteht ein Gleitlaut zwischen den beiden Lauten.

12 Thorn **þ* (genauer Lautwert unklar) tritt nach Velaren und Labiovelaren auf. Nach Tenues ist entsprich *t*, nach Aspirata t^h (Rix §81).