Notes toward a complete account of the Loglan language

Randall Holmes

August 24, 2013

Version Notes

8/24/2013 5:45 PM: Reformatted the file. It now includes the whole PEG (though some of it which hasnt been commented yet runs off the page – I will fix this as I add comments). The commentary goes up to the end of phonology of predicates as before, but includes the new phonology of names.

It will be easier to add comments to this version. Expect progress!

8/15/2013, 10:35 AM fixed the CVCCCV issue with proposal 1

8/11/2013, 11:20 AM: first version posted.

1 Introduction; PEGs Discussed

The intention of this document is to present a full account of the formal aspects of the Loglan language as implemented in the proposed PEG in the form of a line by line commentary on the actual commands in the PEG. It should be noted that in the course of this design I made choices which differ from those implicit in our official definition; I will highlight these as I go. This document has various proposals for changes implicit in it.

Since this document contains PEG rules we give a brief guide. Looking at Bryan Ford's paper might be good. Pieces of PEG notation denote sets of strings occurring in the context of a larger string (what we are really talking about is a set of substrings of a given context string, where the information specifying a substring includes its location in the larger string).

literal strings: `'a literal string'` is a name for a class of strings consisting of just that string.

character classes: `[aeiou]` is a name for the class of one-character strings which consist of one vowel. `[a-z]` is a name for the class of one-character strings which consists of one lower-case letter. `[a-zA-Z]` is the class of one-character strings consisting of an upper case or lower case letter. A period `.` represents the class of one-character strings.

names of classes: An unadorned string `name` is the name of a class: it will be defined by a PEG line of the form `name <- <definition>`. Definitions can be recursive (and if you aren't careful, they can lead to infinite loops, which will cause the parser to hang; my parser generator does a check which guarantees termination, though it accepts rules that fail this check, and it is well-known that such a test cannot be exact).

sequence: `(name1 name2 name3)` defines the class of strings which consist of a string from `name1` followed by a string from `name2` followed by a string from `name3`. The parentheses are optional. I'm not going to try to describe order of operations here.

optional: `name?` represents a string of class `name` or an empty string not followed by a string of class `name` (notice the context dependence). So `name1 name2? name3` represents a sequence `name1 name2 name3` in which the `name2` component is optional: note that if it takes the shape `name1 name3` there will not be a string of class `name2` beginning at the same place as the string of class `name3`.

repetition: `name*` represents a sequence of zero or more strings of class `name` not followed by a string of class `name`. `name+` represents a sequence of one or more strings of class `name` not followed by a string of class `name`.

logical conditions: `!name` represents an empty string not followed in the context by a string of class `name`. `&name` represents an empty string which is followed in the context by a string of class `name`. Thus for example `!name1 name2` represents a string of class `name2` which does not have an initial segment or is not itself an initial segment of a string of class `name1` in the context, and `name1 !name2` represents a string which is of class `name1` and not followed by a string of class `name2` in the context. Logical conditions allow a very flexible way of expressing context dependence. `!.` represents an empty string at the end of the context (end of file).

priority: `(name1/ name2 /name3)` represents the class of strings which are either of class `name1` or of class `name2` and don't start at the same point as a string of class `name1` in the context, or of class `name3` and don't start at the same point as a string of class `name1` or `name2` in the context; the idea is that in the list of alternatives one must choose the one which occurs first.

An advantage of PEGs is that they are automatically unambiguous. The language of PEGs is quite powerful and expressive. The difficulty with PEGs is making sure that in priority or option situations one is actually always taking the intended alternative. PEGs can be bad not in being ambiguous but in making unintended readings of strings. Situations in which this is likely to be happening can be formally described, and I am trying to develop a tool which will warn a grammar writer about these situations; it is very tricky.

2 Loglan Phonology

1. The letters used in Loglan are the usual 26 upper-case and lower-case letters of the Latin alphabet. There is a proposal afoot to remove the aberrant letters **q, w, x** from the alphabet. Conventions on the use of upper-case and lower-case letters will be discussed below.

```
##Loglan phonology

lowercase <- [a-z]
```

2.
```
uppercase <- [A-Z]
```

3.
```
letter <- [A-Za-z]
```

4.
```
V <- [AEIOUWYaeiouwy]
```

5. The class `V` of all Loglan vowels (including the special **y** and the aberrant **w**) is not actually used in the grammar.

   The class `V2` includes the usual Loglan vowels, which have the standard values common in European languages other than English.

   y is pronounced with the schwa sound.

   w has the sound of **ü** in German. There is a move afoot to eliminate it, and in fact it cannot be used in this grammar in any Loglan word.

```
V2 <- [AEIOUaeiou]
```

6. The class `C` as defined here includes the aberrant **q** and **x**, which are not actually expected to occur in Loglan words as a rule. The value of **c** is that of **sh** in English. The value of **j** is the voiced analogue of the value of **sh**, which occurs in English but does not have a standardized orthography. The value of the aberrant `q` is the unvoiced value of **th** in English; the value of **x** is the value of **ch** in Scottish "loch". The values of other consonants may be taken to be as expected in English with the following remarks: **n** undergoes the same transformation before **g** and **k** that it does in English;

g never takes the soft value of English **j**. **m,n,l,r** may all under certain circumstances be vocalic (as they can be in English).

The aberrant consonants **q** and **x** may eventually be removed.

```
C <- (!(V) letter)
```

7. This rule describes the pairs of vowels which may be pronounced as monosyllables. The pairs **ao** (pronounced like `ow` in English), **ai** (the value of long **i** in English) , **ei** (the value of long **a** in English) and **oi** (pronounced as in English) will always be monosyllables when paired in a word or word unit. The pairs beginning with **i** or **u** may be pronounced as monosyllables (**i** and **u** becoming consonantal **y** **w** respectively) or as disyllables when paired in a word unit. All other pairs of vowels will be pronounced as disyllables when paired in a word or word unit.

   This parser does not distinguish optional monosyllables from obligate monosyllables; it always chooses monosyllable readings of pairs of vowels where it can. This does not mean that the disyllable pronunciation is deprecate; it appears however that the structure of the language sometimes forces a monosyllabic pronunciation of an optional monosyllable but never forces a disyllable pronunciation.

```
Mono <- ('ao' / (V2 [i]) / ([Ii] V2) / ([Uu] V2))
```

2.1 Phonology of Structure Words

8.
```
CVVSyll <- (C Mono)
```

9. This rule implements units from which little words are built in rule `LW` below. This implements phonetic rules for compound little words given in NB3.

```
LWunit <- ((V2 V2) / (CVVSyll V2) / (C V2 V2) / (C V2))
```

10. This defines the phonetic units (other than single `V2`) from which compound structure words are built. Pairs of vowels appearing within such units are mono- or di-syllables according to the rules (and options) given above.

```
LW1 <- ((V2 V2) / (C V2 V2) / (C V2))
```

11. This rule imposes the usual capitalization convention on a block of letters that it precedes; the letters will either all be lower case or the first will be upper case and all the rest lower case.

```
caprule <- ((uppercase / lowercase) (lowercase)* !(letter))
```

12. This implements the description of phonetic structure of compound little words in NB3. It is important to note that this rule plays no significant role in the grammar.

```
LW <- (&(caprule) (((V2)? (LWunit)+) / V2))
```

2.2 Phonology of Syllables and Names

The rules for syllable formation that I give involve some arbitary decisions of my own. They were originally developed for application to borrowings, and they do appear to parse all existing borrowings. I have imposed them on names with the modification that **y** can be used as a vowel freely in names as it cannot in borrowings.

13. This is the list of pairs of consonants which may occur at the beginnings of syllables (initial pairs). A sequence of three consonants may appear at the beginning of a syllable iff each successive pair of consonants in the triple is an initial pair. The pair **sv** was added in Appendix H, and we restored the pair **zl** because it occurs in a composite primitive. Note support for capitalization.

```
InitialCC <- ('bl' / 'br' / 'ck' / 'cl' / 'cm' / 'cn' / 'cp' / 'cr' / 'ct' / 'dj'
/ 'dr' / 'dz' / 'fl' / 'fr' / 'gl' / 'gr' / 'jm' / 'kl' / 'kr' / 'mr' / 'pl' / 'pr'
 'sk' / 'sl' / 'sm' / 'sn' / 'sp' / 'sr' / 'st' / 'tc' / 'tr' / 'ts' / 'vl' / 'vr' /
'zb' / 'zv' / 'zl' / 'sv' / 'Bl' / 'Br' / 'Ck' / 'Cl' / 'Cm' / 'Cn' /
 'Cp' / 'Cr' / 'Ct' / 'Dj' / 'Dr' / 'Dz' / 'Fl' / 'Fr' / 'Gl' / 'Gr' / 'Jm' /
 'Kl' / 'Kr' / 'Mr' / 'Pl' / 'Pr' / 'Sk' / 'Sl' / 'Sm' / 'Sn' / 'Sp' / 'Sr' /
 'St' / 'Tc' / 'Tr' / 'Ts' / 'Vl' / 'Vr' / 'Zb' / 'Zv' /
 'Zl' / 'Sv')
```

14. This is the set of pairs of consonants which may not appear adjacent to one another (as at a juncture between syllables).

```
NonmedialCC <- ('bb' / 'cc' / 'dd' / 'ff' / 'gg' / 'hh' / 'jj' / 'kk'
/ 'll' / 'mm' / 'nn' / 'pp' / 'qq' / 'rr' / 'ss' / 'tt' / 'vv' / 'xx' / 'zz'
/ ([h] C) / ([cjsz] [cjsz]) / 'fv' / 'kg' /
 'pb' / 'td' / ([fkpt] [jz]) / 'bj' / 'sb')
```

15. This is the set of triples of consonants which may not appear adjacent to one another (as at a juncture between syllables).

```
NonjointCCC <- ('cdz' / 'cvl' / 'ndj' / 'ndz' / 'dcm'
/ 'dct' / 'dts' / 'pdz' / 'gts' / 'gzb' / 'svl' / 'jdj' / 'jtc'
/ 'jts' / 'jvr' / 'tvl' / 'kdz' / 'vts' / 'kdz' / 'vts' / 'mzb')
```

16. This is the set of repeated vowels to which the stress rule applies: if a repeated vowel of one of these forms appears without an intervening pause, one of them must carry the primary stress of the word in which they appear (which implies that such pairs cannot appear without an intervening pause in two adjacent words).

```
RepeatedVowel <- ('aa' / 'ee' / 'oo' / 'Aa' / 'Ee' / 'Oo')
```

17. A context which signals that a consonant is being used vocalically (a joint for making borrowings non-complexes). This is the only use of vocalic consonants which I allow in borrowings. (the same restrictions on vocalic consonants are placed on names, just because it is easier).

```
RepeatedVocalic <- ('mm' / 'nn' / 'll' / 'rr' / 'Mm' / 'Nn' / 'Ll' / 'Rr')
```

18. A set of initial consonants in a syllable. There may one to three consonants in the sequence, each adjacent pair being an initial pair, and the final one may not make a vocalic consonant pair with the following letter. The first version is for use in borrowings. The second version is for use in names, allowing **y** as a vowel.

```
FirstConsonants <- ((!((C C RepeatedVocalic)) &(InitialCC)
 (C InitialCC)) / (!((C RepeatedVocalic)) InitialCC) /
 ((!(RepeatedVocalic) C) !([y])))
```

19.
```
FirstConsonants2 <- ((!((C C RepeatedVocalic)) &(InitialCC) (C InitialCC)) /
 (!((C RepeatedVocalic)) InitialCC) / (!(RepeatedVocalic) C))
```

20. The vowel in a syllable; this will be a monosyllable vowel pair, a single vowel, or the first consonant of a vocalic consonant pair.

```
VowelSegment <- (Mono / V2 / (&(RepeatedVocalic) C))
```

21. A syllable. It consists of an optional set of first consonants, followed by a vowel segment (mandatory of course) followed by up to two optional final consonants (the rule defininig these is just below). This notion of syllable was originally developed for borrowings; a second version is given below for names. In structure words and non-borrowing parts of complexes, a class of word units which includes VV and CVV disyllables is the correct class of segments.

    My rules for forming syllables involve actual decisions which I made myself. They do appear to parse all existing borrowings. The design decisions I made myself were to allow no more than three consonants in an initial sequence and no more than two final consonants, and a preference for attaching consonants to the next syllable if possible.

    It is important to note that the intention here is not to mandate the actual syllabification presented in the parse. The existence of a pronounceable syllabification is what is established by the parse.

```
Syllable <- ((FirstConsonants)? VowelSegment (FinalConsonant)? (FinalConsonant)?)
```

22. Here is the notion of final consonant used in the syllable rule above; notice the recursive dependency on the syllable rule. A final consonant cannot form a forbidden medial consonant sequence with what follows, nor can it form a syllable with what follows (this parser prefers vowel-final syllables and will make them when it can).

```
FinalConsonant <- (!(NonmedialCC) !(NonjointCCC) !(Syllable) C)
```

23. This version of a syllable allows **y** as a vowel and is used in names.

```
Syllable2 <- ((FirstConsonants2)? (VowelSegment / [y])
(FinalConsonant2)? (FinalConsonant2)?)
```

24. Here is the notion of final consonant used in the syllable rule above (version for names); notice the recursive dependency on the syllable rule. A final consonant cannot form a forbidden medial consonant sequence with what follows, nor can it form a syllable with what follows (this parser prefers vowel-final syllables and will make them when it can).

```
FinalConsonant2 <- (!(NonmedialCC) !(NonjointCCC) !(Syllable2) C)
```

25. The rule for name words. A sequence of syllables (with **y** allowed as a first-class vowel), ending with a consonant followed by either a comma (included in the name) or a punctuation mark or end of file not included in the name.

    The use of vocalic consonants in names is quite arbitrarily restricted to work just as in borrowings. So "Earl" becomes "`Rrl`". This is easier for the grammar writer and appears harmless.

```
Name <- (([ ])* &(((uppercase / lowercase) ((&((lowercase lowercase))
 lowercase))* C (!(.) / ',' / &(period)))) ((Syllable2)+ (!(.) / ',' / &(period)))
 !((([ ])* [,])))
```

2.3 Phonology of Word Units and Predicates

One of the nastiest bits of the Loglan grammar in the largest sense. "Word unit" usually means "affix" in the weird Loglan sense, but a primitive consists two word units. Word units can be disyllables. Recall that structure words consist of similar phonetic units which are not exactly syllables. Here complexes are made up of "word units", whereas borrowings are analyzed into syllables.

26.
```
CV <- (C V2 !(V2))
```

27. A final consonant of a word unit; it will either be followed by a **y** hyphen or will be followed by a consonant with which it will not make a forbidden medial pair or triple of consonants.

```
Cfinal <- ((C [y]) / (!(NonmedialCC) !(NonjointCCC) C !(V2)))
```

28. A CVV word unit in which the two vowels form a repeated vowel pair on which stress must be placed. The final bit (repeated in a couple of rules below) describes the option of a hyphen to fix its joint with the following syllable: this will not form a forbidden medial consonant sequence, and it will be either an **r** not followed by an **r** or a vowel, or an **n** followed by an **r**, or a **y**.

```
CVVStressed <- (C RepeatedVowel !(RepeatedVowel)
 ((!(NonmedialCC) !(NonjointCCC) (([r] !([r]) !(V2)) / ([n] &([r])) / [y]))?)
```

29. A CVV word unit which must form a disyllable. In this parser, I always presume that I form monosyllables where I can. I have found places where monosyllable readings of vowel pairs are forced; I have found no place in the grammar where one must read an optional disyllable as a disyllable. There is such a place in the current official definition, but it is excluded here: the official definition now allows CCVV borrowings (though there aren't any) and in such a borrowing the VV must be a disyllable. There are proposals which I believe will pass which again banish CCVV borrowings, and I repeat there aren't any in the dictionary. This parser implements one of these proposals.

```
CVVDisyllable <- (C !(Mono) V2 V2 ((!(NonmedialCC) !(NonjointCCC)
 (([r] !([r]) !(V2)) / ([n] &([r])) / [y]))?)
```

30. A CVV word unit. I forbid a CVV word unit to be followed by a vowel in such a way that the repeated vowel stress rule comes into play, so as to force hyphenation when a CVV affix is followed by a vowel-initial borrowing in a complex in a way which would otherwise invoke the stress rule.

```
#The restriction on repeated vowels prevents a CVV affix from forcing stress on the

CVV <- (C V2 !(RepeatedVowel) V2 ((!(NonmedialCC) !(NonjointCCC)
(([r] !([r]) !(V2)) / ([n] &([r])) / [y])))?)
```

31. A CVC word unit. Note that `Cfinal` includes the facilities for hyphenation of such units.

```
CVC <- (C V2 Cfinal)
```

32. A CCV word unit. Once again, I forbid formation of a repeated vowel pair to which the stress rule applies with a following vowel, presumably the initial vowel of a borrowing in a complex, and allow the option of a **y** hyphen.

```
#The repeated vowel rule and option of a y-hyphen here are strictly to deal with fol

CCV <- (InitialCC V2 !(RepeatedVowel) ([y])?)
```

33. A full primitive appearing as an affix, either the entire form appearing at the end of a word or the **y**-final form not appearing at the end of a word. (either of the next two rules).

```
CCVCV <- (CCV ((CV !(letter)) / (C [y] &(letter))))
```

34.
```
CVCCV <- (C V2 !(NonmedialCC) C ((CV !(letter)) / (C [y] &(letter))))
```

35. A syllable beginning with two or three consonants.

```
CCSyllable <- ((&(C) C) FirstConsonants (Mono / V)
(FinalConsonant)? (FinalConsonant)?)
```

36. A syllable forming a consonant pair with the following syllable (or possibly ending with a pair of consonants).

```
CCFinalSyllable <- ((FirstConsonants)? VowelSegment FinalConsonant
&((C / ([y] C))) (FinalConsonant)?)
```

37. A syllable with a vocalic consonant as vowel segment; this of course forms a consonant pair.

```
CCSyllableB <- ((FirstConsonants)? &(RepeatedVocalic) C
(FinalConsonant)? (FinalConsonant)?)
```

38. A syllable which forms a repeated vowel to which the stress rule applies with a following vowel.

```
SyllableVV <- ((FirstConsonants)? &(RepeatedVowel) V2)
```

39. I don't think this is used.

```
Syllables <- (Syllable)+
```

40. A CVV or CV sequence; one reads CVV if one can.

```
CW1 <- ((C V2 V2) / (C V2))
```

41. A class of sequences not allowed at the beginning of borrowings. Sequences of four vowels are not permitted (this contravenes NB3 which allows arbitrarily long initial strings of vowels; there are no strings of more than two vowels at the beginning of a borrowing in the dictionary). A vowel or two before a CW1 is not permitted. A CW1 followed by two vowels is not permitted (again, NB3 allows C followed by an arbitrarily long string of vowels; I restrict such a string of vowels to three). Two CW1s are not permitted. The idea is to prevent the peeling of a structure word off the front of a borrowing; my rule is more conservative than the NB3 rule.

```
BadSequence <- ((V2 V2 V2 V2) / (V2 (V2)? CW1) / (CW1 (V2 V2)) /
 (CW1 CW1))
```

42. A syllable not causing a repeated vowel stress with the following syllable.

```
#Provisionally forbidding repeated vowels in borrowings altogether

NoBadstress <- (!(SyllableVV) Syllable)
```

43. The specification of a borrowing. It is horrible. CCVV borrowings are specifically excluded (my proposal 1). A borrowing may not be a name (it will not be consonant final). It will not start with a `BadSequence`. It will not begin with a vowel followed by an initial pair followed by a vowel, because this may allow the initial vowel to be peeled off (`iglu` found in NB3 is excluded; this is in Appendix H). A word unit (V, VV, CV, CVV) may not appear as the initial segment followed by a consonant initial predicate (**tosmabru** test) [of course the initial unit would fall off as a structure word]. The first syllable may not have a vocalic consonant (this makes sense, and also saves **hidrroterapi** in the dictionary). It will be a sequence of at least two syllables in which no repeated vowels to which the stress rule applies occur. It will contain at least one CC pair.

    Note that this is in line with my proposal 1: the `slinkui` and `aslinkui` tests are not used, but CCVV is excluded so that **y** hyphens are not forced in two-affix words with initial CVC forming an initial pair with the following syllable re the decision in Appendix H. I do know how to install the **slinkui** test if it is restored.

```
Borrowing <- (!((C C V2 V2 !(letter))) !(Name) !(BadSequence)
!((V2 InitialCC V2)) !(((CW1 / (V2 V2) / V2) (!(V2) Predicate)))
!(CCSyllableB) &((NoBadstress (NoBadstress)+))
((!((CCSyllable / CCSyllableB / CCFinalSyllable)) Syllable))* (Syllable)+)
```

44. This defines the class of affixes from which complexes are built. We go beyond the scope of NB3 in implementing the decision outlined in Appendix H to use borrowings in full (with following **y** hyphen unless final) as affixes. The repeated vowel stress rule is enforced by not allowing CVV affixes with a repeated vowel to have multisyllable forms following them. CVC affixes are of course barred from being final, which is achieved by requiring that they be followed immediately by a letter, and requiring that complexes be followed by a non-letter below.

    John Cowan tells us that we should call these “combining forms”, not “affixes”.

```
Affix <- ((Borrowing (([y] &(letter)) / !(letter))) / CCVCV / CVCCV /
CCV / (!((CVVStressed Affix Affix))
 !((CVVStressed (CVVDisyllable / CCVCV / CVCCV / Borrowing))) CVV) /
(CVC &(letter)))
```

45. A precomplex is simply a sequence of two or more affixes. The additional rules imposed on complexes proper are that they do not have an initial CV or CVV that falls off (**tosmabru** test) and initial CVC affixes which would form an initial pair with the following affix must be **y**-hyphenated except in the cases CVCCVV or CVCCCV (proposal in Appendix H eliminating the **slinkui** test as modified by my Proposal 1).

```
PreComplex <- (Affix (Affix)+)
```

46.
```
#I have installed the stress rules for repeated vowel syllables in complexes

Complex <- (!((C V2 (V2)? (PreComplex / CVV)))
!((C V2 (!((C C (V2 / C) V2 !(letter))) InitialCC))) PreComplex !(letter))
```

47. A primitive is familiar to any Loglanist.

```
Primitive <- ((CCVCV / CVCCV) !(letter))
```

48. A predicate word is a primitive, borrowing or complex. Note that primitives and borrowings are not a special case of complexes, as a complex is defined as containing more than one affix. Note that a predicate cannot be followed by another letter; the word space, punctuation or end of utterance at the end of a predicate word must appear; in phonetic terms, it signals the presence of a stress.

```
Predicate <- (&(caprule) (Primitive / Complex / Borrowing) !(letter))
```

3 Lexing Structure Words and Forms Involving Non-Loglan Text

49.

```
##The lexer, so to speak

__LWinit <- (([ ])* !(Predicate) &(caprule))
```

50.
```
#Used to prevent CV cmapua from absorbing a following vowel

Oddvowel <- (((V2 V2))* V2 !((((!(Predicate) ('ma') / 'fi'))))
```

51.
```
A0 <- ((([aAEOUeou] !([IAEOUaeiou])) / (__LWinit 'ha') / (__LWinit 'Ha'))
 !(Oddvowel))
```

52.
```
A <- (__LWinit !(Predicate) (('no' / 'No'))?
 __LWinit ((('nu' / 'Nu') &([u])))? A0 ('noi')? (PA)? ('noi')? !(Oddvowel))
```

53.
```
__LWbreak <- (!(Oddvowel) !((A / ICI / ICA / IGE / I))
 ((', ' !((([ ])* [,])) !((([ ])* Predicate))))?)
```

54.
```
A1 <- (A __LWbreak)
```

55.
```
A2 <- (A __LWbreak)
```

56.
```
A4 <- (A __LWbreak)
```

57.
```
A3 <- (A __LWbreak)
```

58.
```
ACI <- (A __LWinit 'ci' __LWbreak)
```

59.
```
AGE <- (A __LWinit 'ge' __LWbreak)
```

60.
```
BI <- (__LWinit ('bia' / 'bie' / 'cie' / 'cio' / 'Bia' / 'Bie' / 'Cie' / 'Cio' / 'Bi
 __LWbreak)
```

61.
```
CA0 <- ((__LWinit 'ca') / 'ce' / 'co' / 'cu' / 'ze' / 'Ca' / 'Ce' / 'Co' / 'Cu' / (
```

62.
```
CA1 <- (__LWinit (('no' / 'No'))? ((('nu' / 'Nu') &([\'cu])))? CA0 ('noi')? (PA)? (
```

63.
```
CA <- (__LWinit &(caprule) CA1 __LWbreak)
```

64.
```
CI <- (__LWinit ('ci' / 'Ci') __LWbreak)
```

65.
```
CUI <- (__LWinit ('cui' / 'Cui') __LWbreak)
```

66.
```
NI0 <- (__LWinit ('kua' / 'gie' / 'giu' / 'hie' / 'hiu' / 'kue' / 'Kua' / 'Gie' / 'G
```

67.
```
NI1 <- (__LWinit NI0 (('mo' / 'ma'))* !(Oddvowel))
```

68.
```
RA <- (__LWinit ('ra' / 'ri' / 'ro' / 'Ra' / 'Ri' / 'Ro') !(Oddvowel))
```

69.
```
NI <- (__LWinit ('ie')? __LWinit ((((RA / (NI1 &(NI1))))* NI1) / RA) ('cu')? !(Oddvo
```

70.
```
DA0 <- (__LWinit ('tao' / 'tio' / 'tua' / 'Tao' / 'Tio' / 'Tua' / 'mio' / 'miu' / 'm
```

71.
```
TAI0 <- (((V !(Predicate) 'ma') / (V !(Predicate) 'fi') / (C 'ai') / (C 'ei') / (C '
```

72.
```
TAI <- (__LWinit TAI0 __LWbreak)
```

73.
```
DA1 <- (__LWinit (TAI0 / DA0) (('ci' NI))? !(Oddvowel))
```

74.
```
DA <- (__LWinit DA1 __LWbreak)
```

75.
```
DIE <- (__LWinit ('die' / 'fie' / 'kae' / 'nue' / 'rie' / 'Die' / 'Fie' / 'Kae' / 'N
```

76.
```
DIO <- (__LWinit ('beu' / 'cau' / 'dio' / 'foa' / 'kao' / 'jui' / 'neu' / 'pou' / 'g
```

77.
```
DJAN <- Name
```

78.
```
END <- [.]
```

79.
```
FI <- ('fi' __LWbreak)
```

80.
```
LIO <- (('lio' / 'Lio') __LWbreak)
```

81.
```
GA <- (__LWinit ('ga' / 'Ga') __LWbreak)
```

82.
```
GE <- (__LWinit ('ge' / 'Ge') __LWbreak)
```

83.
```
GE2 <- (__LWinit ('ge' / 'Ge') __LWbreak)
```

84.
```
#Put in the new word geu, but the old word cue is still supported

GEU <- (__LWinit ('cue' / 'Cue' / 'geu' / 'Geu') __LWbreak)
```

85.
```
GI <- (__LWinit ('gi' / 'goi' / 'Gi' / 'Goi') __LWbreak)
```

86.
```
GO <- (__LWinit ('go' / 'Go') __LWbreak)
```

87.
```
GU <- ((__LWinit ('gu' / 'Gu') __LWbreak) / (',' ([ ])+ !([,])))
```

88.
```
GUI <- (__LWinit ('gui' / 'Gui') __LWbreak)
```

89.
```
GUO <- (__LWinit ('guo' / 'Guo') __LWbreak)
```

90.
```
GUU <- (__LWinit ('guu' / 'Guu') __LWbreak)
```

91.
```
GUU1 <- (__LWinit ('guu' / 'Guu') __LWbreak)
```

92.
```
GUU2 <- (__LWinit ('guu' / 'Guu') __LWbreak)
```

93.
```
HOI <- (__LWinit ('hoi' / 'Hoi') __LWbreak)
```

94.
```
ICA <- (__LWinit !(Predicate) ('i' / 'I') CA1 __LWbreak)
```

95.
```
ICI <- (__LWinit ('i' / 'I') (CA1)? __LWinit 'ci' __LWbreak)
```

96.
```
IE <- (__LWinit ('ie' / 'Ie') __LWbreak)
```

97.
```
IGE <- (__LWinit ('i' / 'I') (CA1)? __LWinit 'ge' __LWbreak)
```

98.
```
JE <- (__LWinit ('je' / 'Je') __LWbreak)
```

99.
```
JI <- (__LWinit ('jie' / 'jae' / 'pe' / 'ji' / 'ja' / 'Jie' / 'Jae' / 'Pe' / 'Ji' /
```

100.
```
JIO <- (__LWinit ('jio' / 'jao' / 'Jio' / 'Jao') __LWbreak)
```

101.
```
JO <- (__LWinit ('jo' / 'Jo') __LWbreak)
```

102.
```
JUE <- (__LWinit ('jue' / 'Jue') __LWbreak)
```

103.
```
KA0 <- (('ka' / 'ke' / 'ko' / 'ku' / 'Ka' / 'Ke' / 'Ko' / 'Ku') !(Oddvowel))
```

104.
```
KOU <- (('kou' / 'moi' / 'rau' / 'soa' / 'Kou' / 'Moi' / 'Rau' / 'Soa') !(Oddvowel)
```

105.
```
KA <- (__LWinit &(caprule) __LWinit (('no' / 'No'))? __LWinit ((('nu' / 'Nu') &([\'k
```

106.
```
KOUKI <- (__LWinit &(caprule) (('nu' / 'Nu'))? __LWinit (('no' / 'No'))? __LWinit K0
```

107.
```
KA3 <- (KA / KOUKI)
```

108.
```
KA1 <- KA3
```

109.
```
KA2 <- KA3
```

110.
```
KI <- (__LWinit ('ki' / 'Ki') ((!((KOU / 'nu')) PA))? ('noi')? __LWbreak)
```

111.
```
KIE <- (__LWinit ('kie' / 'Kie') __LWbreak)
```

112.
```
KIU <- (__LWinit ('kiu' / 'Kiu') __LWbreak)
```

113.
```
LAO <- (__LWinit ('lao' / 'Lao') __LWbreak)
```

114.
```
LAU <- (__LWinit ('lau' / 'lou' / 'Lau' / 'Lou') __LWbreak)
```

115.
```
Quotemod <- ('za' / ('zi' !(Oddvowel)))
```

116.
```
LI <- (__LWinit ('li' / 'Li') (Quotemod)? [ ] (' luli' / (!([ ] 'lu')) .))* [ ]
```

117.
```
stringnospaces <- (([ ])+ ((![ ]) .)+)
```

118.
```
LIE <- (([ ])* ('lie' / 'Lie') ((![ ]) NI))? (Quotemod)? stringnospaces ((([ ])* (
```

119.
```
LIU <- (__LWinit ('liu' / 'lii' / 'niu') __LWbreak)
```

120.
```
LIU1 <- (__LWinit (('liu' / 'Liu') (Quotemod)? ([ ])+ (Predicate / Name / LW) __LWb
```

121.
```
LUA <- (__LWinit ('lua' / 'luo' / 'Lua' / 'Luo') __LWbreak)
```

```
SOI <- (__LWinit ('soi' / 'Soi') __LWbreak)
```

```
ME <- (__LWinit ('mea' / 'Mea' / 'me' / 'Me') __LWbreak)
```

```
LEPO <- (__LWinit ('le' / 'lo' / 'Le' / 'Lo' / NI0 / RA) PO1 __LWbreak)
```

```
NO1 <- (__LWinit ('no' / 'No') __LWbreak)
```

```
NU0 <- (('nuo' / 'fuo' / 'juo' / 'Nuo' / 'Fuo' / 'Juo' / 'nu' / 'fu' / 'ju' / 'Nu' /
```

```
mex <- (__LWinit NI __LWbreak)
```

```
NU <- (__LWinit ((NU0 (NI)?))+ __LWbreak)
```

```
PA0 <- (__LWinit ('gia' / 'gua' / 'pia' / 'pua' / 'Gia' / 'Gua' / 'Pia' / 'Pua' / 'n
```

```
PA <- (__LWinit (('no' / 'No'))? __LWinit ((!(PA0) NI))? ((('nu' / 'Nu') &(KOU)))? (
```

```
PA2 <- (__LWinit PA __LWbreak)
```

```
PA3 <- PA2
```

```
PA4 <- PA2
```

```
LE <- (__LWinit ('lea' / 'leu' / 'loe' / 'lee' / 'laa' / 'Lea' / 'Leu' / 'Loe' / 'Le
```

135.
```
LA <- (__LWinit ('la' / 'La') ((DA1 / TAI0))? (PA)? __LWbreak)
```

136.
```
I <- (__LWinit ('i' / 'I') (PA)? __LWbreak)
```

137.
```
GA2 <- (__LWinit ('ga' / 'Ga') __LWbreak)
```

138.
```
PA1 <- ((PA2 / GA) __LWbreak)
```

139.
```
CANCELPAUSE <- (('y,' ([ ])+) / ('cuu' __LWbreak))
```

140.
```
PAUSE <- ([,] ([ ])+ !(CANCELPAUSE) !([,]))
```

141.
```
PO1 <- (__LWinit ('po' / 'pu' / 'zo' / 'Po' / 'Pu' / 'Zo') !(Oddvowel))
```

142.
```
PO <- (__LWinit PO1 __LWbreak)
```

143.
```
PREDA <- (([ ])* &(caprule) ('he' / 'dua' / 'dui' / 'bua' / 'bui' / 'He' / 'Dua' / '
```

144.
```
HUE <- (__LWinit ('hue' / 'Hue') __LWbreak)
```

145.
```
SUE <- (__LWinit ('sue' / 'sao' / 'Sue' / 'Sao') ([ ])+ (letter)* __LWbreak)
```

146.
```
UI0 <- (('ua' / 'ue' / 'ui' / 'uo' / 'uu' / 'oa' / 'Ua' / 'Ue' / 'Ui' / 'Uo' / 'Uu'
```

147.
```
UI <- (__LWinit ((UI0 UI0) / UI0 / (PA 'hu') / (NI 'fi')) __LWbreak)
```

148.
```
ZE2 <- (__LWinit ('ze' / 'Ze') __LWbreak)
```

149.
```
ZI <- ('zi' / 'za' / 'zu')
```

150.
```
##The implementation of trial.85

guo <- ((GUO / GU) (freemod)?)
```

151.
```
gui <- ((GUI / GU) (freemod)?)
```

152.
```
GUE <- 'gue'
```

153.
```
gue <- ((GUE / GU) (freemod)?)
```

154.
```
guu <- ((GUU1 / GU) (freemod)?)
```

155.
```
lua <- LUA
```

156.
```
geu <- GEU
```

157.
```
gap <- ((PAUSE / GU) (freemod)?)
```

158.
```
juelink <- (JUE (freemod)? argument (freemod)?)
```

159.
```
links1 <- (juelink ((links1 gue))? (freemod)?)
```

160.
```
links <- (((links1 / (&((((KA1 (freemod)?))+ links1)) (KA1 (freemod)? links KI (free
```

161.
```
jelink <- ((JE (freemod)? argument) (freemod)?)
```

162.
```
linkargs1 <- (jelink (links)? (gue)? (freemod)?)
```

163.
```
linkargs <- (((linkargs1 / (&((((KA1 (freemod)?))+ linkargs1)) KA1 (freemod)? linkar
```

164.
```
predunit1 <- ((SUE / (NU (freemod)? GE (freemod)? despredE (geu)?) / (NU (freemod)?
```

165.
```
predunit2 <- ((((NO1 (freemod)?))* predunit1) (freemod)?)
```

166.
```
predunit3 <- (((predunit2 linkargs) / predunit2) (freemod)?)
```

167.
```
predunit <- ((((PO (freemod)?))? predunit3) (freemod)?)
```

168.
```
kekpredunit <- (&((((KA1 (freemod)?))+ (PA / NO1 / PO / BI / CUI / predunit1))) ((NO
```

169.
```
despredA <- (((predunit ((CI (freemod)? despredA))?) / kekpredunit) (freemod)?)
```

170.
```
despredB <- (((!(PREDA) CUI (freemod)? despredC CA (freemod)? despredB) / despredA)
```

171.
```
despredC <- ((despredB)+ (freemod)?)
```

172.
```
despredD <- ((despredB ((CA (freemod)? despredB))*) (freemod)?)
```

```
despredE <- (despredD)+
```

```
descpred <- (((despredE GO (freemod)? descpred) / despredE) (freemod)?)
```

```
senpred1 <- (((predunit CI (freemod)? senpred1) / predunit) (freemod)?)
```

```
senpred2 <- ((senpred1 / (CUI (freemod)? despredC CA (freemod)? despredB)) (freemod)
```

```
senpred3 <- ((senpred2 ((CA (freemod)? despredB))*) (freemod)?)
```

```
senpred4 <- ((senpred3 (despredD)*) (freemod)?)
```

```
sentpred <- (((senpred4 GO (freemod)? barepred) / senpred4) (freemod)?)
```

```
mod1 <- (((PA2 (freemod)? argument (freemod)? (gap)?) / (PA2 (freemod)? !(barepred)
```

```
mod <- ((mod1 / (NO1 (freemod)? mod1)) (freemod)?)
```

```
kekmod <- ((((NO1 (freemod)?))* (KA3 (freemod)? modifier KI (freemod)? mod)) (freemo
```

```
modifier <- (((mod / kekmod) ((A2 (freemod)? mod))*) (freemod)?)
```

```
namemarker <- (([ ])* ('la' / 'La' / 'hoi' / 'Hoi' / 'ci' / 'Ci' / 'hue' / 'Hue'))
```

185.
```
#Added CI label to predunits in names ; solves the serial name versus sentence probl

name <- ((DJAN (((CI (freemod)? DJAN) / (CI (freemod)? predunit) / (!(namemarker) DJ
```

186.
```
descriptn <- (((LE (freemod)? descpred) / (LE (freemod)? mex (freemod)? descpred) /
```

187.
```
#Eliminated unmarked vocatives; to restore, eliminate HOI before name

voc <- (((HOI name gap) / (HOI (freemod)? name (gap)?) / (HOI (freemod)? descpred (g
```

188.
```
arg1 <- (((LEPO (freemod)? uttAx (guo)?) / (LEPO (freemod)? sentence (guo)?) / (LIO
```

189.
```
arg1a <- ((TAI / DA / arg1 / (GE2 (freemod)? arg1a)) (freemod)?)
```

190.
```
argmod1 <- (((JI (freemod)? predicate (gui)?) / (JIO (freemod)? sentence (gui)?) / (
```

191.
```
argmod <- ((argmod1 ((A3 (freemod)? argmod1 (gap)?))*) (freemod)?)
```

192.
```
arg2 <- ((arg1a ((argmod (gap)?))*) (freemod)?)
```

193.
```
arg3 <- ((arg2 / (mex (freemod)? arg2)) (freemod)?)
```

194.
```
indef1 <- ((mex (freemod)? descpred) (freemod)?)
```

195.
```
indef2 <- ((indef1 (gap)? ((argmod (gap)?))*) (freemod)?)
```

196.
```
indefinite <- (indef2 (freemod)?)
```

197.
```
arg4 <- (((indefinite / arg3) ((ZE2 (freemod)? (indefinite / arg3)))*) (freemod)?)
```

198.
```
arg5 <- ((arg4 / (KA3 (freemod)? argument KI (freemod)? argx)) (freemod)?)
```

199.
```
arg6 <- ((arg5 / (DIO (freemod)? arg6) / (IE (freemod)? arg6)) (freemod)?)
```

200.
```
argx <- ((((NO1 (freemod)?))* arg6) (freemod)?)
```

201.
```
arg7 <- ((argx ((ACI (freemod)? arg7))?) (freemod)?)
```

202.
```
arg8 <- ((arg7 ((A4 (freemod)? arg7))*) (freemod)?)
```

203.
```
argument <- ((((LAU wordset) / (arg8 AGE (freemod)? arg8) / arg8) ((GUU (freemod)? a
```

204.
```
term <- ((argument / modifier) (freemod)?)
```

205.
```
terms <- ((term)+ (freemod)?)
```

206.
```
word <- ((arg1a (gap)?) / (UI (gap)?) / (NI (gap)?) / (PA2 (gap)?) / (DIO (gap)?) /
```

207.
```
words <- (word)+
```

208.
```
wordset <- (((words)? lua) (freemod)?)
```

209.
```
termset1 <- (terms (guu)?)
```

210.
```
termset2 <- (((termset1 / (KA3 (freemod)? termset2 KI (freemod)? termset1)) ((A4 (fr
```

211.
```
termset <- (((terms GO (freemod)? barepred) / termset2 / guu) (freemod)?)
```

212.
```
kekpred <- ((kekpredunit (despredD)*) (freemod)?)
```

213.
```
barepred <- (((sentpred (termset)?) / (kekpred (termset)?)) (freemod)?)
```

214.
```
markpred <- ((((NO1 (freemod)?))* ((PA1 (freemod)? barepred) / (PO (freemod)? (gap)?
```

215.
```
backpred1 <- (((NO1 (freemod)?))* (barepred / markpred / kekpred) (freemod)?)
```

216.
```
backpred <- ((backpred1 ((ACI (freemod)? backpred))*) (freemod)?)
```

217.
```
barefront <- (barepred ((A1 (freemod)? backpred (termset)?))*)
```

218.
```
bareekpred <- (barefront A1 (freemod)? backpred)
```

219.
```
markfront <- ((markpred ((A1 (freemod)? backpred (termset)?))*) (freemod)?)
```

220.
```
markekpred <- ((markfront A1 (freemod)? backpred) (freemod)?)
```

221.
```
predicate2 <- (((NO1 (freemod)?))* (barefront / markfront) (freemod)?)
```

222.
```
predicate1 <- (((predicate2 AGE (freemod)? predicate1) / predicate2) (freemod)?)
```

223.
```
identpred <- (((NO1 (freemod)?))* (BI (freemod)? termset) (freemod)?)
```

224.
```
predicate <- ((predicate1 / identpred) (freemod)?)
```

225.
```
gasent <- (((NO1 (freemod)?))* (PA1 (freemod)? barepred GA2 (freemod)? terms) (freem
```

226.
```
statement <- ((gasent / (terms gasent) / (terms predicate)) (freemod)?)
```

227.
```
keksent <- (((NO1 (freemod)?))* ((KA3 (freemod)? sentence KI (freemod)? uttA1) / (KA
```

228.
```
sen1 <- ((statement / predicate / keksent) (freemod)?)
```

229.
```
sentence <- ((sen1 ((ICA sen1))*) (freemod)?)
```

230.
```
headterms <- ((((terms GI))+ (freemod)?) (freemod)?)
```

231.
```
uttA <- ((A4 / IE / mex) (freemod)?)
```

232.
```
uttAx <- ((headterms sentence (gap)?) (freemod)?)
```

233.
```
period <- [!.:;?]
```

234.
```
uttA1 <- ((sen1 / uttAx / NO1 / links / linkargs / argmod / (terms keksent) / terms
```

235.
```
freemod <- ((UI / (SOI (freemod)? descpred (gap)?) / DIE / (NO1 DIE) / (KIE utteranc
```

236.
```
neghead <- ((NO1 (freemod)? (gap)?) (freemod)?)
```

237.
```
uttC <- (((neghead uttC) / uttA1) (freemod)?)
```

238.
```
uttD <- ((uttC ((ICI (freemod)? uttD))*) (freemod)?)
```

239.
```
uttE <- ((uttD ((ICA (freemod)? uttD))*) (freemod)?)
```

240.
```
uttF <- ((uttE ((I (freemod)? uttF))*) (freemod)?)
```

241.
```
utterance0 <- ((uttE IGE utterance0) / (I (freemod)?) / freemod / uttF / (I (freemoc
```

242.
```
utterance <- ((freemod utterance) / (freemod !(.)) / (uttE IGE utterance) / (I (free
```