MacGram Notebook No. 1

THE MACHINE GRAMMAR & CORPUS OF LOGLAN

Incorporating the Trial.19 Grammar
Found 18 March 1982

The Notebook contains four separately paginated Sections which may be rearranged in any order:-

2261 Soledad Rancho Road
San Diego, California 92109
U.S.A

FOREWORD

The grammar described in these pages was developed by Scott Layson and myself in San Diego from mid-January to mid-March of this year. It was based on Jeff Prothero's August 1981 grammar, which was in turn based on Sheldon Linker's June 1980 grammar, which was in turn based on his own February 1978 grammar, which was the first grammar of Loglan to be successfully "yacced"; see Glossary.

Linker's 1978 grammar was in turn based on the 1967 Formal Grammar, originally part of Loglan 2, that had been published in The Loglanist in 1977. The 1967 grammar was in turn based on the 1962-63 sequence of grammars, developed on the University of Florida computers, by which the language itself was built. So MacGram already has a long history. But it is this March 1982 grammar which, for the first time demonstrates the machine-readability of the "whole langauage", which means that part of it that we have so far managed to capture in this Corpus.

The language, its corpus, and its grammar are all likely to grow fairly rapidly from this decisive point. Please communicate to me directly any errors, deficiencies, or possible improvements that you may find.

JCBrown

James Cooke Brown
San Diego
16 June 1982

THE MACHINE GRAMMAR OF LOGLAN

GRAMMAR.MAR Loglan grammar as of March 1982 Trial.19P
18 Mar 82

Created from JSP's Aug 81 grammar by SWL & JCB.

INTRODUCTION

This is the annotated (P for 'Publication') version of the Trial.19 machine grammar of Loglan produced in San Diego last March. In this annotated form, of course, the grammar will not yacc. The "actions" and all the special punctuation marks required by Yacc have been removed for easier reading.

The notes are intended to give rough insights into why the rules are written the way they are, and why they work the way they do. Keener insight can, of course, only be obtained by close study of the way the Parser and Preparser actually behave under the control of these rules and of the Preparser algorithms... interactively with LIP, for example, or by studying the Corpus. But as a first step toward understanding, anyway, I offer the loglanists this annotated time-slice of the still-moving MacGram.

Pages 3 through 7 of this listing give the 63 Lexemes (word-classes) of the current grammar in the alphabetical order of their names: the upper-case expressions A, BA, CA, etc. The words used as lexeme names are nearly always the simplest members of their lexemes. They are also semantically typical, so that replacing the actual words of an utterance with the names of the lexemes to which they belong will nearly always produce an intelligible pseudo-utterance with, of course, the same grammatical structure. The one non-typical lexeme name is PREDA, which names the lexeme to which all predicate words belong; but 'preda' is of course a nonce word that means nothing. This semantically empty label probably catches the vast and varied world of predicate meanings better than any concrete predicate would do.

The word or words in lower-case after the colon in each entry show some or all of the allolexes of that lexeme. Some lexemes, like CI, CUI, GA, GE, etc., are monolexic. These are nearly always one of Loglan's "spoken punctuation marks". Others, like JIO, LAE and PO, have very few members. In these cases the list of allolexes is exhaustive. Some lexemes, like DJAN, NI, PA, PREDA and UI, are unlimited. Each has in principle an infinite number of allolexes, and in in fact a great number of each are found in the current dictionaries. It is probably a distinctive characteristic of Loglan among speakable languages that the number of its unlimited lexemes is very small, while the number of its small, finite lexemes is proportionally very large.

The Grammar itself begins on Page 8. It currently consists of 159 grammar rules defining 62 Gramemes (or "nonterminals"). The grameme names are in lower-case except for an occasional capital-letter or numerical suffix which shows the grameme so-marked to be part of some developmental sequence. This printing convention distinguishes the grameme names from the names of the lexemes (or "terminals"), all of which are in upper-case.

The left-half of every grammar rule is a single grameme. Rules with the same left-half are grouped together in the listing and their common left-half is given only once. The right-half of every rule is a string of one or more gramemes and/or lexemes, up to 5 in the longest case. The sign '=>' may be read 'becomes' or 'may be replaced by'. The grameme names are completely arbitrary and do not, apparently, appear at all in human consciousness. They are evidently quite unnecessary for the learning of a grammar. Still, to facilitate the study of the formal properties of this still-trial grammar, every effort has been made to give the 62 gramemes useful names, in particular, names which reflect the sequences in which the rules are developed. This has not always been easy; some names remain obscure. Suggestions for better grameme names are welcome.

A generator would take these grammar rules and, starting with the Initial Grameme ("utterance", the last rule in the grammar), it would expand it by successive applications of such rules as suited its purpose, and end up with a string of lexemes, which would then be converted (by something else) into words. A parser would take that utterance and the same grammar rules, and (after getting something else to convert the words back into lexemes) it would search the rules for ones that fit, and, by applying them in the opposite direction, try to reduce the string of lexemes back into the Initial Grameme once again. Obviously, there is no trick in being a generator, or in writing a grammar that will drive a generator. The trick is in writing a set of grammar rules that a parser can use infallibly to retrace the steps of a generator backwards, no matter what the generator ends up saying provided it obeys the rules. This is a grammar that permits just such "infallible parsing" of grammatical utterances.

It is the business of a human grammar to provide a common rule-structure for listener and speaker so that the listener can "clamp together" the fragments of each "burst" of speech he hears with a fair chance of traversing exactly the same route, but in the opposite direction, as that taken by the speaker in "exploding" that utterance inside his head. It is the achievement of this grammar that, with Yacc's formal demonstration of its freedom from syntactical ambiguity, that "fair chance" of the listener using the same "route map" in the disassembly of a human utterance as its speaker used in its assembly, has in principle become certainty. What remains to be discovered is whether the human brain can acquire this particular way of forming grammatical route maps, and so enjoy the formal properties of such a grammar. For if it can, then for the first time in a **human** language, what might be called the "customary disambiguation burden" on the human listener, heretofore both very large and un-put-downable, will have been reduced to zero.

Please refer to the Glossary for the meanings of the numerous technical words and abbreviations used for brevity in the notes, both here and in the Corpus.

JCB, 4 Jun 82

THE LEXICON

Lexeme A
: ha a e o u
(also CPDs anoi, apa, efa, noanoi, etc.)

The eks; notice that the new interrogative ha is simply one of them. efa, apa, etc., are new A+PA CPDs; see PA. efa, for example, means 'and-then'; enusoa = 'and-therefore'.

Lexeme BAD

Used by Yacc to keep on "chewing" even if it finds something wrong; I think Scott disabled this in favor of outright refusal.

Lexeme BUA
: bua bue

The non-designating predicate variables.

Lexeme CA
: ca ce co cu
(also CPDs noca, canoi, nocanoi, etc.)

The sheks. Note that ha can't be used to ask about sheks. Nor about keks, for that matter.

Lexeme CI
: ci

The pred-string hyphen. Monolexic.

Lexeme CUI
: cui

The pred-string left paren. Monolexic.

Lexeme DA
: ba be bo bu da de di do du mi tu mu ti ta tau tiu tua mia mua toi toa

All the variables except letter-variables; see TAI for these.

Lexeme DJAN
: (all C-final words found by lexer)

All name-words.

Lexeme END
: 0 .

A special lexeme used by the Preparser to mark the end of the specimen, which may be composed of multiple utterances.

Lexeme GA
: ga

The optional end-of-description punctuator, formerly the "timeless tense" operator.

Lexeme GE
: ge

The pred-string "group-starter". Monolexic.

Lexeme GI
: gi

The "De-localizer". Prefixed to modifiers that are to be taken as having an utterance-wide significance.

Lexeme GO
: go

The pred-string inverter. Monolexic.

Lexeme GOU

The possibly temporary right-mark of the

: gou

prenex quantifier; it needs a better word.

Lexeme GU
: gu

The general comma. Occurs as an optional element after argsets including null ones; and so, after all predicate expressions.

Lexeme GUE
: gue

The pred-string "group-ender". Another of the optional punctuators like GA and GU.

Lexeme GUU
: guu

The possibly temporary right-mark of shifted arguments. Needs a better word.

Lexeme HU
: hu
(used only by CPD-lexer
to find nahu-CPDs;
otherwise like DA)

The argument interrogative. Not in DA only because the Preparser uses it to form CPDs. The nahu = 'when?' CPDs formed with hu are new to the language.

Lexeme I
: i
(also CPDs ica, icinusoa, etc.)

The utterance continuer. It really stands between "utterances" (in the narrow sense).

Lexeme IE
: ie

The identity interrogative. Means 'Who?' or 'Which?' Has a more limited function than formerly.

Lexeme JA
: ja
(also CPDs raja, toja, etc.)

The metaphorizer, a gobbled right-mark. Its CPDs modify its semantics only.

Lexeme JE
: je

The first link in forming linked args.

Lexeme JI
: ji

The 'who is...' operator.

Lexeme JIO
: jia jio

The subordinate clause "conjunction".

Lexeme JUE
: jue

The 2nd-&-subsequent (sutori) link in forming linked args. There is no explicit 3rd or 4th link now.

Lexeme KA
: ka ke ko ku
(also CPDs kanoi, nuku,
kuikou, kuinukou, etc.
Possibly also kuipa,
kuivi, kuisea, etc.)

The general kek, used with KI to forethink connections. The KUI-CPDs are phonemically awkard. We need a better way of identifying the causal (and possibly other PA-form) keks; see PA for the words now lexemically equivalent to kou.

Lexeme KI
: ki
(also the CPD kinoi)

The kek-infix, as in 'ka...ki...'.

Lexeme KIE
: kie

Left-paren. The Preparser gobbles the parenthetic expression into this lexeme.

Lexeme KIU
: kiu

Right-paren. Never seen by Parser.

Lexeme KUI
: kui
(used only in KA-CPDs)

A prefix used to make keks out of the causal PAs...possibly other PAs as well. A temporary morphophonemic solution; see KA above.

Lexeme LAE
: lae sae

The "pointer" descriptor.

Lexeme LE
: le lo la lua lea

All other descriptors except LIO. These are not recursive; the pointer is.

Lexeme LEPO
(recognized by CPD-lexer)

The event-descriptor; makes subordinate clauses.

Lexeme LI
: li

Left-quote. The Preparser does not gobble these quotations; the Parser parses them.

Lexeme LIE
: lie

Left-strong-quote, used with a freely-chosen pair of identical terminators. The Preparser gobbles the quoted string into LIE before even attempting to lex it; thus the string quoted can be foreign...or nonsense.

Lexeme LIO
: lio

The number-descriptor.

Lexeme LIU
: liu

The single-word quote. The quoted Loglan word is gobbled into LIU before parsing.

Lexeme LOI
: loi

Greeting word; and, now, sign of vocatives after names and of the new "Carter-Vocs".

Lexeme LU
: lu

Right-quote in 'li...lu' quotation.

Machine-Lexemes

Lexeme M4
: M4
(inserted before PA
before pred-signs)

The first of the "Machine-Lexemes". M4 is a sign of a PA used as a predicate-inflector. The M's start with M4 because the 3 earliest ones were found unnecessary.

The Preparser can recognize things like "pred-signs" without doing any parsing.

Lexeme M5
: M5
(inserted before KA
before pred-signs)

The Preparser inserts the M-lexemes algorithmically; the Parser treats them as words; and the Postparser eliminates them and all signs of their having been there.

Lexeme M6
: M6

Considering that all free mods have been gobbled before M-insertion, lookahead extension is never more than LR2 except for M7 and

(inserted before A
before pred-signs)

the possibility that the negatives looked-over for M11-12 may be recursively repeated.

Lexeme M7
: M7
(inserted before BUA
in prenex quantifiers)

M7 has the "long lookahead". It can be as long as any prenex quantifier.

Lexeme M8
: M8
(inserted before KA
before JE/JUE)

All other lookaheads are well within the range of what we humans easily do. In 8 out of 9 cases, M-insertion is just "looking over", i.e., on the other side of, a single word, or a recursive clump of words, before deciding what to do. Most often that looked-over word is an A or a KA.

Lexeme M9
: M9
(inserted before A
before JE/JUE)

Lexeme M10
: M10
(inserted before A
before PA/JI/JIO)

For M4 it was a PA that was looked over.

Lexeme M11
: M11
(inserted before NO
before GA/POGA/M4)

For M11 & M12, it is the negative NO, which may be a recursive clump of NO's.

Lexeme M12
: M12
(inserted before NO
before PA)

Lexeme ME
: me

The "predicator". Monolexic.

Lexeme NI
: ho ni ne to te fo fe so
se vo ve pi re ro ru sa
se si so su ma mo kua
(also CPDs neni, nenisei,
iesu, ietoni, etc.)

All number words; indeed, all mathematical expressions including dimensioned numbers. NI will of course eventually have its own internal grammar: the "expression" part of MEX.

Lexeme NO
: no

The negative: one of the most slippery words in any grammar.

Lexeme NOI
: noi

The negative suffix: used only by the Preparser in lexing CPDs.

Lexeme NU
: nu fu ju
(also CPDs nufu, nufuju, etc.)

The conversion operators.

Lexeme PA

: va vi vu pa na fa via vii viu ciu dia duo kae lia lui mou neu pie rui sau sea sie tie kou moi rau soa
(also CPDs pana, pazi, pacenoina, etc.)

A great congeries of words are now PA: the tensors, locators, modals and causals. This leads to a major unification of the grammar.

Lexeme PAUSE
: , #

This is "lexemic pause"; formed by the Preparser and used sparingly in the grammar. It is always accompanied in the grammar rules by GU as a high-noise alternative.

Lexeme PO
: po pu zo

The abstraction operators; always short-scope when not in LEPO or POGA CPDs.

Lexeme POGA
: (recognized by CPD-lexer)

A new CPD: it gives PO long scope.

Lexeme PREDA
: he bi bie dua
(also all pred-wds found by lexer and CPDs rari, nenira, sutori, etc.)

All predicate words except the numerical predicates and BUA. Notice that PREDA includes the little word predicates and the new predicate interrogative he. dua is in PREDA temporarily. dua & kin should probably have their own lexeme.

Lexeme PUA
: pua pue pui puo puu

The HB-Tags: the argument ordinals.

Lexeme RA
: ra ri

These two words are not in NI only because the the Preparser needs them to recognize numerical predicates.

Lexeme TAI
(simple forms like ama bai cai tai tei teo tao are recognized by the lexer; also CPDs like baicai, ebaicai, ebaiocai, haitosaiofo, etc.)

All the letter-variables and the acronyms made from them and from number words. Acronym-making is new; the Corpus exhibits the procedure.

Lexeme UI
: ua ue ui uo uu oa oe oi ou ia ii io iu ea ei eo eu ae ai ao au bea beu cia coa dau diu dou feu foi gea kau kia kuo lau nau nea nie pae pou rae sui voi loa sia siu
(also CPDs nahu, vihu, kouhu, duohu, nusoahu, etc.)

Another congeries. This one contains the attitudinals, discursives, loa, sia & siu, and the new hu-CPDs. Another major unification: this one a vast simplification via the notion of "grammatical noise"...noise which is now being filtered out before parsing by the "gobbling up" of these free mods by the Preparser. This move leaves the Grammar free to deal with the real grammatical issues presented by the utterance.

```
Lexeme ZI
   :    za zi zu
```

The tense auxiliaries; these occur only in CPDs.

THE TRIAL-19 GRAMMAR

Section A. Punctuators

```
err          =>  error

ga           =>  err
             =>  GA

gu           =>  err
             =>  GU

gue          =>  err
             =>  GUE
```

These are the 3 optional punctuators & the benign "error" grameme that makes them possible. That is, Yacc regards the absence of a punctuator where one "should" occur as an "error". But it then goes on...presumably to find and report more "errors". Thus we are using a feature of Yacc originally designed for an entirely different purpurpose, namely to build compilers that diagnose faulty programs. But we are using its error-tolerance to provide our grammar with an elegantly humanoid optinality of punctuation.

GA bounds descriptions; GU bounds argsets, and so, predexps; and GUE matches some GE in a predstring. And none needs to be expressed unless it is actually to be used to alter the structure of the utterance.

Section B. Linked Arguments

```
links1       =>  JUE arg
             =>  JUE arg links1 gu
```

The order of gramemes is the one in which the listener would search them, i.e., from the "leaves" of the parse-tree to the "root".

```
links        =>  links1
             =>  links M9 A links1
             =>  M8 KA links KI links1
```

So we start with linked args, which occur in pred-strings, which occur in arguments; and so on.

```
linkargs1    =>  JE arg gu
             =>  JE arg links gu
```

Notice that we are already relying on M-lexemes to tell us that these eks & keks are followed by JE/JUE.

```
linkargs    =>  linkargs1
            =>  linkargs M9 A linkargs1
            =>  M8 KA linkargs KI linkargs1
```

The 1st optional gu's appear here as well; these are the only gu's that occur inside pred-strings. It is one of the major unifications of this grammar to treat the internal "specifications" of pred-strings and the external links between the arguments of a predicate as instances of the same grammeme. Thus linkargs get into the grammar at only one place: in predB in the next section.

Section C. Predicatively-Used Predicate Strings

```
predA       =>  BUA
            =>  PREDA
            =>  NU BUA
            =>  NU PREDA
            =>  GE kekableF gue
            =>  ME argument gu
```

We now commence building predstrings. Notice that BUA/PREDA have parallel roles. In fact it is only BUA's role in prenexes that keeps it out of PREDA. 'kekable-' means a predstring that may have a kekked pair of predas at its head. The distinction will be important for descriptions.

```
predB       =>  predA
            =>  predA linkargs
```

predB is a single pred word, or ge/gue-ed string, or me-ed argument, to which linkargs can be attached. This is a superset of what we actually do.

```
predC       =>  predB
            =>  NO predC
```

predC provides for the recursive negation of predB's.

```
predD       =>  predC
            =>  PO kekableA
```

predD provides for the abstraction of predC's. kekableA is simply the kekable version of predC; see below. In other words, after a PO the predC CAN be "head-kekked".

```
predE       =>  predD
            =>  predD CI kekableD
            =>  CUI kekableC CA kekableD
```

predE provides for both CI-ing and "CUI...shekking". Note that kekables may occupy non-initial positions in the growing pred-strings.

```
predF       =>  predE
            =>  predF CA kekableD
```

More shekking, this time without CUI.

```
predG       =>  predF
            =>  predG kekableE
```

Recursive concatenation. This builds the strings. Notice that the right-hand forms are always kekables.

```
predstring  =>  predG
            =>  predG GO kekable
```

Inversion with GO. "kekable" on the right is the largest kekable string. predstring will be used later as the predicate portion of a predexp, i.e., before argsets are attached. It is used in only one place in the grammar: in the first rule ("barepred") of Sec. G. We turn now to the predstrings used in descriptions.

Section D. Descriptively-Used Predicate Strings

```
kekableA    =>  predB
            =>  M5 KA kekable KI kekableA
            =>  NO kekableA
```

Pred-strings used in descriptions have one privilege that pred-strings used predicatively don't have: they can have kekked head-predas.

```
kekableB    =>  kekableA
            =>  PO kekableA
```

So kekableA repeats predC but with the kekking allogram added. And kekableB repeats predD.

```
kekableC    =>  kekableD
            =>  kekableC kekableD
```

kekableC produces a concatenation that is used in one place only in both series, namely in CUI...CA in both kekableD and in predE above.

```
kekableD    =>  kekableB
            =>  kekableB CI kekableD
            =>  CUI kekableC CA kekableD
```

kekableD repeats predE but uses kekables in both halves.

```
kekableE    =>  kekableD
            =>  kekableE CA kekableD
```

kekableE repeats predF.

```
kekableF    =>  kekableE
            =>  kekableF kekableE
```

kekableF repeats predG.

```
kekable     =>  kekableF
            =>  kekableF GO kekable
```

kekable is the end of the sequence, and so corresponds to predstring. It is in fact, a pred-string with the possibility of kekked head-predas. Since such strings would fall apart if used predicatively, the kekable grameme is used only in descriptions; see Sec. F below.

Section E. Term & Utterance Modifiers

```
gap         =>  GU
            =>  PAUSE

mod1        =>  PA gap
            =>  PA argument gap
```

These forms, unlike the gobbled free mods, have meaningful attachments: when non-initial, to some "term" of the utterance; when initial, to the utterance as a whole. A term is either an argument or a predicate.

```
mod2        =>  mod1
            =>  M12 NO mod2
```

Notice that GU and PAUSE are "gap" options here, as for high vs. low noise conditions,

```
                                            or with machine vs. human interlocutors.
                                            gap occurs once more: in neghead in Sec. H.

mod          =>  mod2                       GI is a semantic signal that the mod, how-
             =>  GI mod2                    ever attached, is to be taken as modifying
                                            the utterance as a whole.

argmod1      =>  JI arg2                            The mod-forms may apply to predi-
             =>  JIO sentence gu                    cates or utterances as well as to
                                                    arguments.  The argmod-forms at-
                                                    tach only to arguments.

argmod2      =>  mod                                A defect of the current grammar is
             =>  argmod1                            that mods are not yet kekked and
                                                    ekked, and that argmods are only
                                                    ekked.  This will be repaired.

argmod       =>  argmod2                            argmod is used only in arg2 in the
             =>  argmod M10 A argmod2 gu            next section.  mod is used in both
                                                    Secs. G & H, where it will be at-
                                            tached to both predicates and utterances.
```

Section F. Arguments & Argument Sets

```
mex          =>  RA                                 The only reason RA & NI are in sep-
             =>  NI                                 arate lexemes is that they are dif-
                                                    ferently involved in the recogni-
                                                    of CPDs.

descriptn    =>  LE kekable                         Note that kekable is the operand
             =>  LE mex kekable                     of description.  This is the pred-
             =>  LE arg1 kekable                    string with possible kekked head-
             =>  LE mex arg1                        predas that was built in Sec. D.
                                                    In a description such a string
                                                    cannot "fall apart".

name         =>  DJAN                               Name is left-recursive, the general
             =>  name DJAN                          human pattern whenever indefinite
                                                    continuation is possible.

arg1         =>  DA                                 The list of argument forms.  HU is
             =>  HU                                 the interrogative arg.  TAI is a
             =>  TAI                                letter-variable or an acronym.
             =>  M7 BUA                             M7 BUA is that special use of BUA
             =>  LE name                            in prenexes.  Note that 'la' is
             =>  LIO TAI                            now an allolex of LE.  The LIO TAI
             =>  LIO mex                            form is for the representation of
             =>  descriptn ga                       mex by letter-variables.  The oper-
             =>  LIU                                ands of LIU and LIE are gobbled;
             =>  LIE                                so they alone need to be shown to
             =>  LI utterance LU                    the parser.  But note that LI...LU
             =>  LEPO sentence                      forms are actually parsed.  The
                                                    LEPO-form is the "big one" here.

arg2         =>  arg1                               arg2 provides for modifying arg1
```

```
             => arg1 argmod gu
             => KA argument KI arg
```

and for kekking arguments in general. It uses arg & argument below.

```
arg          => arg2
             => mex arg2
             => PUA arg2
             => IE arg
             => LAE arg
```

arg's are the arg2's quantified with mex, tagged with PUA, recursively questioned with IE or made into pointers with LAE.

```
argument     => arg
             => argument A arg
```

And finally an argument is either an arg or a string of ekked args.

```
arguments    => argument
             => arguments argument
```

And arguments is a left-recursively concatenated string of such strings.

```
argset1      => gu
             => arguments gu

argset       => argset1
             => argset A argset1
             => KA argset KI argset1
```

We come now to argset, one of the more powerful structures in this grammar. Note that argset can be null: an optional gu. What this means is that predicate expressions, which are made up of predstrings and argsets in the next section, always end with an optional gu even if they have no arguments attached. Note also that argsets may THEMSELVES be kekked and ekked, and that this maneuver must be somehow kept distinct from the kekking and ekking of the arguments themselves. That it IS kept straight is one of McG's more mysterious accomplishments.

Section G. Predicate Expressions

```
barepred     => predstring argset
             => predstring mod argset
```

The basic predicate, "barepred", is a predicate without a tense op or other leading mark. It's made of a predstring + an argset with an optional mod between them.

[Note: The ternary form of the second barepred allogram is a mistake. It leaves the "clamping question" unanswered for the human mind...indeed, for a mentally humanoid machine: What does mod modify? Predstring or argset? This rule slipped by our efforts to "humanize" the parses produced by the grammar and will be replaced with a couple of binary rules in Trial.20. Probably no specimen revealing the "meaningless" parses produced by this allogram occurs in the Corpus.]

```
markpred     => POGA sentence
             => M4 PA barepred
```

Next come the "markpreds". These are either POGA-forms or barepreds marked

```
              =>  GA barepred

backneg       =>  markpred
              =>  M11 NO backneg
```

either by a GA or by a PA foretold by its M4. The GA-allogram will be used only for forming negatives. After descriptions the Parser will always take any GA to be the optional description terminator no matter which GA the speaker thinks he's used!

```
backpred      =>  barepred
              =>  backneg
```

Only markpreds can be negated, i.e., given long-scope no. The 1st round of this is in backneg, which is used only in "backpreds", i.e., in the right part of an ekked pair.

```
bareekpred    =>  barefront M6 A backpred
```

If the front end is bare, it's a "bare ek-pred"; if not, it's a "mark ek-pred"; see below.

```
barefront     =>  bareekpred argset
              =>  barepred

markekpred    =>  markfront M6 A backpred

markfront     =>  markekpred argset
              =>  markpred
```

Barefront and markfront help to manage this. Note the two tracks through the grammar which preserve this distinction. Not only negation but the definition of imperatives will depend on it.

```
frontneg      =>  markfront
              =>  M11 NO frontneg
```

Now we can negate the "markfronts" to get the frontnegs.

```
imperative    =>  barefront
              =>  frontneg
```

And finally we can define "imperatives", i.e., predicate expressions without 1st arguments.

```
kekpred       =>  M5 KA predexp KI predexp

predexp       =>  imperative
              =>  kekpred argset
```

And a predicate expression is either an imperative or a kekked pair of such imperatives. The latter will have their own joint argset, even if null, and may of course be nested.

Section H. Sentences & Utterance Parts

```
statement     =>  argument predexp
              =>  NO statement
```

A statement is an argument plus a predicate expression, possibly recursively negated.

```
sentA         =>  statement
              =>  imperative
              =>  keksent
```

For the purposes of kekking sentences in all possible ways, we form the class of statements, imperatives, and kekked sentences.

```
keksent     =>  KA sentA KI sentA
            =>  M5 KA sentA KI sentA
            =>  NO keksent
```

We then provide for kekking sentences through two distinct allograms. The one with M5--which shows that there is a pred-sign beyond the KA--will catch the ones with imperatives as antecedents. The one without M5 will catch the ones with statements as antecedents. And, of course there is provision for recursive negation of the result.

```
sentence    =>  sentA
            =>  PA sentA
            =>  mod sentA
```

We then allow these objects to be frontally modified in 2 ways. We now have something that can be called a sentence.

```
uttA        =>  A
            =>  NO
            =>  mex
            =>  mod
            =>  arguments
            =>  sentence
            =>  arguments GOU sentence
            =>  arguments GUU sentence
```

We next create a list of utterance types, starting with various fragments of sentences, and ending with two special classes of sentences, namely those with prenex quantifiers marked by GOU, and those with shifted arguments marked by GUU. This is oversimple still. The next yaccing problem is to bury prenexes more deeply in the grammar by studying their interactions with negation and argument-shifting.

```
headmod     =>  UI
            =>  LOI
            =>  KIE
            =>  DJAN
```

Since free mods are gobbled into the preceding lexeme, they now have to be accommodated when initial.

```
uttB        =>  uttA
            =>  headmod
            =>  headmod uttA
```

So the 2nd class of utterances are the ones that may or may not be fitted with these "headmods", or that may simply be such a headmod.

```
neghead     =>  NO gap
            =>  headmod NO gap
```

A special provision must now be made for global utterance negatives. These may or may not be preceded by headmods. "gap"--which is either a PAUSE or gu, remember--is used to set them off from more closely-attached negatives.

```
uttC        =>  uttB
            =>  neghead uttC
```

These negheads can then be recursively attached to utterances.

```
utterance   =>  uttC
```

And finally, continuing utter-

```
=>  I uttC                            ances marked with I-words are
                                      provided for.  But in this grammar,
                                      these "continuing" forms are simply
                                      "utterances".
```

What has not been done is provide for the left-recursive concatenation of these continuing utterances into strings which are utterances in a broader sense. This omission is both temporary and deliberate. It makes the parses of the Corpus easier to read when the specimens happen to be strings of short "utterances" as defined in this narrow sense. The Parser now parses such utterance-strings one utterance at a time and then simply concatenates the parses to create the parse of such a specimen.

End of Trial.19 Grammar