1 Appendix: Annotated Provisional PEG Grammar (an old version)

This is the grammar of 2/13/2016 (with subsequent piecemeal edits): enough changes had been made that I thought piecemeal editing wasn't suitable at this point. However, the inserted comments are mostly the original ones. A good deal of decluttering of both the grammar itself and the comments has happened since the last total update of the appendix.

```
lowercase <- (!([qwx]) [a-z])

uppercase <- (!([QWX]) [A-Z])

letter <- (!([QWXqwx]) [A-Za-z])

juncture <- (([-] &(letter)) / ([\'*] !(juncture)))
```

The Loglan alphabet. **q, w, x** are excluded.

```
stress <- ([\'*] !(juncture))

juncture2 <- ((([-] &(letter)) /
 ([\'*] !((([ ])* Predicate)) ((', ' ([ ])* &(Predicate)))?)) !(juncture))
```

Syllable breaks (including stress markers). The hyphen must be followed by a letter. The stress markers may not be followed by another juncture. The form `juncture2` is used to enforce the rule that a stress at the end of a cmapua and preceding a predicate must be followed by a comma pause.

```
Lowercase <- (lowercase / (juncture (letter)?))
```

A lowercase letter or a juncture (possibly followed by a capital letter) ; what can occur in a word after the initial letter which might be capitalized.

```
Letter <- (letter / juncture)
```

An upper-case letter or a juncture.

```
comma <- ([,] ([ ])+ &(letter))

comma2 <- (([,])? ([ ])+ &(letter))

end <- ((([ ])* '#' ([ ])+ utterance) / (([ ])+ !(.)) / !(.))

period <- (([!.:;?] (&(end) /
(([ ])+ &(letter)))) ((&(HUE) freemod (period)?))?)
```

Punctuation. A comma must be followed by whitespace followed by a letter. The class `period` is inhabited by terminal punctuation, all equivalent for the grammar so far, which may be followed either by the end of text or whitespace followed by a letter. An element of class `period` may have an appended inverse vocative (optionally followed by another `period`: installed 12/21/2015 to support Alex Leith's usage in his novel.

The silence/change of voice marker `#`, which we take from JCB's notation, is not really a piece of Loglan punctuation. It can never be quoted as part of Loglan text.

```
V1 <- [AEIOUYaeiouy]

V2 <- [AEIOUaeiou]
```

Classes of vowels. `V1` includes the irregular vowel **y**; `V2` is the class of regular vowels.

```
C1 <- (!(V1) letter)
```

The class of consonant letters.

```
Mono <- (([Aa] [o]) / (V2 [i]) / ([Ii] V2) / ([Uu] V2))

EMono <- (([Aa] [o]) / ([AEOaeo] [i]))
```

Classes of monosyllabic vowel pairs. `Mono` is the class of pairs which can be monosyllabic; `EMono` is the class of pairs which must be monosyllabic.

```
NextVowels <- (EMono / (V2 &(EMono)) / Mono / V2)
```

This class defines the next syllable to be extracted from a long chain of vowels in a name or borrowing. Exclusive monosyllables are chosen first, then single vowels which are followed by exclusive monosyllables, then optional monosyllables, then the next vowel.

```
BrokenMono <- (([a] juncture [o]) / ([aeo] juncture [i]))
```

This is an exclusive monosyllable forcibly broken by a juncture.

```
CVVSyll <- (C1 Mono)
```

A CVV syllable.

```
LWunit <- (((CVVSyll (juncture)? V2) /
 (C1 !(BrokenMono) V2 (juncture)? V2) /
 ([Zz] 'iy' (juncture)? ('ma')?) / (C1 V2)) (juncture2)?)
```

Phonetic units of a multisyllable cmapua according to the definition in NB3. The names **ziy** and **ziyma** of the letter **y** are included as irregular permissible units.

```
LW1 <- (((V2 V2) /
 (C1 !(BrokenMono) V2 (juncture)? V2) /
 (C1 V2)) (juncture)?)
```

The structural units of cmapua, VV, CVV independent of syllablehood, CV, with possibility of medial and final appearance of junctures. Where is this used? Is there any conflict with our recent introduction of some Cvv-V cmapua?

```
caprule <- ((uppercase / lowercase) ((('z' V1) /
lowercase / (juncture (caprule)?) / TAI0))* !(letter))
```

A sequence of letters and junctures obeying Loglan capitalization rules. A letteral may appear independently capitalized in such a sequence: this supports usages like `leSai` found in Leith and other Loglan texts. It does allow odd capitalizations of phonetic occurrences of letterals in other contexts. Similarly, a vowel may appear capitalized after lower-case z to support internal capitalization in acronyms, and any letter may appear capitalized after a juncture to support natural capitalization in certain compound names (and possibly other compound contexts).

```
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')
```

The consonant pairs which are allowed to appear in initial position in a syllable.

```
MaybeInitialCC <- (([Bb] (juncture)? [l]) /
 ([Bb] (juncture)? [r]) / ([Cc] (juncture)? [k]) /
 ([Cc] (juncture)? [l]) / ([Cc] (juncture)? [m]) /
 ([Cc] (juncture)? [n]) / ([Cc] (juncture)? [p]) /
 ([Cc] (juncture)? [r]) / ([Cc] (juncture)? [t]) /
 ([Dd] (juncture)? [j]) / ([Dd] (juncture)? [r]) /
 ([Dd] (juncture)? [z]) / ([Ff] (juncture)? [l]) /
 ([Ff] (juncture)? [r]) / ([Gg] (juncture)? [l]) /
([Gg] (juncture)? [r]) / ([Jj] (juncture)? [m]) /
([Kk] (juncture)? [l]) / ([Kk] (juncture)? [r]) /
 ([Mm] (juncture)? [r]) / ([Pp] (juncture)? [l]) /
 ([Pp] (juncture)? [r]) / ([Ss] (juncture)? [k]) /
([Ss] (juncture)? [l]) / ([Ss] (juncture)? [m]) /
([Ss] (juncture)? [n]) / ([Ss] (juncture)? [p]) /
 ([Ss] (juncture)? [r]) / ([Ss] (juncture)? [t]) /
 ([Tt] (juncture)? [c]) / ([Tt] (juncture)? [r]) /
 ([Tt] (juncture)? [s]) / ([Vv] (juncture)? [l]) /
 ([Vv] (juncture)? [r]) / ([Zz] (juncture)? [b]) /
 ([Zz] (juncture)? [v]) / ([Zz] (juncture)? [l]) /
 ([Ss] (juncture)? [v]))
```

A pair of consonants which is either an initial pair or an initial pair separated by a juncture.

```
NonmedialCC <- (([b] (juncture)? [b]) /
 ([c] (juncture)? [c]) / ([d] (juncture)? [d]) /
([f] (juncture)? [f]) / ([g] (juncture)? [g]) /
 ([h] (juncture)? [h]) / ([j] (juncture)? [j]) /
([k] (juncture)? [k]) / ([l] (juncture)? [l]) /
 ([m] (juncture)? [m]) / ([n] (juncture)? [n]) /
 ([p] (juncture)? [p]) / ([q] (juncture)? [q]) /
 ([r] (juncture)? [r]) / ([s] (juncture)? [s]) /
 ([t] (juncture)? [t]) / ([v] (juncture)? [v]) /
([z] (juncture)? [z]) / ([h] (juncture)? C1) /
([cjsz] (juncture)? [cjsz]) / ([f] (juncture)? [v]) /
 ([k] (juncture)? [g]) / ([p] (juncture)? [b]) /
([t] (juncture)? [d]) / ([fkpt] (juncture)? [jz]) /
 ([b] (juncture)? [j]) / ([s] (juncture)? [b]))
```

Pairs of consonants which are not permitted (with or without a syllable break).

```
NonjointCCC <- (([c] (juncture)? [d] (juncture)? [z]) /
 ([c] (juncture)? [v] (juncture)? [l]) /
 ([n] (juncture)? [d] (juncture)? [j]) /
 ([n] (juncture)? [d] (juncture)? [z]) /
([d] (juncture)? [c] (juncture)? [m]) /
 ([d] (juncture)? [c] (juncture)? [t]) /
 ([d] (juncture)? [t] (juncture)? [s]) /
 ([p] (juncture)? [d] (juncture)? [z]) /
 ([g] (juncture)? [t] (juncture)? [s]) /
([g] (juncture)? [z] (juncture)? [b]) /
 ([s] (juncture)? [v] (juncture)? [l]) /
([j] (juncture)? [d] (juncture)? [j]) /
 ([j] (juncture)? [t] (juncture)? [c]) /
 ([j] (juncture)? [t] (juncture)? [s]) /
 ([j] (juncture)? [v] (juncture)? [r]) /
([t] (juncture)? [v] (juncture)? [l]) /
 ([k] (juncture)? [d] (juncture)? [z]) /
 ([v] (juncture)? [t] (juncture)? [s]) /
 ([m] (juncture)? [z] (juncture)? [b]))
```

Triples of consonants forbidden, with or without a syllable break.

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

This is a device for ensuring that CV cmapua are not mistaken for initial segments of CVV cmapua. Under normal circumstances, a cmapua unit will not be followed by an odd number of vowels without a pause (junctures are ignored). The reason for this is that a stream of vowels which is not in initial position in a block of letters will be a string of VV attitudinals (UI words). So a little word will not be followed by a specimen of class Oddvowel (a stream of vowels of odd length).

This rule is still used, but not nearly as much, as it is now mandatory to pause before a compound VV attitudinal.

This is also useful for recognizing ends of complex predicates.

```
RepeatedVowel <- (([Aa] (juncture)? [a]) /
 ([Ee] (juncture)? [e]) / ([Oo] (juncture)? [o]) /
 ([Ii] juncture [i]) / ([Uu] juncture [u]))
```

Doubled vowel disyllables, notable because one of the vowels must be stressed and one unstressed. In the cases of **i** and **u** the juncture is not optional for this class, as the letter pairs admit a monosyllabic pronunciation.

```
RepeatedVocalic <- (([Mm] [m]) / ([Nn] [n]) / ([Ll] [l]) / ([Rr] [r]))

Syllabic <- [lmnr]

Nonsyllabic <- (!(Syllabic) C1)
```

Syllabic consonants. Where a consonant is used syllabically, it must be doubled. Individual potentially syllabic consonants are used below to define pairs of consonants forbidden in syllable-final position.

```
Badfinalpair <- (Nonsyllabic !('mr') !(RepeatedVocalic)
 Syllabic !((V2 / [y] / RepeatedVocalic)))
```

A bad final pair is a nonsyllabic consonant followed by a syllabic consonant but not by **mr** or a syllabic pair, the whole not being followed by a vowel or syllabic pair, and the two consonants not separated by a juncture. This amounts to a nonsyllabic consonant followed by a syllabic consonant in syllable-final position. The reason it would be bad is that it could really only be pronounced as another syllable.

```
FirstConsonants <- (((!((C1 C1 RepeatedVocalic))
 &(InitialCC) (C1 InitialCC)) /
(!((C1 RepeatedVocalic)) InitialCC) /
((!(RepeatedVocalic) C1) !([y]))) !(juncture))

FirstConsonants2 <- (((!((C1 C1 RepeatedVocalic))
&(InitialCC) (C1 InitialCC)) /
(!((C1 RepeatedVocalic)) InitialCC) /
 (!(RepeatedVocalic) C1)) !(juncture))
```

Permitted initial sequences of consonants in syllables. The general idea is that one has a sequence of one to three consonants, any adjacent pair in which is an initial pair, and not sharing a consonant with a syllabic consonant pair. The first version, which appears in syllables in predicates, may not be followed by **y**. Of course the initial consonants in a syllable are not followed by a juncture.

```
VowelSegment <- ((NextVowels !(RepeatedVocalic)) /
(!((C1 RepeatedVocalic)) RepeatedVocalic))

VowelSegment2 <- (NextVowels /
 (!((C1 RepeatedVocalic)) RepeatedVocalic))
```

The vowel segment component of a syllable. The first version, which appears in borrowed predicates, is either the next vowel syllable chosen according to the priority scheme given above or a syllabic consonant pair, and if it is a vowel or vowel pair it cannot be followed by a syllabic pair. In the second version, found in names, the restriction on being followed by a syllabic pair does not apply. A syllabic pair will not be followed by an occurrence of the same consonant.

```
SyllableA <- ((C1 V2 &(C1) !(Badfinalpair) (FinalConsonant)?
 ((!(Syllable) FinalConsonant))?) (juncture)?)

SyllableB <- ((FirstConsonants)? !(RepeatedVowel)
 !((&(Mono) V2 RepeatedVowel)) VowelSegment
 !(Badfinalpair) ((!(Syllable) FinalConsonant))?
 ((!(Syllable) FinalConsonant))? (juncture)?)
```

The general definition of a syllable, as found in a borrowed predicate. A syllable consists of an optional `FirstConsonants` followed by a vowel segment followed by zero one or two final consonants (defined below). There are side conditions: the vowel segment cannot be the first vowel in a repeated vowel pair, nor can it be a vowel pair followed by another occurrence of the second vowel in the pair, and if there are two final consonants, they cannot make up a `Badfinalpair` (because such a pair is impossible to pronounce except as another syllable: this rule is almost unique among my phonetic rules in not being found in some form in the original material–but also, it is nowhere violated except in names where we correct it by doubling the continuant).

The strictures on repeated vowels enforce the rule that the repeated vowel pairs which force stress cannot occur in borrowed predicates.

This version has careful dynamics: the intention is that if one is actually reading a pre-complex, one will respect the boundaries of the djifoa.

```
Syllable <- (SyllableA / SyllableB)

BrokenInitialCC <- (&(MaybeInitialCC) C1 juncture C1 &(V2))

JunctureFix <- ((InitialCC V2 BrokenInitialCC) /
 (((C1 V2))? V2 BrokenInitialCC) /
 (C1 V2 !stress juncture InitialCC V2 Letter) / (C1 BrokenInitialCC V2))
```

The rule `JunctureFix` is a technical device to prevent the parser from allowing borrowings which are illegal variants of complexes formed simply by moving syllable breaks. The point of this rule is to prevent complexes with

illegally placed syllable breaks from parsing as borrowings. Explicit syllable breaks of some of the kinds excluded do occur in complexes, and some of these syllable breaks may likely occur in speech in borrowings with no harm done; this is basically a rule of orthography rather than phonetics.

Complete details are in an essay in the text. This could not be hacked; it basically had to be written out as a proof to get the right form!

```
SyllableFinal1 <- ((FirstConsonants)? !(RepeatedVocalic)
VowelSegment !(stress) (juncture)?
 !(V2) (&(Syllable) / &([y]) / !(Letter)))

SyllableFinal2 <- ((FirstConsonants)? !(RepeatedVocalic)
 VowelSegment !(stress) (juncture)? (&([y]) / !(Letter)))

SyllableFinal2a <- ((FirstConsonants)? !(RepeatedVocalic)
 VowelSegment (juncture)? (&([y]) / !(Letter)))

SyllableFinal2b <- ((FirstConsonants)? !(RepeatedVocalic)
VowelSegment stress (&([y]) / !(Letter)))
```

The first two classes describe syllables which can be identified as (possibly) final in a borrowed predicate word because they are vowel-final, not explicitly stressed and followed by another syllable, `y` or a non-Letter, or definitely final because followed by `y` or a non-Letter.

The second two classes give the same descriptions modified for the case of a syllable (possibly) final in a borrowing affix. Syllables final in a borrowing affix are always followed by `y` and may be explicitly stressed: the classes given are not incorrect but could be simplified. Recall that a Letter is a letter or juncture.

```
StressedSyllable <- (((FirstConsonants)? !(RepeatedVowel)
 !((&(Mono) V2 RepeatedVowel)) VowelSegment
 !(Badfinalpair) (FinalConsonant)? (FinalConsonant)?) stress)
```

This is an explicitly stressed syllable appropriate for a borrowed predicate.

```
FinalConsonant <- (!(RepeatedVocalic)
 !(NonmedialCC) !(NonjointCCC) C1 !(((juncture)? V2)))
```

This is the class defining each of the two optional final consonants of a syllable in a borrowed predicate. They may not start a forbidden pair or triple of consonants. They may not start a syllable. They may not be followed by a juncture then a vowel. Both of these last two measures enforce the idea that syllables start as soon as possible. They will not start syllabic pairs.

```
Syllable2 <- (((FirstConsonants2)? (VowelSegment2 / [y])
 !(Badfinalpair) ((!(Syllable2) FinalConsonant2))?
 ((!(Syllable2) FinalConsonant2))?) (juncture)?)

FinalConsonant2 <- (!(RepeatedVocalic)
 !(NonmedialCC) !(NonjointCCC) C1 !(((juncture)? V2)))
```

These classes define syllables in names and the final consonant class in names. The main difference is that the vowel segment may be **y** and various context restrictions on the vowel segments in borrowed predicates are lifted: the vowel segment may overlap a repeated vowel pair and may be followed by a syllabic pair. The rules governing the final consonant are the same, except that the kind of syllable it cannot start is the one appropriate to names.

The preceding block of rules defines the Loglan syllable, in two versions, one for borrowed predicate words and one for names, with some variations and subclasses specified.

The most general form of the Loglan syllable is the one which can appear in names. It consists of an optional permitted sequence of initial consonants, followed by a vowel segment or **y**, followed by one or two optional final consonants, which may not make up a pair consisting of a non-continuant followed by a continuant, followed by an optional juncture. The rules governing the final consonants are that a final consonant may not begin a forbidden sequence of two or three consonants from the lists above taken from NB3 (whether this continues into the next syllable or not, and ignoring junctures), it may not

begin a syllable (junctures are placed as far to the left as possible if not explicitly given) and it may not be followed by an explicit juncture followed by a vowel (a syllable will always incorporate at least one of a preceding block of consonants).

In borrowed predicate words, the additional conditions are imposed that the vowel segment may not be replaced by `y` and the last vowel in the vowel segment may not start a repeated vowel sequence which forces a stress.

A class of stressed syllables appropriate in borrowings is given.

Classes of syllables are given which can be identified as (possibly) final in a borrowed predicate word because they are vowel-final, not explicitly stressed and followed by another syllable, `y` or a non-letter, or definitely final because followed by `y` or a non-letter. Syllables final in a borrowing affix are always followed by `y` and may be explicitly stressed: the classes given are not incorrect but could be simplified.

Note use of VowelSegment2 in Syllable2 so that names may contain vowels followed by syllabic consonants, ruled out now in predicates.

```
Name <- (([ ])* &(((uppercase / lowercase)
 ((!((C1 (stress)? !(Letter))) Lowercase))* C1 (stress)?
 !(Letter) (&(end) / comma / &(period) / &(Name) / &(CI))))
 ((Syllable2)+ (&(end) / comma / &(period) / &(Name) / &(CI))))
```

This is the definition of the class of name words. A name word must resolve into syllables of class `Syllable2`. It must follow the capitalization conventions. Its final letter must be a consonant, which may be followed optionally by an explicit stress mark, which may further be followed by a `comma` (included in it). If it does not end with a `comma`, it must be followed by one of the following items not included in it: end of text, a `period`, another `Name` (noting that a name includes initial whitespace), or the cmapua **ci**.

```
CCSyllableB <- (((FirstConsonants)? RepeatedVocalic
 !(Badfinalpair) ((!(Syllable) FinalConsonant))? ((!(Syllable)
 FinalConsonant))?) (juncture)?)
```

This is the class of syllables in borrowings with syllabic pairs as their vowel segments.

```
BorrowingTail <- ((!(JunctureFix) !(CCSyllableB) StressedSyllable
((!(StressedSyllable) CCSyllableB))? !(StressedSyllable) SyllableFinal1) /
 (!(CCSyllableB) !(JunctureFix) Syllable
((!(StressedSyllable) CCSyllableB))? !(StressedSyllable) SyllableFinal2))
```

A borrowing tail is a final segment of a borrowed predicate, consisting of a stressed syllable, not containing a syllabic pair, followed optionally by an unstressed syllable with a syllabic pair as its vowel segment, followed by a required unstressed final syllable of one of the kinds above.

This is implemented either as an explicitly stressed syllable followed optionally by an unstressed syllabic consonant pair syllable, followed by a `SyllableFinal1`, which is merely possibly final (vowel final, not explicitly stressed, followed by another syllable or a **y** or a non-Letter), or as a mere syllable followed by an optional syllabic pair syllable followed by a `SyllableFinal2` (which is known to be final because followed by **y** or a non-Letter). Recall that the class of Letters includes junctures.

The leading syllable in a borrowing tail will not be in the `JunctureFix` class.

```
PreBorrowing <- (((!(BorrowingTail) !(StressedSyllable) !(JunctureFix)
 !((CCSyllableB CCSyllableB)) Syllable))* !(CCSyllableB) BorrowingTail)
```

A pre-borrowing is a sequence of syllables none of which are explicitly stressed or satisfy the `JunctureFix` condition or are initial in a borrowing tail, followed by a borrowing tail. It also may not contain two successive syllables with syllabic consonants as vowel segments. This class is used in the algorithm for finding the left end of a borrowing, below.

```
HasCCPair <- ((((C1)? ((V2 ((!(stress) juncture))?))+ !(Borrowing)
 !((&(MaybeInitialCC) C1 (!(stress) juncture)
 !(CCVV) PreBorrowing)) (stress)?))?
 C1 (juncture)? C1)
```

This class captures the condition on a pre-borrowing that it must contain a CC pair. A word of this class begins with an optional prefix which is either a consonant followed by one or more vowels or just one or more vowels. If the prefix is present and unstressed (tricky because the stress may fall after an intervening consonant), what follows it cannot be a borrowing on its own [in a phonetic transcript, dropping a finally stressed initial CV^n might leave a residue which could be read as a predicate, by reading further than intended!]. The rest of the word must begin with two consonants, which may be separated by a juncture. I have said in earlier versions that the two consonants could not be separated by a juncture if they were an initial pair, but this was an error.

If there is a juncture between the consonants, it checks whether what is after the juncture is a pre-borrowing.

```
CVCBreak <- (C1 V2 (juncture)? &(MaybeInitialCC)
 C1 (juncture)? &((PreComplex / ComplexTail)))
```

A word of this class begins with CVCC where the last C is initial in a pre-complex (see below) or complex tail, and the CC pair is initial (possibly broken by a juncture). Such words may not be predicates. The problem is that such a word is illegal as a complex, and we do not want it to be subsequently accepted as a borrowing. The only case where the rule possibly does something is where there is an explicit syllable break after the CVC, and I think that HasCCPair is now smart enough that this rule is redundant.

```
CCVV <- ((&(BorrowingTail) C1 C1 (C1)? V2 stress !(Mono) V2) /
 (&(BorrowingTail) C1 C1 (C1)?
V2 (juncture)? V2 (!(Letter) / ((juncture)? [y]))))
```

This is a word which would otherwise qualify as a borrowing of the form CCVV or CCCVV (these forms are excluded: banning them allows some slightly larger complexes to be formed without difficulties).

```
Borrowing <- (&(HasCCPair) !(CVCBreak) !(CCVV)
 !(((((C1)? (V2 (juncture)?) ((V2 (juncture)? &(V2)))+))? V2 (juncture)?
 MaybeInitialCC V2)) !(CCSyllableB)
((((!(BorrowingTail) !(StressedSyllable)
 !((CCSyllableB CCSyllableB)) !(JunctureFix) Syllable))*
 !(CCSyllableB) BorrowingTail))
```

A borrowed predicate cannot be of the classes `CVCBreak` or `CCVV`. It must be of class `HasCCPair`. It cannot begin VCCV where the CC is an initial pair (even if broken by a juncture). It cannot begin with a syllabic pair syllable. It then consists of a series of syllables which are not explicitly stressed, do not satisfy the `JunctureFix` condition, and are not initial in a borrowing tail, followed by a borrowing tail. [The condition that the borrowing tail not begin with a syllabic pair syllable appears to be redundant.]

I added the further condition, needed for the same reason that VCCV cannot be allowed, that (C)VVV^{n}CCV cannot be an initial sequence in a borrowing when the CC is initial.

We now give the formal definition of a borrowing. A borrowing is a pre-borrowing (look at the end of the rule). In addition, it either begins with a pair of consonants or has a (C)V^n (meaning an optional consonant followed by a sequence of vowels) followed by a pair of consonants which does not start a shorter borrowing itself. This rule both makes sure that there is a pair of adjacent consonants and prevents a cmapua from falling off the front of a borrowed predicate.

It does not begin with a CVCC-sequence where the CC is an initial consonant pair and peeling the CVC off would give a pre-complex (see below).

Nor can it begin with VCCV where the CC is an initial pair. The first syllable of a borrowing does not contain a syllabic consonant. A borrowing may not contain two successive syllables with syllabic consonants.

There are no CCVV or CCCVV borrowings.

Nothing in this definition is new, except for technical details of placement of syllable breaks, which do not contradict anything in the sources (in any significant way: JCB did write some very odd junctures).

```
PreBorrowingAffix <- ((((!(StressedSyllable) !(SyllableFinal2a)
 !((CCSyllableB CCSyllableB))
 !(JunctureFix) Syllable))+ SyllableFinal2a)
(juncture)? [y] !(stress) (juncture)? (([ ,] ([ ])*))?)
```

This is a preliminary definition of borrowing affix; this resolves into syllables, not stressed, not in `JunctureFix`, followed by a possibly stressed final syllable, followed by **y** and possibly an explicit pause. We permit stress on the final syllable of a borrowing affix; note that the placement is not the same as in the borrowing it is derived from. It also may not contain two successive syllables with syllabic consonants. This condition makes sense, and imposing it on borrowings and borrowing affixes prevents a kind of ill-formed borrowing affix from being accepted which would otherwise be hard to detect.

```
BorrowingAffix <- (&(HasCCPair) !(CVCBreak) !(CCVV)
 !(((((C1)? (V2 (juncture)?) ((V2 (juncture)?
 &(V2)))+))? V2 (juncture)? MaybeInitialCC V2))
 !(CCSyllableB) ((((!(StressedSyllable) !(SyllableFinal2a)
!((CCSyllableB CCSyllableB)) !(JunctureFix) Syllable))+
 SyllableFinal2a) (juncture)? [y] !(stress) (juncture)? (comma)?)
```

This class adds the restrictions that pick out the actual borrowing affixes from the wider class just defined. They are the same as the additional formal restrictions on borrowings.

A borrowing affix is exactly a borrowing followed by **y** (a ruling in Appendix H) but it differs from a borrowing in having its stress, if any, placed

finally. These are not new conditions; they are found in our sources. Separate but very similarly motivated rules are needed because of the different treatment of the stress.

```
StressedBorrowingAffix <- (&(HasCCPair) !(CVCBreak)
 !(CCVV) !(((((C1)? (V2 (juncture)?) ((V2 (juncture)?
&(V2)))+))? V2 (juncture)? MaybeInitialCC V2))
!(CCSyllableB) (((!(StressedSyllable) !(SyllableFinal2a)
 !((CCSyllableB CCSyllableB)) !(JunctureFix) Syllable))* SyllableFinal2b)
(juncture)? [y] !(stress) (juncture)? !([,]))
```

This is an explicitly stressed borrowing affix.

We now begin the definition of the class of complex predicates. These are peculiar in being built from djifoa ("affixes") rather than syllables. In defining possible syllable breaks in complexes, I followed the rule that a syllable in a complex predicate may not cross the boundary of a djifoa.

```
yhyphen <- ((juncture)? [y] !(stress) (juncture)? !([y]) &(letter))
```

This is a **y**-hyphen, used to fix sound conflicts at djifoa boundaries. It is unstressed, may be preceded and/or followed by actual juncture hyphens and must be followed by a letter after any juncture hyphen.

A y-hyphen is a buffering syllable which may be placed between djifoa to fix pronunciation problems. It is regarded as part of the preceding djifoa for parsing purposes. It is not stressed and will be followed (after an optional hyphen) by a letter other than `y`.

```
CV <- (C1 V2 !(stress) (juncture)? !(V2))
```

This defines the final unstressed CV syllable of a five letter djifoa. It will not be followed by a vowel or stress marker. It may be followed by a juncture.

```
Cfinal <- ((C1 yhyphen) / (!(NonmedialCC)
!(NonjointCCC) C1 !(((juncture)? V2))))
```

This is the final consonant of a CVC djifoa. It is either a consonant followed by a **yhyphen** or a consonant not beginning a forbidden medial pair or triple of consonants and not followed by a vowel (even with an intervening juncture).

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

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

This is a class of phonetic hyphens used to fix sound conflicts at boundaries between djifoa. This will be an **r** not followed by another **r** or by a vowel (including across junctures), an **n** followed by an **r**, or an unstressed **y**; it will be followed by a letter, possibly with intervening juncture, which will not be a **y**.

The class `noyhyphen` excludes the **y** phonetic hyphens, and does not absorb a following juncture.

```
StressedSyllable2 <- (((FirstConsonants)?
VowelSegment !(Badfinalpair) (FinalConsonant)?
 (FinalConsonant)?) stress (yhyphen)?)
```

This is an explicitly stressed syllable of a kind that can occur in a complex predicate. Repeated vowels are not restricted.

```
CVVStressed <- (((C1 &(RepeatedVowel)
 V2 !(stress) (juncture)? !(RepeatedVowel)
 V2 (noyhyphen)?) (juncture)? (yhyphen)?) /
(C1 !(BrokenMono) V2 !(stress) juncture
V2 (noyhyphen)? stress (yhyphen)?) /
 (C1 !(Mono) V2 V2 (noyhyphen)? stress (yhyphen)?))

CVVStressed2 <- (C1 Mono (noyhyphen)? stress (yhyphen)?)
```

The first class is the class of all finally stressed two-syllable CVV dijifoa. The first sort, with a repeated vowel, may be qualified as possibly finally stressed. The class BrokenMono applies to optional monosyllables which are forced to be two syllables by an explicit juncture. One has CVV djifoa here with optional hyphen in the middle but in any case certain to be two syllables, and either with explicit stress at the end or repeated vowels of the sort which force a stress.

The second class is the explicitly stressed CVV monosyllable

```
CVV <- (!((C1 V2 stress V2 (hyphen)? stress))
 ((C1 !(BrokenMono) V2 (juncture)?
 !(RepeatedVowel) V2 (noyhyphen)?)
 (juncture)? !(V2) (yhyphen)?))
```

This is a general CVV djifoa. It does not have two stressed syllables. It is not followed by another vowel the same as its final vowel.

```
CVVFinal1 <- (C1 !(BrokenMono) V2 stress
 !(RepeatedVowel) V2 !(stress) (juncture)? !(V2))

CVVFinal2 <- (((C1 !(Mono) V2 V2) /
 (C1 !(BrokenMono) V2 juncture
 !(RepeatedVowel) V2)) !(Letter))

CVVFinal5 <- (((C1 !(Mono) V2 V2) /
 (C1 !(BrokenMono) V2 juncture V2))
 &(((juncture)? [y])))

CVVFinal3 <- (C1 &(Mono) V2 V2
 !(stress) (juncture)? !(V2))

CVVFinal4 <- (C1 Mono !(Letter))
```

These are various forms of CVV djifoa which must be or can be final in a complex. The first one is a disyllable with an explicit stress on the first syllable. The second is either a mandatory disyllable or a disyllable formed by an explicit juncture, not followed by a letter or juncture. The third (oddly labelled 5) is the same as the previous but followed by **y** (with a possible intervening juncture). The fourth (oddly labelled 3) is an unstressed possible monosyllable (this one is a merely possible final syllable; the others must be final). The fifth (oddly labelled 4) is a possible monosyllable not followed by a letter or juncture.

```
CVC <- ((C1 V2 Cfinal) (juncture)?)

CVCStressed <- (C1 V2 !(NonmedialCC)
 !(NonjointCCC) C1 stress !(V2) (yhyphen)?)
```

The first class is the class of CVC djifoa. The second is the class of stressed CVC djifoa, possibly followed by a yhyphen which would be included.

```
CCV <- (InitialCC !(RepeatedVowel)
 V2 (juncture)? !(V2) (yhyphen)?)

CCVStressed <- (InitialCC
 !(RepeatedVowel) V2 stress !(V2) (yhyphen)?)

CCVFinal1 <- (InitialCC
 !(RepeatedVowel) V2
 !(stress) (juncture)? !(V2))

CCVFinal2 <- (InitialCC V2 !(Letter))
```

These are forms of CCV djifoa. A CCV djifoa begins with an initial pair, is not followed by another copy of the same vowel, may be followed by and include a yhypen and/or a juncture. The second form is an explicitly stressed form satisfying the same conditions. The last two are possibly final forms: the first is simply unstressed, the second (which must be final) is not followed by a letter or juncture.

Laying to rest a silly remark made in older documents, a CCV djifoa may require a **y** hyphen, but only when standing before a borrowing djifoa.

```
CCVCVMedial <- (InitialCC V2 (juncture)?
C1 [y] !(stress) (juncture)? &(letter))

CCVCVMedialStressed <- (CCV
stress C1 [y] !(stress) (juncture)? &(letter))

CCVCVFinal1 <- (InitialCC V2 stress CV)

CCVCVFinal2 <- (InitialCC V2 (juncture)? CV !(Letter))

CCVCVY <- (InitialCC V2 (juncture)? CV [y])
```

These are the CCVCV five letter djifoa. The first two are the medial four-letter djifoa, with the final vowel replaced with **y**. These must be followed

by a letter, possibly with an intervening juncture. The last two are five-letter final forms, one explicitly stressed and one not followed by a letter or juncture.

```
CVCCVMedial <- (C1 V2 ((juncture &(InitialCC)))?
 !(NonmedialCC) C1 (juncture)? C1 [y]
 !(stress) (juncture)? &(letter))

CVCCVMedialStressed <- ((C1 V2 (stress &(InitialCC))
 !(NonmedialCC) C1 C1 [y] !(stress) (juncture)? &(letter)) /
(C1 V2 !(NonmedialCC) C1 stress C1
[y] !(stress) (juncture)? &(letter)))

CVCCVFinal1a <- (C1 V2 stress
 InitialCC V2 !(stress) (juncture)? !(V2))

CVCCVYa <- (C1 V2 (juncture)? InitialCC
 V2 !(stress) (juncture)? [y])

CVCCVFinal1b <- (C1 V2
 !(NonmedialCC) C1 stress CV)

CVCCVYb <- (C1
 V2 !(NonmedialCC) C1 (juncture)? CV [y])

CVCCVFinal2 <- (C1 V2
 ((juncture &(InitialCC)))? !(NonmedialCC) C1
(juncture)? CV !(Letter))

FiveLetterY <- (CCVCVY / CVCCVYa / CVCCVYb)
```

These are the CVCCV five letter djifoa. The first two are the medial forms. The optional juncture between the syllables can be placed before the two consonants if they are an initial pair, and in any case between. The last are final five-letter forms, either explicitly stressed (in the two possible ways) or not followed by a letter or juncture. The Y forms are part of the mechanism to exclude borrowing djifoa of these forms.

```
GenericFinal <- (CVVFinal3 / CVVFinal4 / CCVFinal1 / CCVFinal2)

GenericTerminalFinal <- (CVVFinal4 / CCVFinal2)
```

These are final or possibly final monosyllables. The second one includes only those which are not followed by a letter or juncture and so must be final.

```
Affix1 <- (CCVCVMedial / CVCCVMedial / CCV / CVV / CVC)
```

These are the non-borrowing djifoa.

```
Peelable <- (&(PreBorrowingAffix) !(CVVFinal1) !(CVVFinal5)
 Affix1 (!(Affix1) /
 &((&(PreBorrowingAffix)
!(CVVFinal1) !(CVVFinal5) Affix1
 !(PreBorrowingAffix) !(Affix1))) / Peelable))
```

The peelable djifoa at the beginning of a borrowing affix are a string of Affix1's each of which is initial in a pre-borrowing-affix, none of which are final in a borrowing affix or followed by an Affix1 final in a borrowing affix. Any peelable djifoa in this sense are actually fake: if we see no Affix1 or a peelable Affix1 we are at the start of a borrowing affix.

The details are horrible: the upshot is that if one has a Peelable initial Affix1 or no initial Affix1, the apparent borrowing affix which follows is not resolvable into djifoa. Something which looks like a borrowing affix, has a front Affix1 but does not have a peelable front Affix1 is in fact resolvable into Affix1's and not a borrowing affix at all. Same remarks for Peelable2 below.

I shall try to narrate this. A `Peelable` starts with an `Affix1` (one of the non-borrowing medial djifoa). It cannot be a `CVVFinal1` or a `CVVFinal5` (it will not by hypothesis be final in the borrowing affix in which it is initial at all; these are two syllable forms which could be the entire tail of the pre-borrowing affix minus the final **y**). It is initial in a `PreBorrowingAffix`. It is

followed either (1) by something not an `Affix1` (so we can see immediately that the `PreBorrowingAffix` in which it is immersed does not resolve into djifoa) or (2) another `Affix1`, similarly starting a `PreBorrowingAffix` and not a `CVVFinal1` or `CVVFinal5` which is followed either by a non-`Affix1` or a non-`PreBorrowingAffix`, so once again we can see that the pre-borrowing affix in which our original djifoa was initial does not resolve into djifoa, or (3) an object of class `Peelable` (which is part of the `Peelable` string we read, unlike the objects in (1) and (2)), from which we can keep peeling initial affixes until we get behaviour (1) or (2) and see that the original pre-borrowing affix does not resolve into djifoa. In all cases, what we have is a pseudo-djifoa.

It is a very important point in the justification of `Peelable` that two successive `Affix1`'s which start pre-borrowing affixes will agree on where their respective pre-borrowing affixes end: because such an djifoa cannot include or be immediately followed by a **y** and a pre-borrowing affix ends with a **y** – as long as the first `Affix1` doesn't have two syllables with initial stress, so it can just be closed off with a **y** itself, which is why the two **CVV** classes are ruled out.

```
FiveLetterFinal <- (CCVCVFinal1 /
 CCVCVFinal2 / CVCCVFinal1a /
CVCCVFinal1b / CVCCVFinal2)

Peelable2 <- (&(PreBorrowing) !(CVVFinal1)
 !(CVVFinal2) !(CVVFinal5) !(FiveLetterFinal) Affix1
 !(FiveLetterFinal) (!(Affix1) / &((&(PreBorrowing)
 !(FiveLetterFinal) !(CVVFinal1) !(CVVFinal2) !(CVVFinal5) Affix1
 !(PreBorrowing) !(FiveLetterFinal) !(Affix1))) / Peelable2))
```

As Peelable, but for borrowings proper.

The narrative here is the same as above under `Peelable`: the difference is that more final forms are specificially excluded. Of course, the `Affix1` found here will not be final in the pre-borrowing in which it is initial. There is a situation in which a djifoa can appear to start a `Peelable2` when it is actually does not in intent: this is if it is the stressed syllable in a two-djifoa predicate and the following djifoa can be read as starting a non-complex borrowing.

It appears that this does no harm, however, since this will successfully be recognized as a `ComplexTail` and so will be read as a complex. An attempt to read it as a borrowing would give the same initial segment.

```
Affix <- ((!(Peelable) !(Peelable2) Affix1) /
(!(FiveLetterY) BorrowingAffix))

Affix2 <- (!(StressedSyllable2) !(CVVStressed) Affix)
```

Since we can exclude fake djifoa read from the fronts of borrowing affixes or borrowings, we can read Affixes (djifoa). Affix2 excludes stressed non-borrowing djifoa.

There are no borrowing djifoa based on the five letter shapes of primitives. The CVCCV forms would cause awful problems in initial position; there is no reason that either of the basic primitive shapes should occur as shapes of borrowing djifoa.

```
ComplexTail <- ((Affix GenericTerminalFinal) /
 (!((!(Peelable) Affix1)) !(FiveLetterY)
 StressedBorrowingAffix GenericFinal) /
 (CCVCVMedialStressed GenericFinal) /
 (CVCCVMedialStressed GenericFinal) /
 (CCVStressed GenericFinal) / (CVCStressed GenericFinal) /
 (CVVStressed GenericFinal) / (CVVStressed2 GenericFinal) /
(Affix2 CVVFinal1) / (Affix2 CVVFinal2) / CCVCVFinal1 /
CCVCVFinal2 / CVCCVFinal1a / CVCCVFinal1b / CVCCVFinal2 /
 (!((CVVStressed / StressedSyllable2)) Affix
 !((!(Peelable2) Affix1)) Borrowing !(((juncture)? [y])))
```

This catalogues the djifoa or pairs of djifoa which are guaranteed to terminate a complex, either because they explicitly end or because they are explicitly stressed. I don't believe I need to reproduce the catalogue of situations in English; I think they are readable from the rule.

I should check this – I discovered that the explicitly stressed CVV monosyllables had been overlooked, which broke some phonetic parses.

```
Primitive <- (CCVCVFinal1 / CCVCVFinal2 /
CVCCVFinal1a / CVCCVFinal1b / CVCCVFinal2)
```

This class is actually redundant (primitives turn out to be complexes), but it is nice to have the parser label the primitive five letter forms.

```
PreComplex <- (ComplexTail /
 ((!((CVCStressed / CCVStressed / CVVStressed /
ComplexTail / StressedSyllable2)) Affix) PreComplex))
```

This is a stream of djifoa none of which have a misplaced stress or start a ComplexTail, followed by a ComplexTail.

```
Complex <- (!((C1 V2 (juncture)? (V2)? (juncture)? CVV))
 !((C1 V2 !(stress) (juncture)? (V2)? !(stress) (juncture)?
 (Primitive / PreComplex / Borrowing / CVV)))
!((C1 V2 (juncture)? &(MaybeInitialCC) C1 (juncture)?
&((PreComplex / ComplexTail)))) PreComplex)
```

A complex is a pre-complex satisfying initial conditions: one of these is that a CV or CVV cannot fall off the front and leave a predicate; the other forbids CVCC- initial predicates with more than six letters with the CC initial.

The issue about CV or CVV falling off the front is tricky in a phonetic transcript; one wants to make sure it doesn't make a predicate by reading further. If the CV or CVV being dropped is the stressed syllable, this provides no evidence against its being a predicate, unless the next syllable is a CVV.

```
Predicate <- (((&(caprule) ((Primitive / Complex / Borrowing)
 ((([ ])* Z AO (', ')? ([ ])* Predicate))?)) /
 (C1 V2 (V2)? ([ ])* Z AO (comma)? ([ ])* Predicate))
!(((juncture)? [y])))
```

The full definition of a predicate word. Note that a borrowing is whatever is not a primitive or complex. Many though not all primitives and complexes would also parse as borrowings, and various tricky errors in complex predicate formation have manifested invisibly to ordinary use by parsing complexes as borrowings due to a bug. The option of building complex predicates from predicate words using `zao` is supported. It is worth noting that I do not require a pause after `zao` when it is followed by a vowel-initial borrowing, but the option of inserting a pause is supported.

A consonant initial cmapua can be affixed with ZAO to the front of a predicate.

It is important to note that there is no grouping with ZAO and there is no distinction between constructions with ZAO and the usual complexes, or indeed between either of these and mixed forms. Of course cmapua can be affixed to the front of a complex with zao which happen not to have corresponding djifoa.

A Predicate cannot be immediately followed by **y**, to avoid confusion with initial borrowing affixes.

We now begin the classification of cmapua.

```
Fourvowels <- (C1 V2 (juncture)? V2 (juncture)? V2 (juncture)? V2)

B <- (!(Predicate) !Fourvowels [Bb])

C <- (!(Predicate) !Fourvowels [Cc])

D <- (!(Predicate)!Fourvowels [Dd])

F <- (!(Predicate)!Fourvowels [Ff])

G <- (!(Predicate) !Fourvowels[Gg])

H <- (!(Predicate)!Fourvowels [Hh])

J <- (!(Predicate)!Fourvowels [Jj])

K <- (!(Predicate)!Fourvowels [Kk])

L <- (!(Predicate)!Fourvowels [Ll])

M <- (!(Predicate)!Fourvowels [Mm])

N <- (!(Predicate) !Fourvowels[Nn])

P <- (!(Predicate)!Fourvowels [Pp])

R <- (!(Predicate)!Fourvowels [Rr])

S <- (!(Predicate)!Fourvowels [Ss])

T <- (!(Predicate)!Fourvowels [Tt])

V <- (!(Predicate)!Fourvowels [Vv])

Z <- (!(Predicate)!Fourvowels [Zz])
```

Consonant letters irrespective of capitalization, initial in cmapua syllables so not starting a predicate.

```
a <-  [Aa] (juncture2)?) !V2

e <-  [Ee] (juncture2)?) !V2

i <- [Ii] (juncture2)?) !V2

o <- [Oo] (juncture2)?) !V2

u <-  [Uu] (juncture2)?) !V2

AA <- ([Aa] (juncture)? [a] (juncture2)?)
 (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
AE <- ([Aa] (juncture)? [e] (juncture2)?)
 (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
AI <- ([Aa] [i] (juncture2)?)  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
AO <- ([Aa] [o] (juncture2)?)
 (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
AU <- ([Aa] (juncture)? [u] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
EA <- ([Ee] (juncture)? [a] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
EE <- ([Ee] (juncture)? [e] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
EI <- ([Ee] [i] (juncture2)?)
 (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
EO <- ([Ee] (juncture)? [o] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
EU <- ([Ee] (juncture)? [u] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
IA <- ([Ii] (juncture)? [a] (juncture2)?)
 (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
IE <- ([Ii] (juncture)? [e] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
II <- ([Ii] (juncture)? [i] (juncture2)?)
```

```
 (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
IO <- ([Ii] (juncture)? [o] (juncture2)?)
 (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
IU <- ([Ii] (juncture)? [u] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
OA <- ([Oo] (juncture)? [a] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
OE <- ([Oo] (juncture)? [e] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
OI <- ([Oo] [i] (juncture2)?)
 (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
OO <- ([Oo] (juncture)? [o] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
OU <- ([Oo] (juncture)? [u] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
UA <- ([Uu] (juncture)? [a] (juncture2)?)
 (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
UE <- ([Uu] (juncture)? [e] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
UI <- ([Uu] (juncture)? [i] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
UO <- ([Uu] (juncture)? [o] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
UU <- ([Uu] (juncture)? [u] (juncture2)?)
  (&(V2 (juncture)? !(V2)) / !(Oddvowel)))
```

Vowels and vowel groups independent of capitalization and presence of medial or final junctures. Note that `juncture2` is used here in final position, because these classes are used to build cmapua syllables.

The vowel in a CV cmapua syllable cannot be followed directly without pause by another vowel which would make a mandatory monosyllable. This defends the Cvv-V cmapua units. No comma is required to mark this pause but at least whitespace must appear.

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

This is a marker for the beginning of a freestanding cmapua (structure word). Whitespace will be scanned over initially. The class excludes fake unit cmapua which are actually the beginning of predicates. The capitalization convention is enforced.

```
__LWbreak <- (!(Oddvowel) !((!((([ ])* Predicate))
 !((&(nonamemarkers) Name)) (A / ICI / ICA / IGE / I)))
((comma &((!((&(nonamemarkers) Name)) (V1 / A)))))?)
```

This class is used to mark the ends of cmapua and some other words. It enforces the condition that the word is not followed without a space by an odd number of vowels. Most of its business is to ensure that vowel initial words which are not predicates (which will be members of certain classes of logical and utterance connectives, which are listed, will not follow a word without being preceded by an explicit pause. Further, the class allows an explicit pause to follow the word (included in the word for parsing purposes) before any vowel-initial or A word (not all A words are vowel initial, but the non-vowel-initial ones (such as `noa`) must nonetheless be preceded by pauses). Checks have been added to ensure that pauses before vowel initial names are treated correctly.

```
CANCELPAUSE <- (comma (('y' comma) / (C UU __LWbreak)))

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

PAUSE contains those pauses which are not mandated by other phonetic conditions. The role of this class is greatly reduced by the abandonment of PAUSE/GU equivalence, under which these were the pauses which could have semantic functions, but it still has some limited use.

CANCELPAUSE is a device for indicating that a pause was made in error. The uses envisaged for it had to do with PAUSE/GU equivalence but there may be situations where one wants to cancel a pause nonetheless, so I am leaving it here.

```
TAI0 <- (!(Predicate) (((V1 (juncture)?
 !(Predicate) !(Name) M a (juncture2)?) /
(V1 (juncture)? !(Predicate) !(Name) F i (juncture)?) /
(C1 AI (u)?) / (C1 EI (u)?) / (C1 EO) /
(Z i (juncture)? V1 (juncture2)? ((M a))? (juncture2)?))
(!(Oddvowel) / (!([ ]) &(TAI0)))))
```

Atomic letter words. The deprecated V-initial names for vowels are retained for the moment. V-`ma` and V-`fi` are the old upper- and lowercase vowel names. The new ones are `zi`-V-`ma` and `zi`-V. The name for `y`, `ziy` is supported though phonetically irregular. C1-`ai` and C1-`ei` name upper and lower case consonants. C1-`eo` names Greek letters.

I have added C-aiu and C-eiu letters. Haiu is X, Kaiu is Q, Vaiu is W (change to e for lower case). This supplies the need for names of these non-Loglan letters, which are common in mathematics.

The principal function of these words is as pronouns!

This appears in this odd location for phonetic reasons: the classes A and I need to exclude it. Its proper place in the development is below just before class DA0.

I installed a completely unprincipled fix to save acronyms with the legacy VCV letterals which also allows a large space of multiletter pronouns: a TAI0 followed by a string of VCV letterals is still a TAI0. This creates no problems but is utterly unprincipled!

Fix installed to allow use of VCV letterals in acronyms.

```
NOI <- (N OI !(Oddvowel))

A0 <- (!(Predicate) !((Mono / BrokenMono))
(([AEOUaeou] / (H a)) (juncture)? !(V2) !(Oddvowel)))

A <- (__LWinit !(TAI0) (((N [u])
&((u / (N [o])))))? ((N [o]))? A0 (NOI)?
!((([ ])+ PANOPAUSES PAUSE))
 !((PANOPAUSES [ ]))
 ((PANOPAUSES ((F i) / PAUSE)))? !(Oddvowel))

ANOFI <- (__LWinit !(TAI0) (((N [u])
&((u / (N [o])))))? ((N [o]))? A0 (NOI)?
 (PANOPAUSES)? !(Oddvowel))

A1 <- (A __LWbreak)
```

These three classes define the main series of logical connectives. A0 contains the atomic logical connectives, the words `a e o u` amd the interrogative connective `ha`.

The class A of compound logical connectives is quite complex. One may prefix a compound logical connective with `no`. A `no` or `u` initial connective may further be prefixed with `nu`. The core of an A connective is an instance of A0. It can then be suffixed with `noi` and further suffixed with a PA class tense/location/relation word. If the word contains a PA component it must be closed with **fi** or a full comma pause.

The class A is not closed up with `LWbreak` because it can serve as an initial component of other classes. Other rules in the grammar (such as `LWbreak` above) ensure that one must explicitly pause before any word or word component of class A.

A1 is inhabited by top level A words.

The class A needs to check that it is not the beginning of one of the words `Vma` or `Vfi`, legacy vowel letterals. We do encourage not using these words, using `ziV` and `ziVma` instead.

An A word cannot be followed by a PA word unless the word is closed with GU or an explicit comma pause.

```
ACI <- (ANOFI C i __LWbreak)

AGE <- (ANOFI  G e __LWbreak)
```

These are two other series of logical connectives. The ACI connectives bind more tightly. The AGE connectives bind more loosely. See the grammar section for details. A modification to the grammar was required to ensure that an A connective followed by GE cannot be confused with AGE: the solution is that classes of predicates and arguments linked by A1 are forbidden to start with GE (which is deprecated as a matter of style already): this is a new grammatical rule though, and causes certain examples given (and deprecated) by JCB not to parse at all.

```
CA0 <- ((((N o))? ((C a) / (C e) / (C o) /
 (C u) / (Z e) / (C i H a)) !(Oddvowel)) (NOI)?)

CA1 <- ((((N u) &(((C u) / (N o)))))?
((N o))? CA0
!((([ ])+ PANOPAUSES PAUSE))
 !((PANOPAUSES [ ]))
 ((PANOPAUSES ((F i) / PAUSE)))? !(Oddvowel))

CA1NOFI <- ((((N u) &(((C u) / (N o)))))? ((N o))?
 CA0 (PANOPAUSES)? !(Oddvowel))

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

ZE2 <- (__LWinit (Z e) __LWbreak)
```

An auxiliary series of logical connectives (used to link predicates internally to metaphors and in the construction of some other classes). Their structure is closely analogous to that of the main series; no pauses are needed.

Includes NO prefix and NOI suffix in core CA connective CA0. I should remove the optional NO in CA1.

The rule given here allows more CA words than LIP seems to permit.

ZE2 signals a variant use of ZE to connect arguments. I am contemplating adding words to this class to give a different solution to the problem of representing sets and lists by enumeration.

```
I <- (__LWinit !(TAI0) i
!((([ ])+ PANOPAUSES PAUSE))
!((PANOPAUSES [ ]))
 ((PANOPAUSES ((F i) / PAUSE)))? __LWbreak)

ICA <- (__LWinit !(Predicate) i ((H a) / CA1) __LWbreak)

ICI <- (__LWinit i (CA1NOFI)? C i __LWbreak)

IGE <- (__LWinit i (CA1NOFI)? G e __LWbreak)
```

Sentence and utterance level connectives with different levels of precedence. See the final rules of the grammar for the details.

Provisions for PA components are similar to those in class A.

In all three of A, CA, I and relatives, one cannot follow a word of these classes with a pause free PA component then whitespace. This is purely technical, designed to detect unclosed APA/IPA words in legacy text.

The class I needs to check that it is not the beginning of a legacy vowel letteral `ima` or `ifi`.

```
KA0 <- (((K a) / (K e) / (K o) / (K u) / (K i H a)) !(Oddvowel))

KOU <- (((K OU) / (M OI) / (R AU) / (S OA)) !(Oddvowel))

KOU1 <- (((N u N o) / (N u) / (N o)) KOU)

KA <- (__LWinit &(caprule) (((((N u) &((K u))))? KA0) /
 ((KOU1 / KOU) K i)) (NOI)? __LWbreak)

KI <- (__LWinit (K i) (NOI)? __LWbreak)
```

The forethought connectives. The causal word classes KOU and KOU1, which are more naturally associated with the PA words, are needed for the causal series.

```
BadNIStress <- ((C1 V2 (V2)? stress ((M a))? ((M OA))? NI RA) /
(C1 V2 stress V2 ((M a))? ((M OA))? NI RA))

NI0 <- (!(BadNIStress) (((K UA) / (G IE) / (G IU) / (H IE) /
 (H IU) / (K UE) / (N EA) / (N IO) / (P EA) / (P IO) / (S UU) /
 (S UA) / (T IA) / (Z OA) / (Z OO) / (H o) / (N i) / (N e) /
(T o) / (T e) / (F o) / (F e) / (V o) / (V e) / (P i) / (R e) /
 (R u) / (S e) / (S o) / (H i)) !(Oddvowel)))

SA <- (!(BadNIStress) ((S a) / (S i) / (S u)) (NOI)? !(Oddvowel))

RA <- (!(BadNIStress) (((R a) / (R i) / (R o)) !(Oddvowel)))

NI1 <- ((NI0 ((!(BadNIStress) M a))?
 ((!(BadNIStress) M OA (NI0)*))?
!(Oddvowel)) ((comma2 !((NI RA)) &(NI)))?)

RA1 <- ((RA ((!(BadNIStress) M a))?
((!(BadNIStress) M OA (NI0)*))?
 !(Oddvowel)) ((comma2 !((NI RA)) &(NI)))?)

IE1 <- (__LWinit IE __LWbreak)

NI2 <- ((((SA)? ((NI1)+ / RA1)) / SA)
 (NOI)? ((CA0 (((SA)? ((NI1)+ / RA1)) / SA) (NOI)?))*)

NI <- (__LWinit (IE1)? NI2 ((&((M UE)) Acronym
 (comma / &(end) / &(period)) !((C u))))? ((C u))? !(Oddvowel))

mex <- (__LWinit NI __LWbreak)
```

SA is a class of quantifiers with special functions as prefixes (which can be negated with **-noi**).

RA is a subset of the quantifier words which do double duty as suffixes creating predicates from quantity words.

guarded from being initial in a predicate, as many word component forms are.

The rule NI1 supports notation for powers of ten (though it also allows other odd things). An atomic quantifier word (in intention a numeral but the grammar does not require this) can be followed by an optional `ma`, then an optional `moa` which may have a numeral attached. The `ma` multiplies by 100; `moa` multiplies by 1000; `moa` followed by n multiplies by 1000^n (so `nomoato` is one million). This is new. The shape of `moa` was changed to avert confusion with the pronoun `mo`.

Notice that this change ensures that the use of `ma` to multiply by one hundred and the use of `ma` to capitalize a letter are distinguishable as what occurs just before them will make it clear what is intended. `ma` and `moa` are not members of NI0.

The rule NI2 describes quantifier cores.

IE1 is the interrogative "which", also used in argument constructions.

NI is the class of mathematical expressions. mex is a freestanding word class, NI can be a component of other classes. There is an optional initial `ie` (which?); this is followed by a quantifier core; one may optionally append an acronym followed by an explicit pause as a dimension, and there is a final option of appending `cu`.

As a "word" class NI is unusual in freely permitting spaces and comma marked pauses in the interior of its instances.

It is worth noting that SARA forms are not predicates but NI words. L1 **sara** is a predicate: for us this is **sarara**.

The NI class has been fixed so that it does not eat a following numerical predicate even though it may contain spaces or comma pauses: **ne, tori** does work. The fix also has the effect that numerical predicates do not contain spaces or comma pauses.

BadNIStress is a device for (almost) forcing penultimate stress on numerical predicates. There is freedom in where to place the stress if the last syllable(s) are **ma**, **moa** – it could be on these syllables or the last one before them.

```
CI <- (__LWinit (C i) __LWbreak)
```

a sort of verbal hyphen used in several constructions.

```
Acronym <- (([ ])* &(caprule) ((M UE) /
 TAI0 / ([Zz] V2 !(V2))) ((NI1 / TAI0 / ([Zz] V2 (!(V2) / ([Zz] &(V2))))))+)
```

There is now a single acronym class. An acronym is a sequence of letter names (possibly abbreviated in the case of vowels, and number names, beginning either with **mue** or a letter (possibly abbreviated) and having more than one component (the dummy **mue** allows the formation of one letter acronyms and also of numeral initial acronyms without confusion with numerals or letterals. Acronyms are used to form dimensioned numbers (dimensions must begin with **mue**) and to form acronymic names (no longer acronymic predicates).

```
TAI <- (__LWinit (TAI0 / ((G AO) !(badspaces)
 !(V2) ([ ])* (Name / Predicate / (C1 V2 V2 (!(Oddvowel) /
&(TAI0))) / (C1 V2 (!(Oddvowel) / &(TAI0)))))) __LWbreak)
```

This is the full class of letterals, including John's proposal that allows construction of letters using `gao` followed by an arbitrary word (name, predicate or consonant-initial unit cmapua).

```
DA0 <- (((T AO) / (T IO) / (T UA) /
 (M IO) / (M IU) / (M UO) / (M UU) /
(T OA) / (T OI) / (T OO) / (T OU) /
(T UO) / (T UU) / (S UO) / (H u) /
(B a) / (B e) / (B o) / (B u) / (D a) /
(D e) / (D i) / (D o) / (D u) / (M i) /
(T u) / (M u) / (T i) / (T a) / (M o)) !(Oddvowel))
```

Atomic pronouns of various kinds. They appear in this section because letterals are also pronouns.

```
DA1 <- ((TAI0 / DA0) ((C i !([ ]) NI0))? !(Oddvowel))

DA <- (__LWinit DA1 __LWbreak)
```

Pronouns. These can be suffixed with mathematical expressions using `ci`. Restriction to single digit indices imposed to ensure that one does not need to pause to recognize the end of such a word. Please note that letteral pronouns are **single letters**, possibly with a numerical subscript. There are no multi-letter variables and thus there is no need to pause between letteral pronouns which appear in sequence as arguments in a sentence. It does appear as if this is what JCB assumed in NB3, though it is not what LIP does. It is far more important to treat sequences of pronouns sensibly than to support use of multi-letter variables. This is a change from LIP but not I think from the actual intentions of the founder.

There was danger of a need to pause after an acronym to avoid it absorbing a letteral: it is actually now always required that one pause after an acronym.

Here begins the definition of tense/location/relation operators. The component classes KOU1 and KOU were referenced early because of their role in defining forethought causal connectives.

```
PA0 <- (((G IA) / (G UA) / (P AU) /
 (P IA) / (P UA) / (N IA) / (N UA) /
(B IU) / (F EA) / (F IA) / (F UA) / (V IA) /
 (V II) / (V IU) / (C IU) / (C OI) / (D AU) /
 (D II) / (D UO) / (F OI) / (F UI) / (G AU) /
 (H EA) / (K AU) / (K II) / (K UI) / (L IA) /
 (L UI) / (M IA) / (M OU) / (N UI) / (P EU) /
 (R OI) / (R UI) / (S EA) / (S IO) / (T IE) /
(V a) / (V i) / (V u) / (P a) / (N a) / (F a) /
(V a) / KOU) !(Oddvowel))

PANOPAUSES <- (((!(PA0) NI))?
((KOU1 / PA0))+
(((comma2? CA0 comma2?)  ((KOU1 / PA0))+))*
 (ZI)? !(Oddvowel))
```

```
PA3 <- (__LWinit PANOPAUSES __LWbreak (freemod)?)

PA <- ((((!(PA0) NI))? ((KOU1 / PA0))+
 (((((comma2)? CA0 (comma2)?) / (comma2 !(mod1a)))
((KOU1 / PA0))+))* (ZI)? !(Oddvowel))
```

I am skeptical of **pau**, ago, but implemented it because it appears in the Visit.

A complex PA word begins with an optional numeral or quantifier, followed by a core PA0 or KOU1 word, which may be linked by CA0 connectives to further possibly negated PA0/KOU1 words, then may optionally close with a ZI qualifier or be followed by another PA word. Pauses are permitted in medial positions (not immediately before ZI).

The version without pauses is used in APA and IPA words.

The word version PA3 without pauses is used in modifiers.

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

A PA word as a preposition, so to speak.

```
GA <- (__LWinit (G a) __LWbreak)

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

A PA word as a predicate marker (a tense, so to speak). GA is the purely grammatical predicate marker with no semantic freight.

```
ZI <- ((Z i) / (Z a) / (Z u))
```

qualifying suffixes for tense and location words. They have other uses as affixes (in the genuine sense).

guarded against being initial in a predicate.

```
(* old version:  LE <- (__LWinit ((L EA) / (L EU) / (L OE) / (L EE) /
 (L AA) / (L e) / (L o) / ((L a) !(badspaces)))
 ((DA1 / TAI0))? !((([ ])+ PA)) (PA)? __LWbreak) *)

LE<-(__LWinit ((L EA)/(L EU)/(L OE)/(L EE)/(L AA)/
(L e)/(L o)/((L a) !badspaces)) __LWbreak)
```

The commonest class of argument constructors. These can be adorned with an optional pronoun followed by an optional tense/location/relation operator. 2/12/2016: the inflections are no longer needed, because class `descriptn` was updated to accept the implicit grammatical construction for more general arguments.

```
LEFORPO <- (__LWinit ((L e) / (L o) / NI2) __LWbreak)
```

These are the possible LE components of a LEPO phrase (an abstraction descriptor). The LE and PO components are separate words, and can be separated by commas or not as one pleases. Note that this includes some NI words.

```
LIO <- (__LWinit (L IO) __LWbreak)
```

The numeral article.

```
LAU <- (__LWinit (L AU) __LWbreak)

LOU <- (__LWinit (L OU) __LWbreak)
```

```
LUA <- (__LWinit (L UA) __LWbreak)

LUO <- (__LWinit (L UO) __LWbreak)

ZEIA <- (__LWinit Z EI a __LWbreak)

ZEIO <- (__LWinit Z EI o __LWbreak)
```

Left and right boundary markers (and now medial markers) for explicit sets and lists above.

Below, the beginning of quotation constructions.

```
LI1 <- (L i)

LU1 <- (L u)

Quotemod <- (((Z a) / (Z i)) !(Oddvowel))
```

affixes (in the true sense) which can qualify the sense of a quotation (as text or sound). At least one member of *la Keugru* strongly deprecates this device.

```
LI <- ((__LWinit LI1 !(V2) (Quotemod)? ((([,])? ([ ])+))?
 utterance0 (', ')? __LWinit LU1 __LWbreak) /
(__LWinit LI1 !(V2) (Quotemod)? comma name
(comma)? __LWinit LU1 __LWbreak))
```

Above is the construction for quoting a Loglan utterance, `li` utterance `lu`. The convention of enclosing the quoted text in explicit pauses is allowed but not required. It is also possible to quote a name with this construction (a comma before the name is mandatory in this case).

```
stringnospaces <- (([,])? (([ ])+
 ((!([ ,]) !(period) .))+) ((([,])? ([ ])+
 &(letter)) / &(period) / &(end)))

stringnospacesclosed <-
 (([,])? (([ ])+ ((!([ ,]) !(period) .))+)
 (([,] ([ ])+) / &(period) / &(end)))

stringnospacesclosedblock <- ((stringnospaces
 ((!(([y] stringnospacesclosed)) [y] stringnospaces))*
 ([y] stringnospacesclosed)) / stringnospacesclosed)
```

a block of text beginning with whitespace or an explicit pause and ending with whitespace, an explicit pause, or before terminal punctuation or end of text, and containing no commas or terminal punctuation otherwise. It may contain other symbols or non-Loglan letters. Initial and final whitespace must be expressed phonetically as a pause.

The further classes build comma-closed alien text or sequences of alien text blocks separated by **y** with the final one comma closed.

```
LAO1 <- (L AO)

LAO <- (([ ])* (LAO1 stringnospaces (([y] stringnospaces))*))
```

JCB advertises this as the Linnaean biological name construction, but it is actually a perfectly general device for constructing foreign names in general (Steve Rice noted this in L3). A name is `lao` followed by one or more stringnospaces blocks. If there is more than one block, they must be separated by `y` bounded with pauses. JCB does not require writing the `y` but for the moment I do. One must pause at the end of a LAO construction though one need not write a comma.

I propose that if one wants to construct names by look rather than sound, one should use `lao`.

```
LIE1 <- (L IE)

CII1 <- ((C II) / [y])

LIE <- (([ ])* LIE1 ((!([ ]) NI0))? (Quotemod)?
stringnospaces ((CII1 ((!([ ]) NI0))? stringnospaces))*)
```

This is my strong quotation construction, a new proposal, quite different from and intended to completely replace the L1 strong quotation proposal. The format is very similar to LAO (accidentally, I was not aware of the latest LAO specification). The format is `lie` (quoted text), with quoted text broken by `cii` where breaks occur. Quoted text may not contain commas (commas like whitespace to be replaced by `cii` (`y` also permitted)) or terminal punctuation. The initial `lie` may be qualified with Quotemod and/or with a numeral to indicate nested quotation. `cii` may also be numeral qualified to indicate level of nesting of quotation. Essay promised with all details.

```
\begin{verbatim}

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

LIU0 <- ((L IU) / (N IU))

LIU1 <- (__LWinit ((LIU0 !(badspaces) !(V2)
(Quotemod)? ((([,])? ([ ])+))? (Name / (Predicate (comma)?) / (CCV (comma)?) /
 (LW (([,] ([ ])+ !([,])) / &(period) / &(end) / &((([ ])* Predicate)))))) /
(L II (Quotemod)? TAI __LWbreak)))
```

This is the single word quotation construction. Oddly, this is the only place where the rule LW implementing the NB3 definition of compound cmapua is used, and so it appears here. A single name, predicate, or compound cmapua word may be quoted. `liu` (compound cmapua) must be comma terminated except when followed by a predicate or terminal punctuation. LIP

appears to quote only actual compound cmapua (which is the basis of my claim that LIP makes no use of the compound cmapua phonetic algorithm anywhere). Mine will quote any phonetically possible cmapua. One can use `niu` to indicate that the quoted word is not actually a Loglan word.

CCV djifoa may be quoted with **liu**.

`lii` (name of a letter) gives an actual name for a letter (letterals being pronouns when unadorned).

It is wise to guard **liu** quotations with a comma whether this is formally required or not. Note that **liu** is a name marker, so text without explicit pauses leading up to a name word may lead to parse problems.

```
SUE <- (__LWinit ((S UE) / (S AO)) stringnospaces)
```

This handles two quite different functions with the same grammar (not quotation constructions but similar in handling non Loglan text). `sao` (foreign word) constructs a predicate synonymous with the foreign word. `sue` (transcribed sound) constructs an onomatopoeic predicate meaning to make the given sound.

```
CUI <- (__LWinit (C UI) __LWbreak)
```

left marker for metaphor grouping with connectives

```
GA2 <- (__LWinit (G a) __LWbreak)
```

The specific use of **ga** to mark the first argument in a gasent

```
GE <- (__LWinit (G e) __LWbreak)
```

general purpose left marker, several uses

```
GEU <- (__LWinit ((C UE) / (G EU)) __LWbreak)
```

The right marker for metaphor grouping. The L1 form is still supported.

```
GI <- (__LWinit ((G i) / (G OI)) __LWbreak)
```

Marker at the end of fronted argument and modifier lists (used for building object-first sentences; also useful for putting something before the predicate in an imperative).

```
GO <- (__LWinit (G o) __LWbreak)
```

Marker used for putting modifiers in metaphors after the modified.

```
GIO <- (__LWinit (G IO) __LWbreak)
```

Tentatively proposed new particle: if the terms before the predicate in the last case of class `statement` contain more than one argument, the particle **gio** must appear between two terms in such a way that exactly one argument appears before it.

```
GU <- (__LWinit (G u) __LWbreak)

GUI <- (__LWinit (G UI) __LWbreak)

GUO <- (__LWinit (G UO) __LWbreak)

GUOA <- (__LWinit (G UO (Z)? a) __LWbreak)

GUOE <- (__LWinit (G UO e) __LWbreak)
```

```
GUOI <- (__LWinit (G UO (Z)? i) __LWbreak)

GUOO <- (__LWinit (G UO o) __LWbreak)

GUOU <- (__LWinit (G UO (Z)? u) __LWbreak)

GUU <- (__LWinit (G UU) __LWbreak)

GUE <- (__LWinit (G UE) __LWbreak)
```

The word **gu** is the general purpose right closure. The other words close specific constructions. **gui** closes subordinate clauses. **guo** closes abstraction predicates and descriptions. **guu** closes termsets (lists of arguments and modifiers after predicates). **gue** closes the tightly bound argument lists built with **je** and **jue**.

```
JE <- (__LWinit (J e) __LWbreak)

JUE <- (__LWinit (J UE) __LWbreak)
```

The word **je** is used to tightly attach an argument or modifier to a predicate; **jue** is used to attach second and further arguments and modifiers in a list similarly tightly.

```
JI <- (__LWinit ((J IE) / (J AE) /
 (P e) / (J i) / (J a) / (N u J i)) __LWbreak)

JIO <- (__LWinit ((J IO) / (J AO)) __LWbreak)
```

Markers for subordinate clauses.

```
DIO <- (__LWinit ((B EU) / (C AU) / (D IO) / (F OA) / (K AO) /
```

```
 (J UI) / (N EU) / (P OU) / (G OA) / (S AU) / (V EU) / (Z UA) /
 (Z UE) / (Z UI) / (Z UO) / (Z UU) / (L AE) / (L UE)) __LWbreak)

DIO2 <- (__LWinit ((B EU) / (C AU) / (D IO) / (F OA) / (K AO) /
 (J UI) / (N EU) / (P OU) / (G OA) / (S AU) / (V EU) / (Z UA) /
(Z UE) / (Z UI) / (Z UO) / (Z UU)) __LWbreak)
```

Case tags, and the words **lae** and **lue** of indirection which have similar grammar. Class DIO2 is not used yet, but I do have ideas about situations where identifying true case tags as a class would be necessary.

```
ME <- (__LWinit ((M EA) / (M e)) __LWbreak)
```

me converts arguments to predicates. I believe **mea** is the result of a misunderstanding by JCB, and **me** properly understood does its work as well.

```
NUO <- (((N UO) / (F UO) / (J UO) / (N u) / (F u) / (J u)) !(Oddvowel))

NU <- (__LWinit ((NUO !((([ ])+ (NIO / RA)))
 ((NIO / RA))? (freemod)?))+ __LWbreak)
```

Conversion operators, identifying arguments or changing the order of arguments in predicates.

```
PO1 <- (__LWinit ((P o) / (P u) / (Z o)) !(Oddvowel))

PO1A <- (__LWinit ((P OI a) / (P UI a) / (Z OI a) / (P o Z a) /
(P u Z a) / (Z o Z a)) !(Oddvowel))

PO1E <- (__LWinit ((P OI e) / (P UI e) / (Z OI e)) !(Oddvowel))

PO1I <- (__LWinit ((P OI i) / (P UI i) / (Z OI i) /
 (P o Z i) / (P u Z i) / (Z o Z i)) !(Oddvowel))

PO1O <- (__LWinit ((P OI o) / (P UI o) / (Z OI o)) !(Oddvowel))

PO1U <- (__LWinit ((P OI u) / (P UI u) / (Z OI u) /
 (P o Z u) / (P u Z u) / (Z o Z u)) !(Oddvowel))

POSHORT1 <- (__LWinit ((P OI) / (P UI) / (Z OI)) !(Oddvowel))

PO <- (__LWinit PO1 __LWbreak)

POA <- (__LWinit PO1A __LWbreak)

POE <- (__LWinit PO1E __LWbreak)

POI <- (__LWinit PO1E __LWbreak)

POO <- (__LWinit PO1O __LWbreak)

POU <- (__LWinit PO1U __LWbreak)

POSHORT <- (__LWinit POSHORT1 __LWbreak)
```

short-scope and long-scope abstraction forming cmapua.

```
DIE <- (__LWinit ((D IE) / (F IE) / (K AE) / (N UE) / (R IE)) __LWbreak)
```

Indicators of attitude toward the person addressed (social register).

```
HOI <- (__LWinit ((H OI) / (L OI) /
 (L OA) / (S IA) / (S IE) / (S IU)) __LWbreak)
```

The vocative marker and other words which can open a vocative.

```
JO <- (__LWinit ((NIO / RA))? (J o) __LWbreak)
```

The right marker for the scare quote construction. The numerical suffix indicates its scope.

```
KIE <- (__LWinit (K IE) __LWbreak)

KIU <- (__LWinit (K IU) __LWbreak)
```

the verbal parentheses used to insert utterances as parenthetical free modifiers.

```
SOI <- (__LWinit (S OI) __LWbreak)
```

The constructor for spoken emoticons.

```
UIO <- ((UA / UE / UI / UO /
 UU / OA / OE / OI / OU / OO /
IA / II / IO / IU / EA / EE / EI /
 EO / EU / AA / AE / AI / AO /
 AU / (B EA) / (B UO) / (C EA) /
(C IA) / (C OA) / (D OU) / (F AE) /
(F AO) / (F EU) / (G EA) / (K UO) /
 (K UU) / (R EA) / (N AO) / (N IE) /
(P AE) / (P IU) / (S AA) / (S UI) /
 (T AA) / (T OE) / (V OI) / (Z OU) /
(L OI) / (L OA) / (S IA) / (S II) /
 (T OE) / (S IU) / (C AO) / (C EU) /
(S IE) / (S EU)) !(Oddvowel))

NOUI<-((__LWinit N [o] [ ]* UIO __LWbreak)/
(__LWinit UIO NOI __LWbreak))

UI1 <- (__LWinit (UIO / (NI F i)) __LWbreak)
```

Attitudinal constructions. The atomic cmapua of class UI can be negated, either with a **no** prefix or a **noi** suffix (the latter I believe is a small proposal). In additional the numerical discursives with **fi** are attitudinals.

Note that certain UI words are also HOI words.

```
HUE <- (__LWinit (H UE) __LWbreak)
```

The inverse vocative marker.

```
NO1 <- (__LWinit !(KOU1) !(NOUI) (N o)
 !((__LWinit KOU)) !(([ ])* (JIO / JI))) __LWbreak)
```

This is the class of occurrences of **no** as a separate word, not absorbed by a following attitudinal, **kou** word, or subordinate clause marker.

```
AcronymicName <- (Acronym (&(end) / ',' / &(period) / &(Name) / &(CI)))

DJAN <- (Name / AcronymicName)
```

The full construction of name words. Acronyms are treated as a separate class of names. `DJAN` contains all name words. `Name` contains the consonant final name words from the phonetics section.

```
BI <- (__LWinit ((N u))? ((B IA) / (B IE) /
 (C IE) / (C IO) / (B IA) / (B [i])) __LWbreak)
```

The special class of predicates of which **bi**, the "identity" predicate, is the best known example. Converses formed with **nu** of these predicates (all of them are binary) are allowed (a proposal).

```
LWPREDA <- (((H e) / (D UA) / (D UI) / (B UA) / (B UI)) !(Oddvowel))
```

predicates realized as cmapua: the question predicate and various "predicate pronouns".

```
PREDA <- (([ ])* &(caprule) (Predicate /
LWPREDA / (!([ ]) NI RA)) !((!((&(nonamemarkers) Name))
 (A / ICI / ICA / IGE / I))) ((',' ([ ])+ &((!((&(nonamemarkers) Name))
 (V1 / A)))))? (freemod)?)
```

Words which are understood as non-identity predicates. These are designed to absorb mandatory pauses which follow them before vowel-initial words, and to recognize when such mandatory pauses have been omitted.

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

guoa <- ((PAUSE)? (GUOA / GU) (freemod)?)

guoe <- ((PAUSE)? (GUOE / GU) (freemod)?)

guoi <- ((PAUSE)? (GUOI / GU) (freemod)?)

guoo <- ((PAUSE)? (GUOO / GU) (freemod)?)

guou <- ((PAUSE)? (GUOU / GU) (freemod)?)

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

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

guu <- ((PAUSE)? (GUU / GU) (freemod)?)
```

These classes realize the various special right closure markers. Each one can be expressed in its literal form, as **gu** or as a pause. Any pauses next to it are absorbed into it.

```
geu <- GEU
```

right closure in metaphors

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

gap2 <- gap

guu1 <- gap
```

The grammar implementation of the general **gu** right closure: **gu** (absorbing pauses). pause/GU equivalence has been abandoned.

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

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

links <- ((links1 / (KA (freemod)? links (freemod)?
KI (freemod)? links1)) (((freemod)? A1 (freemod)? links1))*)

jelink <- (JE (freemod)? term)

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

linkargs <- ((linkargs1 / (KA (freemod)? linkargs (freemod)?
KI (freemod)? linkargs1)) (((freemod)? A1 (freemod)? linkargs1))*)
```

Tightly bound lists of arguments and modifiers, which we may call "link sets". The ability to include sentence modifiers in these lists is a proposal. In any such list, the argument directly after the predicate is linked with **je**; second and subsequent items in these lists are linked with **jue**. Such lists (and individual items linked with **je** or **jue**) can be linked with forethought and afterthought logical connectives. They can be right closed using **gue**.

Items in the class linkargs are called "tightly bound argument lists" or "link sets".

```
abstractpred <- ((POA (freemod)? uttAx (guoa)?) /
(POA (freemod)? sentence (guoa)?) /
(POE (freemod)? uttAx (guoe)?) / (POE (freemod)? sentence (guoe)?) /
 (POI (freemod)? uttAx (guoi)?) / (POI (freemod)? sentence (guoi)?) /
 (POO (freemod)? uttAx (guoo)?) / (POO (freemod)? sentence (guoo)?) /
 (POU (freemod)? uttAx (guou)?) / (POU (freemod)? sentence (guou)?) /
 (PO (freemod)? uttAx (guo)?) / (PO (freemod)? sentence (guo)?))

predunit1 <- ((SUE / (NU (freemod)?
GE (freemod)? despredE (((freemod)? geu (comma)?))?) /
(NU (freemod)? PREDA) /
 ((comma)? GE (freemod)? descpred (((freemod)? geu (comma)?))?) /
 abstractpred / (ME (freemod)? argument (gap2)?) / PREDA) (freemod)?)
```

Predicate units which are in a sense "atomic". Some of these are quite complex structures which are in effect parenthesized. Each item in this rule is described in the list in the section on the basic building blocks of predicates in the reference grammar.

The PO sentence and PO uttAx forms have been moved to this category from a much different position in the trial.85 grammar, which allowed such forms a very limited role in the grammar. They have been moved into a separate class `abstractpred` to allow use of many bracketing forms, solving the closure problem (I hope).

Items in the class predunit1 are called "atomic predicate units".

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

A `predunit2` is a possibly multiply negated predunit1. `NO2` captures occurrences of the word **no** not captured by `predunit1` instances.

```
NO2 <- (!(predunit2) NO1)

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

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

A predunit3 is obtained from a predunit2 by attaching JE/JUE linked arguments. A predunit is then either a predunit2 or a predunit2 preceded by a short-scope abstraction operator. These are the sort of predicates which can occur as units in serial names. Notice that these include no metaphorical modifications or logical connections, except possibly internally to parenthesized items in class predunit1.

Items in the class predunit1 are called "predicate units".

```
kekpredunit <- (((NO1 (freemod)?))*
 KA (freemod)? predicate (freemod)? KI (freemod)? predicate)
```

A kekpredunit is a possibly multiply negated forethought logically connected general predicate. Items in this class are called "forethought connected predicates".

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

A despredA is either a predunit or a kekpredunit or a chain of these linked with ci, the most tightly binding form of metaphorical modification.

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

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

despredB and despredC interlock. A despredA is an item of either class. A despredC is just a chain of one or more despredB's. A despredB, if it is

not simply a despredA, begins with **cui**, followed by a despredC (that is, by one or more despredB's), followed by a tt CA connective, followed by a final despredB. The purpose here is to enable leftward extension of the scope of a CA connective to an occurrence of **cui**. The simple juxtapositions here are left-grouped metaphorical modifications.

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

despredE <- (despredD (((freemod)? despredD))*)
```

despredD allows afterthought logical connection of despredB class items with CA logical connectives.

despredE allows left-grouped metaphorical modification of a series of despredD's. Right grouping is achieved by using the ge...geu construction to pack the right group as a single predunit1, the last despredD item in a despredE.

Items in class despredE are called "simple description predicates";.

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

A descpred is either a despredE, or a structure consisting of a despredE followed by go followed by a descpred, having the effect of causing the first component to be metaphorically modified by the second component.

A descpred is the sort of predicate which can occur after **le** in a description.

Items in class descpred are called "description predicates".

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

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

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

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

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

These classes of predicates, which occur as main predicates of sentences rather than in descriptions, are built in exactly analogous ways to the classes of predicates which occur in descriptions, except that freemods (and so pauses) are allowed between successive items in left-grouped chains of metaphorical modifiers (at the top level; the individual items are description predicates and so have restricted internal pauses; but this seems a natural enough pattern of articulation). Notice though that `senpred2`, and `kekpredunit`s will not occur as first items in chains of modifiers (they can occur later because second and later items in these chains are taken from the description predicate classes. The `barepred` class is a very general class of predicates with the possibility of attached arguments, defined below.

Items in class sentpred are called "sentence predicates".

```
mod1a <- (PA3 argument (gap)?)

mod1 <- ((PA3 argument (gap)?) / (PA2 !(barepred) (gap)?))

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

mod <- (mod1 / (((NO1 (freemod)?))* mod1) / kekmod)

modifier <- ((mod / kekmod) ((A1 (freemod)? mod))*)
```

This rule describes the sentence modifiers (tense/location/relative modifiers). The basic building blocks are a `PA3` operator (no internal pauses) followed by an argument, possibly closed with a `gap`, or a `PA2` operator by itself, closed with a gap.

A `kekmod` is a forethought connected modifier structure.

A `mod` is suitable to be a final item in a `kekmod`: it is either a basic unit or a negated basic unit or a kekmod.

A `modifier` is either a `mod` or an afterthought logically connected series of `mod`'s.

```
maybebreak <- (V1 (stress)? ' ' !((([ ])* V1)))

realbreak <- (!(maybebreak) letter (stress)? ((([,])? ' ') / period / &(end)))

consonantbreak <- (C1 (stress)? ((([,])? ' ') /
period / &(end)))

badspaces <- (!(([,] ' ')) ((!((maybebreak / realbreak)) .))*
 maybebreak ((!(realbreak) .))* consonantbreak)

namemarker <- ((([ ])* ((L a) / (H OI) / (L OI) / (L OA) / (S IE) /
 (S IA) / (S IU) / (C i) / (H UE) / (L IU) / (G AO))) !(badspaces))

nonamemarkers <- (([ ])* ((!((namemarker DJAN)) Letter))+ !(Letter))
```

`badspaces` is an evil hack, used to detect failure to have a mandatory pause (comma marked, after a consonant or before a vowel) before the end of the first name after a name marker word which does not actually stand right before a name.

`namemarker` is the small class of words after which a name word can occur without a pause.

`nonamemarkers` is the class of name words not containing a name marker such that the part of the word after the name marker is a well-formed name. NOTE: this class should probably be defined in terms of `Name` rather than in terms of `DJAN`.

```
name <- ((DJAN ((((([ ])* (freemod)? [Cc] I DJAN) /
 ((([ ])* (freemod)? CI !(badspaces) (freemod)? predunit)
 !((&(nonamemarkers) !(AcronymicName) DJAN))) /
 (([ ])* (freemod)? CI (',' ([ ])+) DJAN) /
 (&(nonamemarkers) !(AcronymicName) DJAN)))*) (freemod)?)
```

This is the class of serial names. The first unit in a serial name will be a name word. Other units will be predunits marked with **ci**, name words marked with **ci**, or unmarked name words containing no false name markers and not preceded in the serial name by a predunit.

Pauses after name words in the interior of serial names must occur, but do not have to be marked with commas.

After CI without a pause, a wellformed name will always be read as such. After CI marked with a comma, a name-final predunit will be recognized even without spaces.

```
LA0 <- (([Ll] a) !(badspaces))

LANAME <- (([ ])* LA0
(CANCELPAUSE / (([ ])* &(C1))) name (gap2)?)

LANAME2 <- (([ ])* LA0
((',' ([ ])+) / (([ ])* &(V1))) name (gap2)?)
```

This is the basic name construction with the article **la**. A comma pause is permitted but not required between **la** and the name. But if a comma is present non-name words will be sought first by the parser. Phonetically there is the assumption that if a comma is not written one does not pause.

There is fine tuning for the case of vowel initial names, where there is always a pause, but the CANCELPAUSE class can be used to cause it to be ignored.

```
HOI0 <- ((([Hh] OI) / ([Ll] OI) / ([Ll] OA) /
([Ss] IA) / ([Ss] IE) / ([Ss] IU)) !(badspaces))

voc <- ((([ ])* HOI0 (CANCELPAUSE /
 (([ ])* &(C1))) name (gap2)?) /
 (HOI !(badspaces) (freemod)?
descpred (((((comma)? CI (comma)?) /
(comma &(nonamemarkers) !(AcronymicName)))
 name))? (gap2)?) / (HOI !(badspaces) (freemod)?
```

```
argument (gap2)?) / (([ ])* HOI0 ((',' ([ ])+) /
(([ ])* &(V1))) name (gap2)?) /
 (H OI stringnospacesclosedblock))
```

This is the general construction of a vocative (a species of free modifier). The first case, **hoi** followed by a serial name, is similar to `LANAME`.

If HOI is not followed by an explicit pause, a name will be read aggressively. If it is followed by a pause, it will try other readings first.

Only **hoi** may be followed by a foreign name (alien text): the other words usable as vocative markers may not be followed by alien text.

Finetuning added for vowel initial names.

```
descriptn<-(!LANAME ((LAU wordset1)/(LOU wordset2)/
(LE freemod? ((!mex arg1a freemod?)? (PA2 freemod?)?)?
 mex freemod? descpred)/
(LE freemod? ((!mex arg1a freemod?)? (PA2 freemod?)?)?
 mex freemod? arg1a)/(GE freemod? mex freemod? descpred)/
(LE freemod? ((!mex arg1a freemod?)? (PA2 freemod?)?)? descpred)))
```

This is a basic class of descriptive arguments. Modified 2/21/2016 to consderably generalize the possessive construction.

Items of this class are called "basic descriptions".

```
abstractn <- ((LEFORPO (freemod)?
POA (freemod)? uttAx (guoa)?) /
(LEFORPO (freemod)? POA (freemod)? sentence (guoa)?) /
 (LEFORPO (freemod)? POE (freemod)? uttAx (guoe)?) /
 (LEFORPO (freemod)? POE (freemod)? sentence (guoe)?) /
 (LEFORPO (freemod)? POI (freemod)? uttAx (guoi)?) /
(LEFORPO (freemod)? POI (freemod)? sentence (guoi)?) /
(LEFORPO (freemod)? POO (freemod)? uttAx (guoo)?) /
(LEFORPO (freemod)? POO (freemod)? sentence (guoo)?) /
 (LEFORPO (freemod)? POU (freemod)? uttAx (guou)?) /
(LEFORPO (freemod)? POU (freemod)? sentence (guou)?) /
 (LEFORPO (freemod)? PO (freemod)? uttAx (guo)?) /
(LEFORPO (freemod)? PO (freemod)? sentence (guo)?))
```

This is the class of abstract descriptions (originally two cases of the following class but pulled out on its own because more bracketing forms have been added to solve the GUO GUO problem).

```
arg1 <- (abstractn / (LIO (freemod)? descpred (gap2)?) /
 (LIO (freemod)? term (gap2)?) / (LIO (freemod)? mex (gap2)?) /
 (LIO stringnospaces) / LAO / LANAME /
 (descriptn (freemod)? (((((comma)? CI (comma)?) /
(comma &(nonamemarkers) !(AcronymicName))) name))? (gap2)?) /
 LANAME2 / LIU1 / LIE / LI)
```

This is a class of basic arguments built with articles.

Will read a name aggressively after LA without a comma; will read a name as last resort after LA with a comma.

Items of this class are called "descriptions".

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

This is the full class of "atomic arguments", adding in the pronouns.

Below we construct subordinate clauses.

```
argmod1 <- (((__LWinit (N o) ([ ])*))? ((JI (freemod)? predicate (gui)?) /
 (JIO (freemod)? sentence (gui)?) / (JIO (freemod)? uttAx (gui)?) /
 (JI (freemod)? modifier (gui)?) / (JI (freemod)? argument (gui)?)))

argmod <- (argmod1 ((A1 (freemod)? argmod1))*)
```

The basic kinds of argument modifier (subordinate clauses) are given (discussed above in the reference grammar). These can be linked with afterthought logical connectives.

```
arg2 <- (arg1a ((argmod (gap2)?))*)

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

indef1 <- (mex (freemod)? descpred)

indef2 <- (indef1 (gap2)? ((argmod (gap2)?))*)

indefinite <- indef2

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

arg5 <- (arg4 / (KA (freemod)? argument (freemod)? KI (freemod)? argx))

arg6 <- (arg5 / (DIO (freemod)? arg6) / (IE1 (freemod)? arg6))

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

L("argxx1 <- &((NO1 freemod?)* DIO) argx")

L("arg7 <- (argx ((ACI (freemod)? argx))?)")

L("arg8 <- (!(GE) (arg7 ((A1 (freemod)? arg7))*))")

L("argument <- (((arg8 AGE (freemod)? argument) / arg8)
 ((GUU (freemod)? argmod (gap)?))*)")

L("argxx2 <- (argxx1 ((ACI (freemod)? argxx1))?)")

L("argxx3 <- (!(GE) (argxx2 ((A1 (freemod)? argxx2))*))")

L("argxx <- (((argxx3 AGE (freemod)? argxx3) / argxx3)
 ((GUU (freemod)? argmod (gap)?))*)")
```

The construction of arguments from basic arguments and argument modifiers is narrated above in the reference grammar. It is important to notice

that top level arguments cannot begin with **ge**, to avoid ambiguity with the `AGE` connectives.

Note the ability to use **guu** to attach a subordinate clause to a complex argument.

`argxx` is a technical device for the second test parser: this is a class of argument guaranteed to have all components explicitly case tagged.

```
term <- (argument / modifier)

terms <- (term (((freemod)? term))*)

modifiers <- (modifier (((freemod)? modifier))*)

modifiersx<-((modifier/argxx)
(freemod? (modifier/argxx))*)
```

`term` is the class of arguments and modifiers. `terms` contains series of terms. `modifiers` is such a sequence with no arguments. `modifiersx` is a sequence of modifiers and explicitly case tagged arguments.

```
word <- (arg1a / indef2)

words1 <- (word ((ZEIA word))*)

words2 <- (word ((ZEIO word))*)

wordset1 <- ((words1)? LUA)

wordset2 <- ((words2)? LUO)
```

The innards of my proposal to implement unordered and ordered lists. Unordered lists are of course finite sets given by enumerating their elements. The two constructions are parallel but separated.

```
termset1 <- ((terms (guu)?) /
 (KA (freemod)? termset2 (freemod)? KI (freemod)? termset1))

termset2 <- (termset1 ((A1 (freemod)? termset1))*)

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

These are the termsets which appear as internals of a predicate. A `terms` followed by an optional closure with `guu` or a forethought connected structure of a `termset2` and a `termset1` is a `termset1`; an afterthought connected sequence of `termset1`'s is a `termset2`.

A `termset` is a `termset2`, or a `terms` followed by **go** followed by a `barepred` predicate metaphorically modifying the predicate to which the termset is attached (an odd but potentially useful construction), or a solitary `guu` (the last is much more useful than one might suppose).

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

The purpose here is to allow head modification with kekpreds in sentence predicates, which was not allowed in earlier versions of the grammar. This calls into question the use of the distinction between sentence and description predicates; I need to think about this.

Items of this class are called "forethought-initial predicates".

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

This is the general class of untensed predicates with attached termset.

Items of this class are called "bare predicates".

```
markpred <- (PA1 barepred)
```

This is the class of tensed predicates. They may also be called marked predicates.

```
backpred1 <- (((NO2 (freemod)?))* (barepred / markpred))

backpred <- (((backpred1 ((ACI (freemod)? backpred1))+ (freemod)? (termset)?)
 ((((ACI (freemod)? backpred))+ (freemod)? (termset)?))?) / backpred1)

predicate2 <- (!(GE) (((backpred ((A1 !(GE) (freemod)? backpred))+
(freemod)? (termset)?) ((((A1 (freemod)? predicate2))+ (freemod)?
 (termset)?))?) / backpred))

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

This series of classes implements logically connected predicates with shared termsets using connectives of three series, the ACI binding most tightly, left grouped, the A binding less tightly, left grouped, and the AGE binding most loosely, right grouped. The ACI connectives are treated in the same way as the A connectives here; they are very strange in the trial.85 grammar. The systematic distinction between marked and unmarked forms is abandoned (it is provably not doing anything in the trial.85 grammar, either). The top-level sentence predicates cannot begin with **ge** due to ambiguity with the AGE connectives (something similar happens with arguments).

Shared termsets may be affixed to ACI linked and A linked predicates; the termset of the final item in the logically linked predicate may need to be closed off with GUU explicitly (or may need to be supplied as a solitary GUU); the precise ways in which this can be done are somewhat limited by the allergy of PEGs to left recursion, but I believe that in practice this grammar will be found fully capable.

In theory, this construction is rather different from the analogous construction in trial.85, but I believe that it will be found in practice to support much the same space of speakable complex predicates.

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

predicate <- (predicate1 / identpred)
```

This completes the definition of general predicates. These are either logically complex predicates of the last type discussed or the identity predicates (possibly negated). Note that the identity predicates *can* enter into logical composites via `kekpredunit`, for example.

```
oneargument <- (((modifiers (freemod)?))?
argument ((modifiers (freemod)?))?)

oneargument<-((modifiers freemod?)?
 ((argxx oneargument)/
(argument (modifiersx freemod?)?)))

gasent1 <- (((NO1 (freemod)?))* (PA1 (freemod)?
 barepred ((GA2 (freemod)? oneargument))?))

gasent2 <- (((NO1 (freemod)?))* (PA1 (freemod)?
sentpred (modifiers)? (GA2 (freemod)? terms)))

gasent <- (gasent2 / gasent1)
```

We differ from trial.85 `gasent` in having the final (ga terms) optional – so that a solitary tensed predicate is an observative (with suppressed indefinite first argument) rather than an imperative.

Further, we enforce the rule that the (ga terms) component contains exactly one argument or all the arguments in the sentence. This takes a little cunning, but is achievable.

Items of class gasent are called "subject-final sentences", though they may lack subjects entirely.

The second version of oneargument is a technical device for another test parser: the idea is to allow more than one argument as long as no more than

one of them fails to be explicitly case tagged. In the second test parser, this rule is renamed `subject`.

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

statement<-(gasent/(modifiers freemod? gasent)/
(oneargument (GIO freemod? terms)? predicate))

statement<-(gasent/(modifiers freemod? gasent)/
(oneargument freemod? predicate))
```

The statement class is modified to include all gasents as modified (thus including what were formerly tensed imperatives). In the former (terms gasent) construction, the terms are constrained to all be modifiers, which was always the intention.

The second version is used in our latest test parser. The intention is that an SOV(O) sentence will be explicitly marked as such by an occurrence of **gio** between its first and second arguments to the left of the predicate. A statement falling under the last alternative and not containing **gio** will have exactly one argument appearing before the predicate.

The second test parser uses the third version of the statement class with the new version of `oneargument`.

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

We remark here on the extreme freedom of the final item in these forethought linked sentences. Look down at `uttA1` to see this.

Items of this class are called "forethought connected sentences".

```
sen1 <- ((modifiers (freemod)? !(gasent) predicate) /
```

```
statement / predicate / keksent)
```

The first and third cases here are the imperative sentences. The first case would parse as a statement, but is captured first by this class (modifiers followed by an untensed predicate should be imperative, and indeed this was always the intention; this parser captures this intent).

Items of this class are called "logical unit sentences".

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

afterthought connected `sen1`'s.

Items of this class are called simply "sentences": when they are not logical unit sentences they may be called "logically connected sentences".

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

Shared fronted final arguments to be attached to a sentence in the next construction.

```
uttAx <- (headterms (freemod)? sentence (gap2)?)
```

A `sentence` with shared final arguments in fronted position. It is important to notice that fronted terms with **gi** will distribute over sentences linked with ICA connectives.

Items of this class are called "sentences with fronted arguments".

```
HUE0 <- ([Hh] UE)

freemod <- ((NOUI / (SOI (freemod)? descpred (gap2)?) /
 DIE / (NO1 DIE) / (KIE (comma)? utterance0 (comma)? KIU) /
 (([ ])* HUE0 (CANCELPAUSE / (([ ])* &(C1))) name (gap2)?) /
```

```
 (HUE !(badspaces) (freemod)? descpred
 (((((comma)? CI (comma)?) / (comma &(nonamemarkers)
!(AcronymicName))) name))? (gap2)?) /
 (HUE !(badspaces) (freemod)? statement (gap2)?) /
(HUE !(badspaces) (freemod)? termset1) /
(([ ])* HUE0 ((',' ([ ])+) / (([ ])* &(V1))) name (gap2)?) /
 (HUE stringnospacesclosedblock) / voc /
 CANCELPAUSE / PAUSE / JO / UI1 /
 (([ ])* '...' ((([ ])* &(letter)))?) /
 (([ ])* '--' ((([ ])* &(letter)))?)) (freemod)?)
```

The free modifiers. These are discussed in the reference grammar. The important change here is the elimination of unmarked names as vocatives. The class `freemod` contains all free modifiers other than pauses; it is used to allow pause/GU equivalence, and if it is set equal to freemod, pause/GU equivalence is disabled. The class `freemod` preserves a very few cases where things must remain pauseless. freemod and freemod are now the same as freemod.

Ellipses and dashes (double hyphens) are freemods, enhancing our punctuation.

There is no preprocessing in this grammar, so `freemod` appears ubiquitously in other rules.

will read a name aggressively after HUE without a comma pause; with explicit comma will try other readings first. Fine tuning added for vowel initial names.

```
uttA <- ((A1 / IE1 / mex) (freemod)?)

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

Note the various fragments of utterances allowed here as answers to questions, along with full classes of sentences.

Items of the class uttA are called "answer fragments".

Items of the class uttA1 are called "sentence fragments".

```
neghead <- (NO1 (gap / PAUSE))

uttC <- ((neghead uttC) / uttA1)

uttD<-((sentence period? !ICI !ICA)/
(uttC (ICI freemod? uttD)*))

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

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

```
utterance0 <- (!(GE) ((!(PAUSE) freemod (period)? utterance0) /
(!(PAUSE) freemod (period)?) / (uttE IGE utterance0) /
uttF / (I (freemod)? (uttF)?) / (I (freemod)? (period)?) /
 (ICA (freemod)? uttF)) ((&(I) utterance0))?)

utterance <- (!(GE) ((!(PAUSE) freemod (period)? utterance) /
 (!(PAUSE) freemod (period)? ((&(I) utterance))? end) /
 (uttE IGE utterance) /
 (I (freemod)? (period)? ((&(I) utterance))? end) /
 (uttF ((&(I) utterance))? end) /
(I (freemod)? uttF ((&(I) utterance))? end) /
 (ICA (freemod)? uttF ((&(I) utterance))? end)))
```

The top-level construction of utterances is discussed above in the reference grammar. Recall that the class `end` signifies end of text or the beginning of a Loglan utterance in another voice marked by `#`. There is no analogous ability of the class utterance0 to be chained in this way: `#` is not a quotable or parenthesizable piece of Loglan punctuation.