Annotated Formal Grammar of Loglan, Provisional

Randall Holmes

October 28, 2013

October 28 2013: very provisional draft, partway through the "lexer" section.

October 5 2013: basically complete account of phonology.

October 12, 2013: starting annotating the lexer section.

This document is a commented version of the PEG generated by the savegrammar() command with the 9/1/2013 version of my grammar files [initially: changes have been made to the grammar as I worked]. This is not an official grammar for TLI Loglan, though it is intended to serve in this role eventually when fully refined. It is based on the official grammar with some modifications which are intended for submission to the Academy; it still likely has bugs in it.

When I make changes in grammar rules, I will try to track them in this file as well.

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 The Annotated PEG file

Notes from the PEG file I am using.

```
#The grammar for TLI Loglan in PEG format by Randall Holmes name boundary
# A CA PA fixes
# ZAO enabled
# final borrowings y hyphenated
# commas between numbers
# really fixed bug in complexes
```

```
# fixed bug in complexes; unofficial solution for acronyms.
# modifiers in je/juelinks
#quotation tweaks
# PAUSE/GU fixes
# subtle correction of definition of LIU
# restored LI...LU to quoting utterances
#LIU is a name marker
#minor changes during reformatting of notes
#name refinements; fixed priority bug in uttA1
# fixes bug in borrowing priority
# fixes to DA1 and arg1
# name-marker-initial names shielded properly by CI
# separated LA from LE.  Made freemods highest priority at beginning of ut
#Further refinements of name boundary detection.  Addition of TAI words to
# I am making this the primary version.  Do note though that one must assu
#Rule arg1 required notable modifications.
#the rules __LWinit and __LWbreak are different, not checking for names, a
#Spaces after predicate words must appear and give phonetic information ab
#In some sense to be made exact, I regard this parser as phonetic.
#This is not yet the official grammar.  That remains trial.85
# plus mysterious authority for lexing and phonology from various sources.
#
#Though private noncommercial use is fine, this remains intellectual
#property of Randall Holmes and the Loglan Institute.  Use in derived work
# should involve consultation.  Members of the Logical Language Group
# (Lojbanists) are encouraged to play
```

1. The class of expressions of PEG grammars. I'll follow the general theme of the file and describe the PEG language in the form of notes on its rules. A PEG grammar is a list of definitions.

   ```
   _Grammar <- (_Spacing (_Definition)+ _EndOfFile)
   ```

2. Each definition takes the form of the name of a grammar class followed by <- followed in turn by an expression in the PEG language describing the grammar class.

```
_Definition <- (_Identifier _LEFTARROW _Expression)
```

3. At the highest level, an expression of the PEG language is a sequence of "sequence expressions" separated by `/`; the expression parses the class of all things which can be parsed using one of the sequence expressions, using the first sequence expression in the list that matches.

```
_Expression <- (_Sequence ((_SLASH _Sequence))*)
```

4. A "sequence expression" is a list of "prefix expressions" written one after the other with nothing but whitespace between them. A "sequence expression" describes an expression which is (as the name indicates) a sequence of expressions, the first one parsed by the first prefix expression in the sequence, the second by the second prefix expression, and so forth.

```
_Sequence <- (_Prefix)*
```

5. A prefix expression is a suffix expression, or an suffix expression preceded by `&` or a suffix expression preceded by `!`. A bare suffix expression parses as described below. A suffix expression preceded by `&` parses an expression of length 0 (an empty string) which is followed by an expression parsed by the suffix expression; a suffix expression preceded by `!` parses an empty string which is followed by an expression not parsed by the following suffix expression. The last two forms do lookahead tests: one makes no forward progress in parsing the expression, but looks ahead to see if there is a following string which the suffix expression would accept or no such string.

```
_Prefix <- (((_AND / _NOT))? _Suffix)
```

6. A suffix expression is a primary expression, or a primary expression followed by one of `?`, `*`, or `+`. A primary expression parses as indicated below. A primary expression followed by `?` parses a string parsed by the primary expression or, if there is no such string, an empty string (this will not fail). A primary expression followed by `*` parses zero or

more successive strings parsed by the primary expression, continuing until there aren't any more; again, this won't fail. A primary expression followed by + parses one or more successive strings parsed by the primary expression, continuing until there aren't any more; it fails if it doesn't find one such expression.

```
_Suffix <- (_Primary ((_QUESTION / _STAR / _PLUS))?)
```

7. A primary expression is a grammar class name (interpreted using its definition in the current grammar), or a general expression in parentheses, or a literal string (described below), or a character class (described below), or a dot (described below).

```
_Primary <- ((_Identifier !(_LEFTARROW))
/ (_OPEN _Expression _CLOSE) / _Literal / _Class / _DOT)
```

8. Without annotating each rule, we simply say that an identifier is a string made of up letters (upper case or lower case), digits, and instances of _ which does not begin with a digit. Identifiers are used as names of grammar classes introduced by definition. Definitions can be recursive.

```
_Identifier <- (_IdentStart (_IdentCont)* _Spacing)
```

9. `_IdentStart <- [a-zA-Z_]`

10. `_IdentCont <- (_IdentStart / [0-9])`

11. A literal is a string enclosed in either single or double quotes. It accepts exactly the string that it represents, which is literally the string between the quotes, with exceptions for certain quite standard escape sequences, and fails if it does not see this string.

```
_Literal <- (([\'] ((![\']) _Char))* [\'] _Spacing)
/ ([\"] ((![\"]) _Char))* [\"] _Spacing))
```

12. A character class consists of a list of characters or ranges of characters (or escape sequences representing characters) enclosed by brackets. It accepts a single character, which must be one of the listed characters or belong to one of the listed ranges, and otherwise fails.

```
_Class <- ([\[] ((!([\]]) _Range))* [\]] _Spacing)
```

13. A range is two characters or escape sequences separated by a hyphen. A result of this is that a hyphen must appear at the beginning or end of the list in a character class to be understood as a character.

```
_Range <- ((_Char [-] _Char) / _Char)
```

14. A character is an escape sequence or a single character. pegparser.sml does not support the ASCII numerical escape sequences yet.

```
_Char <- (([\\] [nrt\'\"\[\]\\])
/ ([\\] [0-2] [0-7] [0-7])
/ ([\\] [0-7] ([0-7])?) / (!([\\]) .))
```

15. Some rules which need no particular comment (characters used in the PEG language)

```
_LEFTARROW <- ('<-' _Spacing)
```

16. `_SLASH <- ('/' _Spacing)`
17. `_AND <- ('&' _Spacing)`
18. `_NOT <- ('!' _Spacing)`
19. `_QUESTION <- ('?' _Spacing)`
20. `_STAR <- ('*' _Spacing)`
21. `_PLUS <- ('+' _Spacing)`

22. `_OPEN <- ('(' _Spacing)`

23. `_CLOSE <- (')' _Spacing)`

24. The dot class accepts a single character. It fails at end of file.

    ```
    _DOT <- ('.' _Spacing)
    ```

25. This is the class of whitespace and comments to be ignored when parsing the PEG language.

    ```
    _Spacing <- ((_Space / _Comment))*
    ```

26. A comment begins with `#` and continues to the end of a line.

    ```
    _Comment <- ([#]
    ((!(_EndOfLine) .))* _EndOfLine)
    ```

27. A space is an actual space, a tab, or the end of a line.

    ```
    _Space <- ([ ] / [\t] / _EndOfLine)
    ```

28. An end of line is one of the various forms of carriage return.

    ```
    _EndOfLine <- (([\r] [\n]) / [\r] / [\n])
    ```

29. The end of file is where there are no more characters. This completes the discussion of the PEG language (the language in which the rules themselves are written).

    ```
    _EndOfFile <- !(.)
    ```

30. Now we begin the grammar of Loglan. Since this parser works from letters up, it treats phonology as part of the formal grammar. The first block of rules are about Loglan phonology.

    The first rule describes the class of lowercase letters. In fact `w` is no longer a Loglan letter in effect even in this grammar, and there is a move afoot to banish `q` and `x` as well.

```
##Loglan phonology
lowercase <- [a-z]
```

31. Uppercase letters. There is no phonetic difference, but as we will see there are capitalization rules.

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

32. A letter in general.

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

33. The general vowel class. I believe that this class is no longer in use, except in the definition of the consonant class.

```
V <- [AEIOUWYaeiouwy]
```

34. The regular vowels of Loglan. I pronounce these as follows

    a as in father, never as in mate

    e as in bed, never as in mete

    i as in machine, never as in vine, and avoiding a following y sound

    o as in loss or perhaps bore never as in bone [the point here being that the vowel in bone, at least in my idiolect, is unmistakably a diphthong **ou**, in Loglan terms]

    u as in tune, but avoiding closing it up with a following w sound [poor might be a better sample word]

    John Cowan tells me this is the vowel system of Polish. Other similar pronunciations of the vowels from continental Europe are appropriate; the standard English pronunciations are as a rule not.

    The irregular vowel y can have the schwa sound, or the sound of oo in look (a suggestion of John's which I like)

    Historically the irregular vowel w had the u-umlaut sound; it is already eliminated in this provisional grammar.

The references to English pronunciation above can be misleading of course, as they refer to how I pronounce English words, and the English vowels are notably mutable between English speakers, as well as being written very strangely by the standards of other European languages.

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

35. The class of Loglan consonants. Their pronunciations are mostly quite standard.

    g is never soft as in gem, always hard as in gun.

    c has the sound of sh as in shun (tc is the sound of ch in chin)

    j has the sound of z in azure (dj is the sound of j in jam)

    x is the sound of ch in Scottish loch - a move is afoot to remove this letter and make this sound an acceptable pronunciation for h in addition to the usual one

    q is pronounced as th in thin (quite strangely!); there is a move afoot to eliminate this letter entirely.

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

36. These are the vowel pairs which can be pronounced monosyllabically in Loglan.

    ao is pronounced as ow is in English cow and must be pronounced monosyllabically

    ai ei oi must be pronounced monosyllabically and are the vowels in E bye, bay, boy respectively.

    The others are optional monosyllables; one may pronounce initial i/u in a vowel pair as a separate syllable or as consonantal y/w (using the E pronunciations of these letters). There are contexts in which the monosyllabic pronunciation of a vowel pair is forced, and this parser *always* assumes monosyllabic pronunciation when it parses syllables. I believe that there is no situation where the disyllabic pronunciation of any of these vowel pairs is forced.

As the grammar always assumes the monosyllabic pronunciation, it draws no distinction between the mandatory and optional disyllables. An eventual phonetic parser will need to be able to draw this distinction.

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

37. The class of CVV syllables.

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

38. According to NB3, pp. 46-7, a compound little word (structure word) is either a sequence of VV's or a sequence of blocks of the following kinds plus possibly an initial V.

    Notice that a (C V2 V2) block is not necessarily a syllable!

    It appears that the (CVVSyll V2) blocks appear only in acronyms.

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

39. Another useful class of structure word units.

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

40. This describes the simplest capitalization rule: a string accepted by caprule is a string of letters in which the first may be upper or lower case, the others are all lower case, and the string is not followed by another letter.

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

41. The pairs of consonants permitted at the beginnings of syllables in Loglan. SV was already noticed and made legal in Appendix H. ZL has been made legal because there is a composite primitive beginning with this consonant pair. Notice that capitalized variants are included.

```
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')
```

42. The class of pairs of consonants which are not permitted to occur adjacent to one another at a syllable break.

```
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')
```

43. A class of triples of consonants which are not permitted to occur at syllable breaks. The assumption here is that one is looking at the juncture of a CVC combining form and a CCV combining form so the break here is expected to be C/CC

    More general situations can occur in borrowings, and I think some other consonant combinations which are currently allowed are quite dubious. This does make it unlikely that borrowing designers will use them, so it may not be necessary to expand this list.

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

44. This is the class of pairs of repeated vowels to which the stress rule applies. In any context in Loglan, if these pairs of vowels occur they are in different syllables and one of them must be stressed.

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

45. These are the vocalic consonants permitted in Loglan. I have based my rules for use of vocalic consonants on the use of vocalic consonants for gluing in borrowings. In those uses, a vocalic consonant is always followed by a non-vocalic occurrence of the same consonant, and I have extended this rule to names (so **Rl** is **Rrl** for me and **Bastn** becomes **Bastnn**). I could fix this, but the easiest thing to do when I made names pronounceable was to adapt the rules already set up for borrowings. There is positive merit to this approach: it makes the appearance of a vocalic consonant very obvious.

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

46. The initial block of consonants in a syllable. This can be one, two or three consonants. Each pair of adjacent consonants should be a permitted initial and the last of the consonants should not be the first of a vocalic consonant pair.

    Note to myself: I'm not certain why only the single consonant is blocked from being followed by y here. Answer: it is to prevent the C in a CVC combining form from grabbing a following y hyphen and being incorrectly regarded as starting a syllable. The other forms are prevented from being followed by y in any case as a FirstConsonants block is to be followed by a vowel segment.

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

47. The initial block of consonants in a syllable. This can be one, two or three consonants. Each pair of adjacent consonants should be a permitted initial and the last of the consonants should not be the first of a vocalic consonant pair. This is the version for use in names (where the vowel segment in a syllable can be y).

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

48. The vowel segment in a syllable. This will either be a monosyllabic vowel pair, or a vowel, or the first consonant in a vocalic consonant pair.

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

49. A syllable is an optional first consonant block, followed by a mandatory vowel segment, followed by zero one or two final consonants. The FinalConsonant class is defined just below, because it has a recursive dependence on Syllable.

    It is important to note that there is no official account of syllabification in Loglan. There is quite a lot of implicit justification for elements of the treatment I use here in the sources. The actual restrictions that I impose that are not in the sources are restrictions on the length of blocks of consonants in borrowings or names (to no more than 5) which seem entirely reasonable, restriction on the number of initial vowels in a borrowing or the number of vowels following the initial C in a borrowing (to no more than 3), and the restriction on name words that they be syllabized in roughly the same way as borrowings except for the permission to use y as a regular vowel. Also, consonants used vocalically always appear doubled, which will affect the way certain names are written. This entire phonetic package will need to be a proposal to the Academy. The corpus of words actually used is compliant, except for the doubling of vocalic consonants in names, which is straightforward to correct.

    Part of the reason that there is no official account of syllabification is that in most contexts the natural units are not syllables: in both structure words and complexes, CVV forms which are possibly disyllabic are natural structural units.

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

50. The requirements for a final consonant are that it not begin a pair or triple of consonants disallowed at a syllable break and also that it not form a syllable with following letters: we prefer to move the syllable break in a block of consonants as far to the left as possible. These rules make the length of the longest possible block of consonants at a syllable break 5 (two final consonants followed by a three consonant first consonant block). Of course this would only happen in a borrowing.

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

51. This is the form of syllable used in names, where the vowel segment can be y.

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

52. This is the form of final consonant used in names.

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

53. This is the definition of the class of name words. A name word is capitalized (the capital letter may be preceded by whitespace). It is made of one or more syllables of class Syllable2. The last letter in the name word is a consonant, followed by a comma [regarded as included in the name word] unless the name word is followed by a period-class punctuation mark or end of file. Phonetically, the comma represents a mandatory pause.

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

54. We are continuing with the phonology of predicate words.

    This class is used to recognize the CV syllables at the ends of primitive predicate words. This is a CV not followed by another regular vowel.

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

55. This is the final consonant of a CVC combining form. It is either a consonant followed by a hyphen (which must be followed by another letter) or a consonant not followed by a vowel and not beginning a bad consonant group.

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

56. This is the general form of a hyphen between combining forms. It will not stand at the beginning of a forbidden consonant group. It may be an r, not followed by another r or a regular vowel, or it may be an n followed by an r, or it may be a y. The hyphen is always followed by a letter (I do not implement the odd permission to break complexes at y's given in chapter 6) and is never followed by a y.

```
hyphen <- (!(NonmedialCC) !(NonjointCCC)
(([r] !([r]) !(V2)) / ([n] &([r])) / [y]) &(letter) !([y]))
```

57. This is a CVV combining form which must carry primary stress due to the repeated vowel stress rule.

    NOTE: error corrected 10/5

```
CVVStressed <- (C &(RepeatedVowel) V2 !(RepeatedVowel) V2 (hyphen)?)
```

58. This is a CVV combining form which must be disyllabic.

```
CVVDisyllable <- (C !(Mono) V2 V2 (hyphen)?)
```

59. This is the general CVV combining form.

```
#The restriction on repeated vowels prevents a CVV affix from
forcing stress on the first syllable of a following borrowing; in
principle this could be allowed if the borrowing had two syllables.

CVV <- (C V2 !(RepeatedVowel) V2 (hyphen)?)
```

60. This is the CVC combining form class.

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

61. The CCV class. The comment in the source says it all about the RepeatedVowel issue.

```
#The repeated vowel rule and option of a y-hyphen here
are strictly to deal with following vowel-initial
borrowings in complexes

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

62. This is the first of the two classes of primitive predicates considered as combining forms. It appears in its full form only if final in a complex, and otherwise with its final vowel replaced by y (and in this case not final in the complex).

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

63. This is the second of the two classes of primitive predicates considered as combining forms. It appears in its full form only if final in a complex, and otherwise with its final vowel replaced by y (and in this case not final in the complex).

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

A syllable beginning with at least two consonants, and not having a vocalic consonant. This is part of a list of syllable classes providing CC joints for predicate words.

NOTE: error corrected 10/5

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

65. A syllable which has a CC joint at the end (either with the initial consonant of a following syllable (possibly with an intervening y hyphen), or because it has two final consonants itself.)

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

66. A syllable with a vocalic consonant, which automatically produces a CC joint.

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

67. This is a syllable which causes a stress with a following syllable due to the repeated vowel stress rule.

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

68. A block of letters which looks like a consonant initial structure word unit.

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

69. A sequence from which we can peel off an initial structure word. We do not allow these sequences to occur at the beginnings of predicate words. This represents a change from NB3, though it does not change the dictionary: JCB explicitly says that a borrowing can begin with an arbitrarily long string of vowels. We allow no more than three; the dictionary is compliant. (words with two initial vowels occur).

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

70. This is the class of syllables not forcing stresses with following syllables via the repeated stress rule. We have now officially banned repeated vowels in borrowings; this class is used in the implementation.

```
#Provisionally forbidding repeated vowels in
 borrowings altogether

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

71. The definition of the class of borrowings, possibly the most delightfully baroque feature of Loglan.

    A borrowing may not be of the form CCVV (part of a recent Keugru ruling).

    A borrowing may not be consonant-final (making it not a name works).

    A borrowing may not begin with a BadSequence (so a structure word cannot be peeled off the front).

    !((V2 InitialCC V2)) prevents a vowel from falling off the front in some odd circumstances. It forbids the word **iglu** which was still permitted in L1; there is a discussion in Appendix H.

    !(((CW1 / (V2 V2) / V2) (!(V2) Borrowing))) prevents a structure word from falling off the front of a consonant-initial borrowing.

    !(CCSyllableB) prevents the first syllable from having a vocalic consonant; this saves the word **hidrroterapi** in the dictionary from having its first syllable fall off.

    &((NoBadstress (NoBadstress)+)) prevents repeated vowels.

    ((!((CCSyllable / CCSyllableB / CCFinalSyllable)) Syllable))* (Syllable)+) forces it to consist of at least two syllables, one of which must in one way or another induce a CC joint.

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

72. This is the class of combining forms (colloquially "affixes") used to build complex predicates in Loglan.

Per the ruling in Appendix H, the affix for a borrowing is the entire borrowing, plus a final y when in non-final position. A borrowing appearing in final position must be preceded by a y (enforced in the next rule).

CVV combining forms are burdened with rules preventing a CVVStressed form from being followed by two syllables.

CVC combining forms cannot be final in the ambient word.

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

73. This rule describes a first approximation to the class of complex predicates. A complex contains at least two combining forms and a final borrowing must be preceded by a y; the logic of the expression is a bit tricky.

```
#It appears that a final borrowing in a complex must be y-hyphenated

PreComplex <- ((Affix (PreComplex / (Affix !(letter))))
/ ((CCVCV / CVCCV / (Borrowing [y]) / (Affix [y])) Borrowing !(letter)))
```

74. This is the exact definition of a complex, imposing additional conditions on the class of precomplexes just defined.

    A complex is not a name (this is probably redundant).

    A complex does not consist of a CV or CVV followed by a precomplex, a final CVV affix, or a borrowing (this would be a little word that could fall off the front, or there might be failure to form a CC joint).

    A complex does not begin with a CVC affix making an initial pair with the next consonant, unless it is a six-letter form (the modification in Appendix H which vacated the slinkui test, plus the recent revision of that modification by Keugru proposal).

    A complex is a precomplex, and it is a complete word (not followed by another letter).

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

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

75. The class of primitive predicates.

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

76. The class of predicates. Predicate words are either primitives, complexes, or borrowings.

    Ths proposal of John Cowan to allow complex predicates to be formed using the structure word ZAO is implemented here.

```
Predicate <- (&(caprule) (Primitive / Complex / Borrowing)
 !(letter) ((([ ])+ ('zao' / 'Zao') ([ ])+ Predicate))?)
```

77. The class __LWinit is used to start reading a structure word; it accepts spaces, followed by a string of letters satisfying the capitalization convention which is not a predicate.

    NOTE: throughout the lexer section, I should provide forward references to the rules using given particles.

```
##The lexer, so to speak

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

78. This grammar rule is used to distinguish between CV structure word units and CVV structure word units. The point is that a string of vowels construed as word units is always of the form V?(VV)*, either

a single V (which must be preceded by a pause) followed by a string of VV's, or a string of VV's. A CV or CVV structure word unit can only be followed by VV*, which is why we can use !Oddvowel to assure that we have ended such a word in the right place.

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

Oddvowel <- (((V2 V2))* V2)
```

79. A logical connective a e o u or ha, just the bare vowel.

    NOTE: I dont see why I allowed __LWinit before ha.

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

80. The form of an A-class word. The class A0 core may start with NU if what follows is NO or U; there follows an optional NO; the mandatory A0 core follows, followed by a possible NOI then a possible tense word (class PA)).

    A space (surface level word break) might appear before the PA; nowhere else.

    A pause expressed by a comma must appear before any word of this class.

    The PA component of this class is recursively defined, making this class infinitely large.

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

81. This grammar class identifies a possible break in a structure word, which must be comma-marked. Such breaks are necessary in for example the infamous le, po situation. Such a break does not occur before a predicate and does not occur before an A word.

    This form can be used as a component of other words.

    NOTE: there is a failure of parallelism with KA words. APA words can have quite complex PA words, KOU initial and compounds included and parallel KA words do not exist for LIP. Of course perhaps APA words should be banished; I have demonstrated that they can cause a need for structure word breaks.

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

82. The class of A words with optional following structure word break.

    NOTE: There is no reason in our grammar for all the numbered versions.

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

83. `A2 <- (A __LWbreak)`

84. `A4 <- (A __LWbreak)`

85. `A3 <- (A __LWbreak)`

86. ACI words. Logical connectives with special grouping properties; see below how they are used.

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

87. AGE words. Logical connective with special grouping properties; see below how they are used.

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

88. Words like BI (the identity predicate). A special class of predicate words.

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

89. The CA series of logical connectives (just the core).

    NOTE: do I want to allow spaces between a CA and a preceding NU or NO? (the LWinit allows this). I don't allow it for A class words, but there there is a strong phonetic reason not to (the mandatory pause before an A word goes in front of the NU and/or NO)/

```
CA0 <- ((__LWinit 'ca') / 'ce' / 'co'
/ 'cu' / 'ze' / 'Ca' / 'Ce' / 'Co' / 'Cu'
/ ('Ze' !(Oddvowel)))
```

90. The fully decorated class of CA words, structure analogous to A words.

    The PA component of this class is recursively defined, making it infinitely large.

    This form can be used as a component of other words.

```
CA1 <- (__LWinit (('nu' / 'Nu')
&(('cu' / 'no'))))? (('no' / 'No'))?
CA0 ('noi')? (PA)? !(Oddvowel))
```

91. CA words further equipped with structure word break permission.

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

92. The word CI (a hyphen).

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

93. The word CUI (used in predicate grouping).

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

94. Bare units of numerical or quantifier words.

```
NI0 <- (('kua' / 'gie' / 'giu' / 'hie' / 'hiu'
/ 'kue' / 'Kua' / 'Gie' / 'Giu' / 'Hie' / 'Hiu'
/ 'Kue' / 'nea' / 'nio' / 'pea' / 'pio' / 'suu'
 / 'sua' / 'tia' / 'zoa' / 'zoi' / 'Nea' / 'Nio'
/ 'Pea' / 'Pio' / 'Suu' / 'Sua' / 'Tia' / 'Zoa'
/ 'Zoi' / 'ho' / 'ni' / 'ne' / 'to' / 'te' / 'fo'
/ 'fe' / 'vo' / 've' / 'pi' / 'Re' / 'Ru' / 'Sa'
/ 'Se' / 'So' / 'Su' / 'Ho' / 'Ni' / 'Ne' / 'To'
/ 'Te' / 'Fo' / 'Fe' / 'Vo' / 'Ve' / 'Pi' / 're'
/ 'ru' / 'sa' / 'se' / 'so' / 'su' / 'Hi' / 'hi') !(Oddvowel))
```

95. Bare unit letteral words. Vowel letters like ama, afi, consonant letters like mai, mei, meo. The terminating logical condition is tricky: it is either not followed by an odd length string of vowels, or it is followed by one or more Vz units followed by a single vowel and no further letters or another TAI0. This all has to do with the scheme for allowing vowels to have one letter names in acronyms separated by z's.

```
TAI0 <- (((V !(Predicate) !(Name) 'ma')
/ (V !(Predicate) !(Name) 'fi') / (C 'ai') / (C 'ei')
/ (C 'eo')) (!(Oddvowel) /
&((((V2 [z]))* ((V2 !(letter)) / &(TAI0))))))
```

96. addition of mo=00 and ma = 000 to numerals.

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

97. quantifier units

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

98. The large class of mathematical words, numerals and quantifiers. Some detail here is caused by the fact that such words cannot be RA final (such words are numerical predicates).

    Dimensioned numbers are supported, but the acronym used as a dimension must be front-marked with MUE, a new proposal; I think that acronyms just have to be front marked.

    Numeral words definitely need to be separated from following numeral words by explicit commas in certain situations, and this is enforced here.

    NOTE: this is a wildly heterogeneous class. The forms with CU are something quite different. There are grammatical contexts where one really wants a complex numeral and nothing else. There might need to be some subclasses of this defined.

```
#I am supporting dimensioned numbers here
 but note that acronyms used in this way are
front marked with MUE

NI <- (__LWinit ('ie')?
 ((((RA / (NI1 &(NI1))))* NI1) / RA)
!((((  ])+ !(Predicate) (NI1 / RA)))
 (([,] &((((  ])+ !(Predicate) (NI1 / RA)))))?
(Acronym2)? ('cu')? !(Oddvowel))
```

99. Bare pronoun units.

```
DA0 <- (__LWinit ('tao' / 'tio' / 'tua'
/ 'Tao' / 'Tio' / 'Tua' / 'mio' / 'miu' / 'muo'
/ 'muu' / 'Mio' / 'Miu' / 'Muo' / 'Muu' / 'toa'
/ 'toi' / 'too' / 'tou' / 'tuo' / 'tuu' / 'suo' / 'Toa'
/ 'Toi' / 'Too' / 'Tou' / 'Tuo' / 'Tuu' / 'Suo'
/ 'Hu' / 'ba' / 'be' / 'bo' / 'bu' / 'da' / 'de'
```

```
/ 'di' / 'do' / 'du' / 'Ba' / 'Be' / 'Bo' / 'Bu'
/ 'Da' / 'De' / 'Di' / 'Do' / 'Du' / 'mi' / 'tu'
/ 'Mi' / 'Tu' / 'Mu' / 'Ti' / 'Ta' / 'mu' / 'ti' / 'ta' / 'hu')
!(Oddvowel))
```

100. Two classes of acronyms. The first is used for acronymic predicates; note that these are front marked with MIE. This is a new proposal. Apart from the initial MIE marking, this should more or less follow the quite complicated recipe for acronyms in NB3. I owe the reader more explanation.

     NOTE: more explanation of details here.

```
#Acronyms are not allowed to
include the fancy new GAO letterals
 for the moment; consistent with NB3 anyway

Acronym <- (&(caprule) ('mie' / 'Mie')
 (!(Oddvowel) / &(TAI0)) !(NI1)
 ((NI1 / TAI0 / (([z])? V2 (!(V2) / ([z] &(V2))))))+
 !((([ ])* !(Predicate) (NI1 / TAI0
/ (([z])? V2 (!(V2) / ([z] &(V2)))))))
 (([,] &((([ ])+ !(Predicate) (NI1 / TAI0
/ ([z] V2 (!(V2) / ([z] &(V2)))))))))?)
```

101. The second class of acronyms, used in dimensioned numbers, front marked with MUE.

```
Acronym2 <- (&(caprule) ('mue' / 'Mue')
 (!(Oddvowel) / &(TAI0)) !(NI1)
 ((NI1 / TAI0 / (([z])? V2 (!(V2) / ([z] &(V2))))))+
 !((([ ])* !(Predicate) (NI1 / TAI0
/ (([z])? V2 (!(V2) / ([z] &(V2)))))))
(([,] &((([ ])+ !(Predicate) (NI1 / TAI0
/ ([z] V2 (!(V2) / ([z] &(V2)))))))))?)
```

102. The full class TAI of letteral words. Includes John Cowan's extended GAO proposal.

     NOTE: I note that I think we still need monosyllabic words for letters that we drop from the Loglan alphabet.

```
#Enabled John Cowan's proposal for a large letteral space

TAI <- (__LWinit (TAI0 / (('gao' / 'Gao')
 !(V2) ([ ])* (Name / Predicate / (C V2 V2 (!(Oddvowel)
 / &(TAI0))) / (C V2 (!(Oddvowel) / &(TAI0)))))) __LWbreak)
```

103. The class of pronoun words (as components of other words).

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

104. Pronouns further equipped with the option of following structure word break.

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

105. DIE words (attitudinals)

```
DIE <- (__LWinit ('die' / 'fie' / 'kae' / 'nue' / 'rie' / 'Die'
/ 'Fie' / 'Kae' / 'Nue' / 'Rie') __LWbreak)
```

106. DIO words (case tags)

```
DIO <- (__LWinit ('beu' / 'cau' / 'dio' / 'foa' / 'kao'
/ 'jui' / 'neu' / 'pou' / 'goa' / 'sau' / 'veu' / 'zua'
/ 'zue' / 'zui' / 'zuo' / 'zuu' / 'lae' / 'lue' / 'Beu' / 'Cau'
/ 'Dio' / 'Foa' / 'Kao' / 'Jui' / 'Neu' / 'Pou' / 'Goa'
/ 'Sau' / 'Veu' / 'Zua' / 'Zue' / 'Zui' / 'Zuo' / 'Zuu'
 / 'Lae' / 'Lue') __LWbreak)
```

107. This just assigns the Name class defined above the name DJAN used for it in the main grammar.

```
DJAN <- Name
```

108. NOTE: this class is not used.

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

109. NOTE: this class is not used.

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

110. The "article" used for numerical arguments such as **lio ne**, One.

```
LIO <- (__LWinit ('lio' / 'Lio') __LWbreak)
```

111. A particle with several different functions. It is most often the marker of the "timeless" tense.

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

112. A word with various uses having to do with grouping – a left marker.

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

113. Another grammar class generating the same word. NOTE: can be conflated with the previous one.

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

114. A right closure mark for certain groups started with **ge**. **cue** is outdated (in the original text of L1 but superseded by **geu** in appendix H).

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

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

115. A particle used for fronting final arguments. NOTE: this gives final arguments of predicates a special status which is perhaps dubious. Various solutions are possible.

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

116. A grouping particle often thought of as a spoken colon (:).

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

117. The spoken comma, with pause as an allophone. In many contexts the pause (comma orthographically) cannot be construed as of class GU because it is identified first as a phonological pause (after a name or before A words) or as a structure word break. The use of pause as an allophone of GU is in fact restricted to positions directly preceding or directly following predicate words; if the need for structure word breaks (LWbreak class) could be eliminated, then pauses could also be construed as GU in many positions between structure words. This use of pauses grammatically is a feature of TLI Loglan not found in Lojban.

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

118. Right boundary marker for JI- and related clauses.

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

119. Right boundary marker for LEPO clauses.

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

120. Right boundary marker for termsets.

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

121. Same as the previous NOTE: to be conflated.

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

122. Same as the previous NOTE: to be conflated.

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

123. Vocative marker. Note that a recent Keugru decision makes marking of vocatives with **hoi** mandatory.

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

124. Logical connectives acting between sentences.

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

125. A form of the spoken period I with a special grouping function.

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

126. ie, LW L4 '75 1.0 (q) What did you say?, a request for repetition when used alone. E.g., Ie? What did you say? (q) which?/who?/what?, a request for further identification when preceding some argument. E.g., Ie le mrenu? Which man?/Which of the men?

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

127. Another form of the spoken period I with a special grouping function.

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

128. A particle for tying arguments very tightly to a predicate .

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

129. A class of particles which form subordinate clauses with arguments.

```
JI <- (__LWinit ('jie' / 'jae' / 'pe' / 'ji'
/ 'ja' / 'Jie' / 'Jae' / 'Pe' / 'Ji' / 'Ja') __LWbreak)
```

130. A class of particles forming subordinate clauses with sentences.

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

131. signals "scare quote" non-literal use of a word. right figurative quote.

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

132. Particle for very tight binding of a final argument to a predicate.

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

133. **ka** particles (forethought logical connectives) in word-component form.

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

134. causal connectives in word-component form.

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

135. The **ka** class words. NOTE: get rid of the second NOI.

A KA word may begin with NU only if it begins NUKU (LIP appears to allow other NU forms). It may be followed by optional NOI then optional PA word not beginning with NU or a causal connective. A KAPA...KI... phrase implements the corresponding APA connective.

NOTE: LIP experiment indicates that I should have not !((KOU / 'nu')) PA) but !(KOU) PA0 here. Of course this does not coordinate with the APA words which have quite unrestricted compounded PA forms including KOU initials. How to handle this? There are worse problems: LIP also wants to absorb PA words into KI which makes no sense to me (but see below that I did provide these KIPA words!)

Maybe APA transforms to KA ... KIPA ... But this would suggest that we do not need KAPA words at all.

```
KA <- (__LWinit &(caprule) ((('nu' / 'Nu') &([\'ku])))?
KA0 ('noi')? ((!((KOU / 'nu')) PA))? ('noi')? __LWbreak)
```

136. Forethought words (analogous to KA words) for the causal connectives.

```
KOUKI <- (__LWinit &(caprule) (('nu' / 'Nu'))?
__LWinit (('no' / 'No'))? __LWinit KOU 'ki' __LWbreak)
```

137. The top level KA-word class.

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

138. NOTE: redundant with KA3 above

```
KA1 <- KA3
```

139. Redundant with KA3 above.

```
KA2 <- KA3
```

140. The form separating the components of a forethought connective expression. NOTE: problems with role of PA here. LIP allows KI to absorb PA words – what are the semantics? LIP does not allow compound PA words here. Fpr the whole APA/KAPA/KIPA problem, I need a solution or perhaps two (one approximating LIP and one that I think ought to be used). The NOI should be before the PA anyway.

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

141. Left spoken parenthesis (creates a freemod)

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

142. Right spoken parenthesis (creates a freemod)

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

143. article for Linnaean names

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

144. left hand marker of an unordered set or an ordered list

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

145. Quotation mark modifiers defined in appendix H – written vs spoken expressions. Not necessarily a good idea.

NOTE: there is a grouping error here re the !Oddvowel

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

146. The class of quoted Loglan utterances **li** utterance0 **lu**

Funny business at the front about Quotemod. NOTE: analyze the stuff about commas and spaces, but I think it is correct. The LI cannot be followed by a vowel; it will be followed by possibly a quote modifier, possibly a comma and one or more spaces, or just one or more speace, then an utterance, then LU.

```
LI <- (__LWinit ('li' / 'Li') !(V2) (Quotemod)?
((([,])? ([ ])+))? utterance0 __LWinit 'lu')
```

147. One or more spaces followed by a string of nonspaces.

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

148. My entirely new strong quotation mechanism. Requires detailed documentation. The basic idea is that **lie** followed without a space by an optional numeral then an optional quotemod then a space then a string without spaces quotes the string with no spaces, possibly multiply as indicated by the numeral. Subsequent units of the same form with **cii** instead of **lie** append additional quotation blocks with intervening spaces. This will be the subject of a separate proposal with examples. The original strong quotation mechanism cannot be implemented with a PEG grammar.

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

149. NOTE: I do not know that this class is used.

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

150. This is the phonological definition of a possible compound structure word, derived from NB3. Its only use is as a case in single-word quotation.

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

151. Single word quotation. LIU quotes a single name word, predicate word (note that ZAO constructions are single words), or phonologically possible compound structure word. NOTE: add NIU as an option for the article, signalling a non-Loglan word. A quoted compound structure word must end with a comma unless it is followed by end of file, a period class punctuation mark, or a predicate. This is because our criteria for a compound cmapua are entirely phonetic. LIP apparently allows LIU to quote only well-formed structure words, which is hard to implement and not necessarily the best thing to do.

    LII makes letter names from TAI words.

```
LIU1 <- (__LWinit ((('liu' / 'Liu') !(V2) (Quotemod)?
((([,])? ([ ])+))? (Name / Predicate
/ (LW (([,] ([ ])+ !([,])) / &(period) / !(.)
/ &((([ ])* Predicate))))))
/ ('lii' (Quotemod)? TAI __LWbreak)))
```

152. right hand closures of sets and lists.

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

153. particle which creates freemods from predicates – imagined acts of the speaker (smilies)

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

154. particle which turns arguments into predicates

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

155. The class of little words which can start LEPO clauses. NOTE: NI0 should be replaced with something closer to NI.

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

156. The negative operator

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

157. Conversion operators which reorder arguments of predicates

```
NUO <- (('nuo' / 'fuo' / 'juo' / 'Nuo' / 'Fuo'
/ 'Juo' / 'nu' / 'fu' / 'ju' / 'Nu' / 'Fu' / 'Ju') !(Oddvowel))
```

158. NI items as full words (mathematical expressions)

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

159. Conversion operators as full words NOTE: only NU and NUO actually take numerical suffixes, and these will be concrete numerals – though a conversion operator choosing the last argument is probably supported – nura.

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

160. The single syllable tense words, in the broadest sense.

```
PAO <- (__LWinit ('gia' / 'gua' / 'pia' / 'pua'
/ 'Gia' / 'Gua' / 'Pia' / 'Pua' / 'nia' / 'nua' / 'biu'
/ 'fea' / 'fia' / 'fua' / 'via' / 'vii' / 'viu' / 'ciu'
/ 'coi' / 'dau' / 'Nia' / 'Nua' / 'Biu' / 'Fea' / 'Fia'
/ 'Fua' / 'Via' / 'Vii' / 'Viu' / 'Ciu' / 'Coi' / 'Dau'
 / 'dii' / 'duo' / 'foi' / 'fui' / 'gau' / 'hea' / 'kau'
 / 'kii' / 'kui' / 'lia' / 'lui' / 'mia' / 'Dii' / 'Duo'
/ 'Foi' / 'Fui' / 'Gau' / 'Hea' / 'Kau' / 'Kii' / 'Kui'
/ 'Lia' / 'Lui' / 'Mia' / 'mou' / 'nui' / 'peu' / 'roi'
/ 'rui' / 'sea' / 'sio' / 'tie' / 'Mou' / 'Nui' / 'Peu'
/ 'Roi' / 'Rui' / 'Sea' / 'Sio' / 'Tie' / 'Va' / 'Vi'
/ 'Vu' / 'Pa' / 'Na' / 'Fa' / 'va' / 'vi' / 'vu' / 'pa'
/ 'na' / 'fa' / KOU) !(Oddvowel))
```

161. The causal operators adorned with NU and/or NO, also basic tense words

```
KOU1 <- (('nu' / 'Nu' / 'no' / 'No' / 'nuno' / 'Nuno') KOU)
```

162. Complex tense words. The initial quantifier allows formation of words like **rana**. I should analyze these forms. I think I am right that it makes no sense to allow that construction to enter into compounds. After this one has strings of tense syllables linked by CA class connectives adorned with no before or noi after (the no before is a novelty), and a final ZI class suffix. These are somewhat more words than are allowed by LIP, but basically the same complex.

    NOTE: It does not really make sense for RANA words to be used as relative operators; they are strictly tenses. RANA is an abbreviation for NA RABA, it has an argument already. LIP allows rana to be used as a relative operator, which really doesnt make sense. Should a grammatical distinction be drawn here?

```
#I'm allowing a lot more here than LIP allows

PA <- (__LWinit &(caprule) ((NI !(Oddvowel)))?
((KOU1 / PA0))+ ((('no')? CA0 ('noi')?)
 ((KOU1 / PA0))+))* (ZI)? !(Oddvowel))
```

163.
```
PA constructions as complete words  NOTE:  remove subsequent redundant cla


PA2 <- (__LWinit PA __LWbreak)
```

164. `PA3 <- PA2`

165. `PA4 <- PA2`

166. `LE <- (__LWinit ('lea' / 'leu' / 'loe' / 'lee' / 'laa'`
`/ 'Lea' / 'Leu' / 'Loe' / 'Lee' / 'Laa' / 'le' / 'lo'`
`/ 'Le' / 'Lo' / 'la' / 'La') ((DA1 / TAI0))? (PA)? __LWbreak)`

167.

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

168.

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

169.

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

170.

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

171.

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

172.

```
PAUSE <- ([,] ([ ])+ !(CANCELPAUSE) !([,])
!((A / ICI / ICA / IGE / I)) !(Name) !((&(V2) Predicate)))
```

173.

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

174.

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

175.

```
PREDA <- (([ ])* &(caprule) (Predicate
/ 'he' / 'dua' / 'dui' / 'bua' / 'bui' / ('He' !(Oddvowel))
 / 'Dua' / 'Dui' / 'Bua' / 'Bui'
/ Acronym / (!([ ]) NI RA)) !((A / ICI / ICA / IGE / I))
 ((',' ([ ])* &((A / ICI / ICA / IGE / I))))? (freemod)?)
```

176.

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

177.

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

178.

```
UIO <- (('ua' / 'ue' / 'ui' / 'uo' / 'uu'
/ 'oa' / 'Ua' / 'Ue' / 'Ui' / 'Uo' / 'Uu'
/ 'Oa' / 'oe' / 'oi' / 'ou' / 'ia' / 'ii' / 'io'
 / 'iu' / 'ea' / 'ei' / 'eo' / 'Oe' / 'Oi' / 'Ou'
/ 'Ia' / 'Ii' / 'Io' / 'Iu' / 'Ea' / 'Ei' / 'Eo'
/ 'eu' / 'ae' / 'ai' / 'ao' / 'au' / 'bea'
 / 'buo' / 'cea' / 'cia' / 'Eu' / 'Ae' / 'Ai'
/ 'Ao' / 'Au' / 'Bea' / 'Buo' / 'Cea' / 'Cia'
 / 'coa' / 'dou' / 'fae' / 'fao' / 'feu' / 'gea'
/ 'kuo' / 'kuu' / 'Coa' / 'Dou' / 'Fae' / 'Fao'
 / 'Feu' / 'Gea' / 'Kuo' / 'Kuu' / 'rea' / 'nao'
/ 'nie' / 'pae' / 'piu' / 'saa' / 'sui' / 'taa'
/ 'Rea' / 'Nao' / 'Nie' / 'Pae' / 'Piu' / 'Saa'
```

```
 / 'Sui' / 'Taa' / 'toe' / 'voi' / 'zou' / 'loi' / 'loa'
/ 'sia' / 'sii' / 'Toe' / 'Voi' / 'Zou' / 'Loi' / 'Loa'
/ 'Sia' / 'Sii' / 'siu' / 'cao' / 'ceu' / 'Siu'
/ 'Cao' / 'Ceu') !(Oddvowel))
```

179.

```
UI <- (__LWinit ((UIO UIO) / UIO / (PA 'hu') / (NI 'fi')) __LWbreak)
```

180.

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

181.

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

182.

```
##The implementation of trial.85

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

183.

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

184.

```
GUE <- 'gue'
```

185.

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

186.

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

187.

```
lua <- LUA
```

188.

```
geu <- GEU
```

189.

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

190.

```
juelink <- (JUE (freemod)? term (freemod)?)
```

191.

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

192.

```
links <- (((links1 / (&((((KA1 (freemod)?))+ links1))
(KA1 (freemod)? links KI (freemod)? links1)))
 ((A2 (freemod)? links1))*) (freemod)?)
```

193.

```
#Now takes terms instead of just arguments

jelink <- ((JE (freemod)? term) (freemod)?)
```

194.

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

195.

```
linkargs <- (((linkargs1 / (&((((KA1 (freemod)?))+ linkargs1))
 KA1 (freemod)? linkargs KI (freemod)? linkargs1))
((A2 (freemod)? linkargs1))*) (freemod)?)
```

196.

```
predunit1 <- ((SUE
/ (NU (freemod)? GE (freemod)? despredE (geu)?)
/ (NU (freemod)? PREDA) / (GE (freemod)? descpred (geu)?)
/ (ME (freemod)? argument (gap)?) / PREDA) (freemod)?)
```

197.

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

198.

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

199.

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

200.

```
kekpredunit <- (&((((KA1 (freemod)?))+
 (PA / NO1 / PO / BI / CUI / predunit1)))
 ((NO1)* KA1 (freemod)? predicate KI (freemod)? predicate) (freemod)?)
```

201.

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

202.

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

203.

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

204.

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

205.

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

206.

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

207.

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

208.

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

209.

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

210.

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

211.

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

212.

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

213.

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

214.

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

215.

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

216.

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

217.

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

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

218.

```
descriptn <- (((LE (freemod)? descpred)
/ (LE (freemod)? mex (freemod)? descpred)
/ (LE (freemod)? arg1 descpred)
/ (LE (freemod)? mex (freemod)? arg1a)
/ (GE2 (freemod)? mex (freemod)? descpred)) (freemod)?)
```

219.

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

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

220.

```
arg1 <- (((LEPO (freemod)? uttAx (guo)?)
 / (LEPO (freemod)? sentence (guo)?)
 / (LIO (freemod)? descpred (gap)?)
/ (LIO (freemod)? term (gap)?)
/ (LIO (freemod)? mex (gap)?)
/ (([ ])* ('la' / 'La') name (gap)?)
/ (LA (freemod)? name (gap)?)
/ (descriptn (name)? (gap)?)
/ LIU1 / LIE / LI / LAO) (freemod)?)
```

221.

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

222.

```
argmod1 <- (((JI (freemod)? predicate (gui)?)
/ (JIO (freemod)? sentence (gui)?)
/ (JIO (freemod)? uttAx (gui)?)
/ (JI (freemod)? modifier)
/ (JI (freemod)? argument)) (freemod)?)
```

223.

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

224.

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

225.

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

226.

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

227.

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

228.

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

229.

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

230.

```
arg5 <- ((arg4 /
```

```
(KA3 (freemod)? argument KI (freemod)? argx)) (freemod)?)
```

231.

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

232.

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

233.

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

234.

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

235.

```
argument <- ((((LAU wordset) / (arg8 AGE (freemod)? arg8) / arg8)
 ((GUU (freemod)? argmod (gap)?))*) (freemod)?)
```

236.

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

237.

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

238.

```
word <- ((arg1a (gap)?) / (UI (gap)?) / (NI (gap)?)
 / (PA2 (gap)?) / (DIO (gap)?) / (predunit1 (gap)?) / indef2)
```

239.

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

240.

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

241.

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

242.

```
termset2 <- (((termset1
/ (KA3 (freemod)? termset2 KI (freemod)? termset1))
 ((A4 (freemod)? termset1))*) (freemod)?)
```

243.

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

244.

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

245.

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

246.

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

247.

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

248.

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

249.

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

250.

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

251.

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

252.

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

253.

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

254.

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

255.

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

256.

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

257.

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

258.

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

259.

```
keksent <- (((NO1 (freemod)?))*
((KA3 (freemod)? sentence KI (freemod)? uttA1)
/ (KA3 (freemod)? gap sentence KI (freemod)? uttA1)
/ (KA3 (freemod)? headterms sentence KI (freemod)? uttA1)) (freemod)?)
```

260.

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

261.

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

262.

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

263.

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

264.

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

265.

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

266.

```
uttA1 <- ((sen1 / uttAx / NO1 / links / linkargs
/ argmod / (terms keksent) / terms / uttA) (freemod)? (period)?)
```

267.

```
freemod <- ((UI / (SOI (freemod)? descpred (gap)?) / DIE
/ (NO1 DIE) / (KIE utterance0 KIU) / (HUE name (gap)?)
/ (HUE (freemod)? name (gap)?) / (HUE (freemod)? statement (gap)?)
/ (HUE (freemod)? terms (gap)?) / voc / JO
/ ([,] ([ ])+ CANCELPAUSE)) (freemod)?)
```

268.

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

269.

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

270.

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

271.

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

272.

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

273. NOTE: error corrected 10/5. The order wasnt parallel with utterance.

```
utterance0 <- ((freemod utterance0)
/ freemod / (uttE IGE utterance0) / (I (freemod)?)
/ uttF / (I (freemod)? uttF) / (ICA (freemod)? uttF))
```

274.

```
utterance <- ((freemod utterance) / (freemod !(.))
/ (uttE IGE utterance)
/ (I (freemod)? !(.)) / (uttF !(.)) / (I (freemod)? uttF !(.))
/ (ICA (freemod)? uttF !(.)))
```