

# Ideas for the chunk parser

Reiner Wilhelms-Tricarico @ Fonix

last update: 12/12/2005

*This is a loose collection of ideas about the chunk parser, which later became the 'sequence parser'.*

## 1 Overview

The purpose of this writing is to come up with a work plan for building a parser for prosodic parsing rules. What's written here are notes and ruminations, not quite a blueprint. By the time this becomes a blueprint, I have probably begun building it in c-code.

Corine had written down a bunch of preliminary rules that are meant to find out "chunks" in a phrase which have either a particular predictable stress structure, or lend themselves to be demarked by pauses. For some of the rules, *chunking* means breaking up the phrase in pieces, for others it means highlighting and "chunking together". The goal and purpose of this parsing is to generate an intonational structure, pauses, and stress markers that, in addition to the lexical stress in each word, will be utilized to generate a pitch modulation and intonational pauses as well as particular shortening and elongations of parts of a phrase. This parser needs to be working at least in a "quick and dirty" way so it can become part of small-footprint speech synthesis.

## 2 First stage: A parser that highlights

The first stab at this problem is to find a simple but flexible meta-language to describe the chunking rules. This meta-language has to be designed so that a parser can be written that (i) checks the validity of a set of written rule as being statements in that language, and in its further development (ii) translates the rules into a set of parsing statements which can be applied to English written sentences in pursuit to find the chunks and other pieces of information relevant for prosodic structure. In all this it is most important to keep the language simple enough so that formulation of rules require very little training.

The similarity between this problem and the previously written **bachus** method, which had been applied to German word morphology, is the following:

In the language describing a word grammar in **bachus**, the terminal elements are morphemes, and the grammar which is a grammar for forming words of the target language (so far only German) can refer to groups of morphemes by a designated name. For example, there can be the (token-) name 'always-stressed-prefix'. This name refers to a group of particular prefixes that always need to receive the highest stress within a word. The word 'always-stressed-prefix' is then used in the written word grammar rules as a representative for any of the prefixes in that particular morpheme list, which is usually written down in a file whose name is associated to the token 'always-stressed-prefix'. (The word grammar then also has to deal with the way the stress is handled if there is more than one of these prefixes at the beginning of a word, which complicates the issue a bit but could be solved by writing rather extensive word grammars.) The difference between the prosodic structure parsing and **bachus**' parsing is that **bachus** operates on individual words only, by its original purpose and design, while here we need to operate on multiple words in a sequence and possibly on entire phrases instead.

It is important to note that the purpose of the new parser shall *not* be to obtain the entire grammar tree of a phrase, which might be a linguist's pursuit. Rather it is to provide annotation of the parts of the phrase for which the parser finds a rule (or multiple rules) while it ignores the rest of the phrase for which it can not account for. As discussed, the first version of the to be designed parser will be a thing one might call an annotation machine: The user enters a phrase (an English sentence or a sequence of English words) and the parser spits out the same sentence together with some kind of representation that represents "colored markers", using different colors for different rules.

## 2.1 Description of the parser

The following concepts may be used in the parser:

- Word groups: This refers to a file containing a group of words, for example, all articles, or all reflexive pronouns. The word group is represented by a token name and an associated file name. Thus, the meta-

grammar under construction needs to contain a declaration statement that declares a word group to be associated with a token name.

- Context (quite fuzzy still): The grammar has to provide for the possibility of words (one or many) forming a left or right context for the application of a rule while not participating in the rule themselves. In the rule these may be represented for example by simple capital letters such as 'X' and 'Y'. In this category are also special contexts such as beginning of sentence, end of sentence as left or right context of a rule.
- Word forming rules: As a subset of the rules in the parser, we need to be able to parse words similar to *bachus*. For example consider expressions such as 'had listened for', 'had written down', 'was given up', 'is ridden with', etc., which can all be seen as examples of the pattern rule: Auxiliary verb form + verb form + preposition. This requires a system that can recognize a verb form, namely 'listened', 'written', and 'given', respectively, if the original roots are available as a word group. Hence, parts of the parsing that take place in *bachus* have to become nodes in this parser, as if the new parser was using *bachus* as a subroutine to check out words by word grammar rules, but in the context and under the auspice of a phrase forming or chunk forming rule.
- Chunking rules and separator rules: A chunking rule is a rule which if it applies to a word sequence, the word sequence is likely to be a chunk. A separator rule is a rule that if it applies to a word sequence, the word sequence is not part of one chunk but must instead be split somewhere - the rule (understood as transformation rule) results in the insertion of a separator.

### 3 More ideas for realization of the parser

*Bachus*' domain was the individual word. A word grammar is essentially the description of a network of possible paths that lead from the first letter of an input word to the last letter. If we restrict attention to a certain class of words, for example verb forms, then the word grammar describing verbs is a subgraph in the word grammar (whereby each graph or subgraph can consist of many paths).

### 3.1 The wrong approach

One could conceptualize a simplistic extension of *bachus*. This would be to apply *bachus* still to each word one by one, and have it generate additional information. For example, if a word can be parsed according to a word grammar, then it is known to the parser (*bachus*) which path was taken to arrive at the end of the word. This information tells us for example that a word like *fishing* was found to be a gerund verb form. Applied to all words of a phrase, *bachus* can likely provide information about most of the words as a sequence of symbols. Subsequently, another “higher level” parser can evaluate the sequence of symbols (according to another to be designed symbol sequence grammar) and check for the presence of certain patterns. For example, *bachus* determines parts of speech (what’s a verb, and what’s a preposition, etc.) and then a second parser looks at this information and decides whether or not a certain phrase-morphological structure can be found or not.

### 3.2 A better approach

Probably such a simplistic extension would not lead very far since a lot more words might look like, for example, a verb form while they are in reality something else (e.g., ‘read’ in: “This book was a nice read”). What it tells us is that any “brainless” (see above) annotation of word form that acts independently on each word may be too unreliable. However, combined with top-down information, it may be very usefull. In the example sentence, “this book was a nice read”, if we were to look for patterns such as *article + adjective-form + noun-form*, the sequence “a nice read” would obviously fit the pattern rule. In fact, it can be widely excluded that “read” is a verb in this context if we accept that a pattern *article + adjective-form + verbform* does not exist at the end of a sentence. It’s like knowing that the very bright star in the east that appears an hour after sunset can not be Venus because Venus can’t be in opposition to the sun. (Venus circles the sun on a smaller orbit than earth, hence seeing the planet in oposite directions from earth isn’t possible, therefore, it’s got to be Jupiter).

### 3.3 Consequences for the structure of a phrase parser

We will have two types of grammars to deal with: A grammar that describes chunks in a phrase, and a grammar that describes word morphology. (Later other structures may have to be added to further constrain the application of particular chunk rules by establishing relations between them)

The grammar that describes chunks in the phrase uses mainly statements of the word grammar as non-terminals, but can also simply refer to individual words, e.g., “and”, “or”, “to”.

To get the general idea:

The following may be a statement of the grammar that we are looking for.

$$\begin{aligned} \text{chunk} &\rightarrow \text{article} + \text{adjective-form} + \text{noun-form}. \\ &\quad | \text{article} + \text{adjective-form} + \text{verb+ing-form} + \text{noun-form}. \\ &\quad \dots \\ &\quad | \text{“much”} + \text{comparative-form} \end{aligned}$$

The non terminal tokens are: *adjective-form*, *noun-form*, *verb+ing form*, and *comparative-form*. Each operate on the unit of one word. The particular non-terminal *article* is simply a list of words. Each of the above used non-terminals, however, are tokens for an entire subgraph of a word grammar. (Of course, “much” is a terminal.)

During parsing the parser may invoke any of the chunk rules, starting at some word in a phrase and apply its token rules to consecutive words. Hereby applying a token to a word means parsing the word according to a specific word grammar. We then have to write down precisely what, e.g., *noun-form* may all be.

To get there quickly, we shouldn’t worry too much about linguistic sophistication. If there is a particular chunk rule that needs some kind of understanding of what a verb form is, there may be another chunk rule that uses a different understanding of what a verb-form is. So there can be a whole plethora of “verb-form” each following particular word grammar descriptions. We are likely to arrive at a rather large number of rules for chunks anyways.

The first version of the parser will try each rule that it finds and apply it to part of a phrase. If a rule applies, a reference to the rule (in the comparison color of the marker) will be used to mark the scope of where the rule applies. This is the highlighting process.

The next extension of the parser will be to go beyond highlighting and

insert direct prosodic markers or “functors” into the text that can be used by the speech synthesiser to influence pitch contour, pause structure, and determine timing and durations.

## 4 Discussion of rules

Now let’s go through some of Corine’s rules, try to make sense of them, while at the same time developing some further ideas of what a meta-language needs to cover that makes it easy to write down these and similar rules.

Here is one:

$$Subj-pronoun+Seq+“to”+Seq+X \rightarrow Subj-pronoun+Seq-H^*+“to”+Seq-H^*+X$$

The intention seems to be that this is a transformation rule. It’s meaning is exemplified by “You are going to visit the museum after ...”, in which the word “going” and the word “museum” receive stress (in the transformation rule indicated by the  $H^*$  mark).

In this rule we have the non-terminal symbols  $\{Subj-pronoun, Seq, X, Seq-H^*\}$ , and the terminal symbol “to”. The non-terminals need to be replaced by rules that explain the them, ultimately in terms of terminals. The simplest way to do this, is by making a direct declaration that all symbols corresponding to, say, *subj – pronoun* are contained in a list (which may be on a file). The list (file) then would contain terminals like { “I+am”, “you+are”, etc., }.

*Seq* is described as word or word sequence, while *X* is described as marked word of a group of categories. The

Here is another example of the Corine’s rules:

$$Pron - Subj + Seq + “to” + Seq + “to” + Seq \rightarrow first Seq contains Verb$$

However, this appears to be just a conjecture, not a transformation rule.

In order to arrive a viable syntax I try to continue to write down example after example and see:

Example: “much hotter here in the front room”. Asks for a pause after ‘hotter’.

Rule:

$$chunk \rightarrow “much” + comparative form + \{pause\} + seq$$

And a little more general, where “way” takes the role of “much”.

$$chunk \rightarrow emphaticqualifier + comparative form + \{pause\} + seq$$

$$emphaticqualifier \rightarrow “much” | “way”.$$

In the above rule statements the word pause is inserted in curly brackets for the following purpose: If the parser goes through the sequence, e.g., “way more appreciated in Alaska” and indeed finds this to be according to the rule above, it can directly leave a trace by inserting the pause (in this example) into the string of properly parsed parser expression. The way this works is that the parser is operating on a stack in which all properly found nodes of a path are found. If a rule could be applied successfully, the stack contains at the end the information on how each word was classified in applying the parser expression. It also contains the symbol pause at the right point in the sequence.

This is indeed the same method on how I introduced stress markers within a word in bachus. For example, a rule there was:

`simpleword => prefmorph+<'>+root+flex[0,1]`

If a word would follow this rule, the stress marker (i'j) at the end is at the right spot in the stack. Now I will extend this method for phrase parsing and insert all kinds of symbols. Essentially the above rule statement

$chunk \rightarrow \text{“much”} + comparative\ form + \{pause\} + seq$

is equivalent to the replacement rule statement:

$\text{“much”} + comparative\ form + \{pause\} + seq \rightarrow \text{“much”} + comparative\ form + \{pause\} + seq$

I prefer strictly the short way of writing it since I know how to implement this in a parser.

Let's go on with more examples.

Example: “Did you forget to set your watch?” Pause after verb before 'to'. Resulting rule:

$chunk \rightarrow do - verb - form + \{pause\} + \text{“to”} + verb - infinitive.$

Some of the prosody rules are slightly complex.

I try to rewrite these in terms of a parser language that begins to emerge.

1.) Example: “you are *going* to visit the museum after...”

The rule that the second syllable of museum is stressed is a consequence of lexical stress, which in turn is actually a consequence of the latinate suffix rule: museum, colosseum, lyceum. In fact, Ed and I observed that the stress in museum is achieved by a strict F0 lowering in the second syllable of the word.

Nevertheless, the stress on 'going' is a prosodic mark. So here would be the rule:

$chunk \rightarrow subj\_pron + Seq\_H^* + "to" + Seq\_H^*$

We had this before. I revisit it here since it we need to find a rule that describes  $Seq\_H^*$ . At the moment I don't know what this looks like but it must be of a form that contains on the right side the symbol insertion  $\{H^*\}$ . For example:

$Seq\_H^* \rightarrow verb\_form + direct\_article + \{H^*\} + noun$

I think from now I stick to the ones that I understand: (Corine's Prosody marker rule no 5) Example: I was planning to go.

$chunk \rightarrow subj\_pron + \{L^*\} + be\_form + "to" + verb\_form$

Rule no 6: Did you forget to set (...your watch)?

$chunk \rightarrow do\_form + \{H^*\} + verb\_form + "to" + verb\_infinitive$

Rule no 7:

$chunk \rightarrow (pronoun|conjunction) + say\_verb\_phrase + \{pause\}$

I realize I still don't know how to make context dependent grammar rules.

## 5 An attempt to see statements in the parser's language

Rather than debating meta structure I'll put the rules in a form for which I am fairly sure I can write a parser. So I first write down expressions in the language I am looking for and then build the meta-language. This is what I have currently:

```
RULE:      PAUSEMARKER
          | PROSODIC_CHUNK
          | HIGHLIGHTER
```

```
PAUSEMARKER: DOVERBFORM + <PAUSE> + "to" + VERBINFINITIVE + SEQUEL
            | COMPARE_EMPH + COMPARATIVEFORM + <PAUSE> + CHUNK
            | NOUNFORM + "of" + ARTICLE + <H*> ADJECTIVEFORM + NOUNFORM
```

```
PROSODIC_CHUNK: SUBJ_PRON + <L*> + BE_FORM + "to" + VERBINFINITIVE
              | SUBJ_PRON + BE_FORM + <H*L> + COMPARATIVEFORM + SEQUEL
              | DO_FORM + <H*> + VERBINFINITIVE + "to" + VERBINFINITIVE
              | SUBJ_PRON + BE_FORM + <H*> + VERB_ING_FORM +
              DIRECTIVE_PRONOUN + NOUNCHUNK
```



```

NOUNCHUNK:    ARTICLE + NOUNFORM
              | ARTICLE + ADJECTIVEFORM + NOUNFORM

COMPARE_EMPH: "much" | "many" | "way"

SUBJ_PRON:    WORDLIST("subjectivepronouns.txt")
NOUNFORM:     WORDGRAMMAR("nounforms.grm")
VERBFORM:     WORDGRAMMAR("VERFORM.GRM")
VERBINFINITIVE: WORDGRAMMAR("VERBINFINITIVE.GRM")
VERB_ING_FORM: WORDGRAMMAR("verb_gerundive.ger")

```

This certainly requires a bit more clarification. The Baccus Naur method of writing rules is here applied. And to keep things simple, not all non-terminals are spelled out. In principle, for the grammar to be complete, each non-terminal has to appear on the left side so that eventually it is entirely resolved into terminals. I want to make the statements as much as possible non-recursive so I can avoid writing complicated code that checks for loops. If there are no loops it means the resulting grammar has a strict tree structure.

Things between sharp brackets,  $\langle \rangle$ , such as PAUSE, means that an operator symbol (PAUSE) gets inserted into the word sequence. Some of the non-terminals in this text are calls to bachus: They are indicated at the end of the grammar file as declarations of a word grammar which can be found on a special file. The word grammar files (now we have multiple, before there was only one) then in turn may access different files to build a word specific grammar, e.g., all verb infinitive forms follow the (bachus-) rules written in a file VERINFINITIVE.GRM (see above).

## 5.1 Ideas on the processing

At this point it isn't entirely clear how to negotiate the contradiction that appears here: The rules as I have written them down so far apply to part of a phrase, in fact, only to a sequence of a few words, while the domain of the word sequences on which we want the thing to operate is, at least, the entire phrase. Cutting an entire phrase into parts that then would be parsing certain statements in the collection of rules can not be done independently of the rules.

We are not planning to build a parser that contains mainly rules to analyse an entire phrase. The things written down so far are only “local”: The rules are for 2,3, or some more words but there is hardly a rule that covers a whole sentence. Which means that this grammar is incomplete, and will stay incomplete.

I propose to look at the processor to be built as a thing that detects: It detects the presence or absence of particular effects in a phrase. What it can detect is described by its flexibly configurable rules. The detection process itself is a hierarchical process: On the highest level the truth or falseness of certain word sequence rules is tested, while on a lower level the existence of certain morphs may be tested, and finally on the lowest level the existence of a certain letter. The processing is top down: The uppermost rules determine which tests are done on the words, and word grammar rules determine for which letter we are looking at a certain place within a word.

There are in principle two types of low level parser calls: Word grammar and lexicon access. For example: If the word oesterize is present in the lexicon and is labeled as verb (VB) then we have a simple way to find that it constitutes a verb: The top parser asks the dictionary: Is oesterize a verb? The dictionary responds with the phoneme information and the information of this being a verb. Or in another case, the dictionary comes back with the question being undecidable. On the other hand the word grammar processor might come up with: oesterize is a verb since it ends in -ize.

Disect a bird:

DO\_FORMQ + <H\*> + VERBINFINITIVE + "to" + VERBINFINITIVE

This rule applies for example to the sentence part “do you forget to set”. So we need a subrule for DO\_FORM, for example:

DO\_FORMQ: VDB + PERPRONSUBJ

VDB: "do" | "did" | "does"

PERPRONSUBJ: "I" | "you" | "he" | "she" | "we" | "they"

This partial grammar obviously allows to return as true if the input is “do I”, or “did he”, but also for “does you”. (If we want to be even more accurate about this, the rules have to be written accordingly).

The next part is “;H\*;”. This is simply the symbol that gets inserted if the complete path in the grammar can be parsed.

VERBINIFITIVE is true if the next word is a verb in the infinitive form. The module for this consists of a quest to the lexicon, whether the word form is a verb infinitive. If undecidable from the lexicon, we must have written a default word grammar for this, for example:

```
VERBINFINITIVE:  WORDGRAMMAR("verbinfinitive.wdg")
```

This of course pushes the bucket only to bachus to figure it out. The wordgrammar for verb-infinitive needs to be in a seperate file since it may be fairly long but, more importantly, since its rules are written in a different syntax.

Bachus' primary purpose was to deliver phonemes to pronounce the word and "lexical" (word-) stress. Therefore, the answer to the question, whether or not the thing that we ask bachus to classify as a verb, is true if bachus returns a phoneme string, and it is false if no statement in the word grammar could be found that matchees the word string.

On the other hand, the above stated structure for DO\_FORMQ can only answer the question of whether or not "did you" is a DO\_FORMQ or not. It doesn't deliver the phoneme string. *Maybe that should be not so. It may be more appropriate to extend the concept that at any rate the terminal operators in all cases return a phoneme string.*

The last remark points to the need to replace also the simple question "to" by something else. And since we already know that "to" gets pronounced in different context in different ways, there is also a method to write this down right here. Let's write it TOINFINF:

```
DO_FORMQ + <H*> + VERBINFINITIVE + TOINTERINF + VERBINFINITIVE
...
TOINTERINF:  <LONG> "to"
```

There can be another rule that fits a sentence like "would you like to dance?". In this, the inquiry for "to" would be formed in a rule such as:

```
CORDIALQUEST + VERBF + TOINF + VERBINF
...
CORDIALQUEST:  ( some subgrammar)
TOINF:  <SHORT> "to"
```

The point here is that "to" is pronounced differently, and if the rule hits, that will be the case, since a different operator (SHORT or LONG) will be inserted. The subtree CORDIALQUEST isn't resolved here for simplicity. This must be a rule system that allows the processing of such chunks as "would you", "will you", "might she", "does he".

## 5.2 Phrases vers. chunks

I still have not answered the question of how to parse sentences if I only write rules for parts of sentences. A simple idea is there to checking each rule at each word of a phrase. As a result, the sentence may be replaced by a multitude of pathes that lead from the starting word in each partial parse to the ending word in it, bracketing or highlighting the part of the phrase that matches a rule. Subsequently, a minimal path method will decide which bracked parts in sequence are used to form the sentence. Need a good example to explain this.

State: Friday 4/30/04 3pm.

## 6 Appendix: Parsing part of bachus' syntax

The following is the core part of bachus' meta grammar, only describing the syntax of the meta-language used for word grammars. I will utilize what I learned here and use it as nodes in a grammar that operates on word sequences.

### Bachus grammar

```
%token <sympt> NAME COMMENTLINE LTRIE TRIE RTRIE PROCESS
                TRIEFILE ENDTRIE MORPH EMPTYNODE
%token <string> FILENAME BOUND OPERATOR
%token <ival>   INTEGER
%left '+'
%%
STATEMENT:  RULE   '\n'
            | STATEMENT RULE '\n'
;
RULE:  /* EMPTY LINE */
      | NAME '=' '>' RIGHTSIDE
      | RIGHTSIDE
```

```

        | DECLARATION
        | COMMENTLINE
;
RIGHTSIDE:  EXPRESSION
           | EXPRESSION '|' RIGHTSIDE
           | '|' RIGHTSIDE
;
EXPRESSION: EXPRESSION '+' EXPRESSION
           | REPEATED
           | NAME
           | MORPH
           | BOUND
           | OPERATOR
           | EMPTYNODE
;
REPEATED:  '(' EXPRESSION ')' ' [' INTEGER ',' INTEGER ']'
           | NAME '[' INTEGER ',' INTEGER ']'
;
DECLARATION:  TRIE NAME '=' FILENAME
             | RTRIE NAME '=' FILENAME
             | LTRIE '.' INTEGER '.' NAME '=' FILENAME
             | PROCESS NAME
             | TRIEFILE '=' FILENAME
             | ENDTRIE
;

```

## 7 Implememtation Methods

As much as possible I want to exploit the existing and working structure of Bachus.

So we have one chunk grammar file. It will be processed based on pretty much the same metagrammar as bachus. The input file for the chunk grammar contains expressions which are made up of works such as “conj”, “noun-form”, “verbform”, etc.

The terminals can be complete word grammars or they can be work-sequence sets. For example, there is grammar for what is a verb. And there is a word-sequence set for all possible conjunctions. A conjunction can be a word as “but”, but also a sequence like “as long as”. (see shallow parser

classes). Maybe should call that a `shallow_class`.

Notes: To maintain the word and string pool idea, have each word grammar be generated separately, but share a string pool: Phonememic strings are inserted into this string pool during the production of `ltrie`'s. In order to maintain just one pool, we will keep the string pool in one file (a huge binary) which is opened, modified and written back, whenever one new word grammar is generated.

The subroutine `morphemize_word` is not directly used anymore. Instead it becomes a call within a new recursive subroutine which is called `Chunkwalker`.

`chunkwalker` is essentially a rewrite of `GtWalker` but operates on classes rather than individual words. While `GtWalker` proceeds using `Ltrie` call, and while it only proceeds by character position, the `Chunkparser` will be able to proceed by whole words or even groups of words. It may proceed by a whole words, in fact by calling `GtWalker`.

Just like we have declarations such as `ltrie` and `rtrie` in a word grammar, we will have declarations such as

`wordgrammar` in a `chunkgrammar`:

`wordgrammar verbinf="verbinfinitive.grm"`

`wordgrammar` is followed just like that by a symbol and a filename. (establishing an association between the two) And the statement means that the file name is opened and a bachus style word grammar is generated.

`chunkclass` in a `chunkgrammar`:

`chunkclass conj ="conj.cls"`

`conj.cls` contains a list of conjunctions.

In complete analogy we need a program that generalizes `builder`.

A `liblookup` what's that?

## grammar.eng

```
chunk => verbform
    | nounform
    | doform - vinf
    | doform - <.H*.*>+not - verbform
    | verbform - <Pause>+ shallowconj - <H>+ verbform
    | shallowconj
    | personal - verbform - article - nounform
    | personal - auxverb - verbform - article - nounform

verbform => vinf
    | gerund
    | doform

vinf => verbs
    | verbsile+"e/"
    | verbsile+"es/s/"

gerund=> verbs+"ing/iG/"
    | verbsile+"ing/iG/"

#
not=> "no/no/" | "not/not/" | "nothing/nozing/"

nounform => noun
    | noun+"s/s/"

trie auxverb="auxverbs.pho"
trie article="articles.eng"
trie personal="personal.pho"
trie verbs="verbs.pho"
trie noun="nouns.pho"
trie doform="doform.pho"
trie shallowconj="shallowconj.pho"
trie verbsile="verbstemsilente.pho"

$eof
```

**shallowconj.pho**

albeit  
all.the.same  
altho  
and.so  
as  
as.a.result.of  
as.if  
as.long.as  
as.well.as  
assuming.that  
because  
but  
but.also

## **8 Recent more raw ideas (Dec 2005)**

Statement of problem:

We are given a large number of sets of words. Each set may contain a relatively large number of words. Natural written language is to be considered here to form sequences of these words and others which may not be in any of the set.

We write down a large sequence grammar which may be expressions containing individual words or entire sets of words. These constitute a number of patterns that we wish to match in the sentences. The problem is, there may be hundreds if not thousands of patterns, and we can't test them all sequentially.

So the question is: Why waste so many questions on each word: Why not collect the information directly word by word?

The form in which the information (sequence grammar statements plus specifications of word sets as well as special word grammars) is top down, but during processing we need bottom up processing. The guiding idea is: The word "KNOWS" to which sets it belongs, the set KNOWS to which sequences it belongs.



Since at the bottom are the words, each word needs to be in a large dictionary which "recalls" all the connections in the upper tiers. This recalled information is basically a list of set numbers. So we see here that is a lexicon in the classic sense.

Furthermore, there are other things at the bottom: words which may not be in the lexicon can be analysed by various word grammars. The set of word grammars contains many branches, and a word may be succesfully parsed by more than one branch

For example, 'feeling' can be a gerund or a noun in a grammar:

```
word -> gerund
      | noun
```

```
gerund -> verbstemm + "ing"
```

```
noun    -> {word ending in "ing"}
```

If the branch "gerund" is one or several sequence patterns of the grammar, then we want that this branch of the grammar KNOWS about all the patterns in which it occures.