ALP – A Latin Parser

Archibald Michiels
amichiels@uliege.be
https://github.com/archibaldmichiels/alp

2025

Table of Contents

Introduction

ALP (**A L**atin **P**arser) is a syntactic parser for a small subset of classical Latin. How small the subset is (or how large, size lying in the beholder's eye), can be guessed through a quick perusal of the test files (included in this document), i.e. the collections of sentences that ALP is able to deal with, i.e. to parse. We do not think that ALP can be described as a toy parser, a mere sketch of what could be done on a larger scale (see for instance Covington 2003). A toy system is seldom extensible without a major revision of the very framework it is based on. That is not the case with ALP, as we will attempt to show.

ALP is a true parser, i.e. it delves into the surface strings looking for deep syntactic structure, which means here predicate-argument pairings, which it delivers according to a canonical order that is fully independent of the linearity of the sentences it deals with.

It is worth stressing that this endeavour amounts to more than simple tagging (the assignment of tags associated in a lexicon with the surface elements) followed by the limited amount of surface structure that can be built on the basis of sequences of tags (a good example of what can be achieved with these limited means is Koster 2005).

The argument structure referred to above is not limited to noun phrases (nps), prepositional phrases (pps) and the like, but includes clauses, which can exhibit the complex structure associated with full independent sentences. In a word, we have to do with true parsing.

Of course, the linearity of discourse will be seen to be preserved in the *wordlist* associated with the surface string, but is also taken into account in assessing the weighting assigned to a given parse, its preference ranking. As a matter of fact, most reasonably complex sentences will receive more than one parse (i.e. will be ambiguous with respect to the grammar embodied in the parser, although they may appear – and be – fully unambiguous as utterances, if we leave aside the body of utterances used as grammatical examples or illustratory material for schoolbooks, even if borrowed from the classical writers, precisely because they are deprived of their context of utterance). The weighting procedure aims at keeping only the most promising parses (or in presenting them first to the user, which amounts to the same). A weighting procedure is an essential part of a parser, even if it is often missing.

Latin exhibits a relatively free word order, and is therefore not likely to be amenable to top-down parsing, which is based on the checking of structural hypotheses derived from the grammar (and specifying a given left-to-right ordering) by confronting their expectations with what is found in the string submitted to analysis.

A better candidate is provided by a bottom-up parser, which looks at what it has under hand (the string with its given word order) and tries to put it together somehow on the basis of the structural descriptions the grammar holds for higher elements (for example, putting together a noun and an adjective sharing gender, number and case to produce a np which will be able to fill in a syntactic slot, let's say subject if the case is nominative - or accusative in an infinitival clause - and number is compatible with that of the VP of which it is supposed to provide the subject).

It is such a bottom-up parser that ALP implements, but in a way that takes full advantage of the 'facilities' offered by Prolog[1], which are in fact the very mechanisms that Prolog is built on, I mean UNIFICATION and BACKTRACKING.

Before moving on to a discussion of those two basic mechanisms, a short and drastically oriented introduction to Prolog as a programming language may be in order[2].

In Prolog we describe a *world*, very often a *micro-world*, the *objects* that populate that world and the *relations* that obtain between them. We do this by means of *facts* and *rules*. In our case the micro-world is a subset of classical Latin, the facts are mainly to be found in the lexicon and the rules in the grammar.

Note that the notion of the distinction between facts and rules running parallel to that between lexicon and grammar is very much a simplification. It can be argued that the description of *multi-word units* that we need for parsing is much nearer to grammar rules than to standard lexical entries. I have attemped to show elsewhere that most multi-word units, unless they are completely frozen, are best captured by rules that look very much like standard grammar rules, except that they contain lexical material that restricts the openness of the purely structural requirements of standard grammar rules (see Michiels 2016a).

Facts in Prolog are like records in a standard data base, but in a much freer format. Rules have conditions of the *if-and-only-if* type, and can lead to the production of new facts, which are then added to the database of facts Prolog is working with and which aims at capturing the basic facts and relations of the world being described.

Once the description of the *world*, its inhabitants and properties, is embodied in a Prolog program, we can submit *queries*, i.e. ask questions. A Prolog query (a question set to the Prolog engine) will be interpreted in the following way: is this provable? If the query contains *variables*, it will involve finding values for the variables that make the query true. So, by the side of very simple queries, which look like queries on a standard data base, we can formulate queries of a much more complex form, asking Prolog to enquire whether the object we propose can be admitted as a new inhabitant of our world (can be *proved* to be such an inhabitant), and what values need to be assigned to the variables in the query to make it so.

In our case (once the parser and the grammatical and lexical resources it draws on have been entered as a Prolog program), we can submit a string to Prolog, and ask whether that string can be read as a Latin sentence (i.e. can the string be *proved* to be a Latin sentence?). We can leave as variables to be instantiated (i.e. given a value) what will turn out to be the parse, i.e. the structural assignments that explain why the string is in fact the embodiment of one or more Latin sentences (there will be more than one *proof* if the string is ambiguous with respect to the grammar and lexicon in use).

To sum up, the variables to be instantiated will be bits of the structural make-up of the sentence, under the structural assignment that made that string a Latin sentence, according to the grammar and lexicon embodied in the Prolog database built as a result of running the program. This data base is partly static (standard lexical entries), but mainly dynamic – applying the rules yields new structures. When Prolog has found a structural description (i.e. a parse) that covers the whole input

1 We use SWI-Prolog, a long-standing high-quality free Prolog. A suitable installation file (i.e. geared towards the OS one is using) can be downloaded from the SWI-Prolog website. See Wielemaker 2003 for an overview.

2 Among the most useful introductions to Prolog is certainly to be ranked *The Art of Prolog* by Leon Sterling and Ehud Shapiro, Second Ed. The MIT Press and https://cliplab.org/~logalg/doc/The_Art_of_Prolog.pdf On Prolog and NLP : Fernando Pereira and Stuart Shieber, *Prolog and Natural-Language Analysis*, Digital Edition, http://www.mtome.com/Publications/PNLA/prolog-digital.pdf

string, it has found one way of making the string a Latin sentence. It is then ready to start all over again, and find the other possible ways of parsing the string that make it a Latin sentence (and thereby dealing with ambiguous sentences, ambiguous with respect to the grammar and lexicon embodied in the Prolog program, which, as we have said, may or may not reflect perceived ambiguity when the sentence is replaced in its discursive context).

Let us not be mean, let's give at least a toy example... Suppose we feed Prolog the following program:

```
noun(regina, agreement(case:nominative,gender:feminine, number:singular)).
noun(reginam,agreement(case:accusative,gender:feminine, number:singular)).
noun(reginae, agreement(case:nominative,gender:feminine, number:plural)).
noun(reginas,agreement(case:accusative,gender:feminine, number:plural)).

noun(fatum, agreement(case:nominative,gender:neuter, number:singular)).
noun(fatum,agreement(case:accusative,gender:neuter, number:singular)).
noun(fata, agreement(case:nominative,gender:neuter, number:plural)).
noun(fata,agreement(case:accusative,gender:neuter, number:plural)).

adjective(clara, agreement(case:nominative,gender:feminine, number:singular)).
adjective(claram,agreement(case:accusative,gender:feminine, number:singular)).
adjective(clarae, agreement(case:nominative,gender:feminine, number:plural)).
adjective(claras,agreement(case:accusative,gender:feminine, number:plural)).

adjective(clarum, agreement(case:nominative,gender:neuter, number:singular)).
adjective(clarum,agreement(case:accusative,gender:neuter, number:singular)).
adjective(clara, agreement(case:nominative,gender:neuter, number:plural)).
adjective(clara,agreement(case:accusative,gender:neuter, number:plural)).
```

The above are *facts*, the first type of Prolog *clause*. They are made up of a *functor* (here, *noun* or *adjective*) with a given *arity*, i.e. number of elements within its domain, known as *arguments* (in this case, 2). In our bundle of facts, the first arg(ument) is *atomic* (to simplify drastically: a word or a number) and the second takes the form of a functor with its own arg(ument)s. The functor is *agreement*, and the args take the form of *features*, i.e. pairs of *feature name: feature value*, both atomic in this case (e.g. *gender*:*neuter*).

We add a *rule* to our program (the second form of Prolog clause). It reads as follows :

```
pair(First, Second,Agreement):-
( (adjective(First,Agreement),noun(Second,Agreement)) ;
  (adjective(Second,Agreement),noun(First,Agreement)) ).
```

A few words of explanation are in order:

pair(First, Second,Agreement):-

% means: we can conclude (regard it as a *fact*)
% that we have a *pair* of two members, *First* and *Second*
% (these are variables, opening with a capital letter as variables do in Prolog),
% exhibiting an agreement triplet referred to as *Agreement.*
% Note that the latter too is a variable, and that the variable name
% gives no information to Prolog as to its contents – *X* or *Y* would have conveyed as much
% information, the program writer being the only one who knows that he means this variable
% to stand for an agreement triplet such as *agreement(case:accusative,gender:neuter, number:plural)*

% *if and only if* (that's the meaning of ':-')

((adjective(First,Agreement),noun(Second,Agreement)) ;
(adjective(Second,Agreement),noun(First,Agreement))).

% we have (i.e. in our data base) a pair adjective-noun,
% in either order (the operator 'OR' is written ';' in Prolog)
% (bracketing being necessary because each branch of the alternative is made up of *two* clauses)

% and the two agreement triplets are UNIFIABLE (here unification boils down to identity – see
% below for a fuller treatment).

Once the program has been fed into the Prolog database, we can enter queries such as

a) (query type: is this true?) e.g. *pair(clara,regina, _).* Answer should be 'true'
pair(fata, clara, _). Answer should be 'true'
pair(reginas, clara, _). Answer should be 'false'

In the above queries, the underline (_) lets Prolog know that we are not interested in the value assigned to the argument, in this case the agreement triplet.

b) (query type: what values should the variables get in order to make this true ?)
e.g. *pair(clara,regina,Accord).*
Accord will be instantiated to *agreement(case:nominative,gender:feminine, number:singular)*.

Try to guess what the following query is likely to yield:
pair(Premier,Second,agreement(case:nominative,_,_)).

The answer being given on the next page, pause a few seconds before moving on.

The answer consists in a series of pairs ranging over the available vocabulary, all in the nominative case. The 'false' at the end of the list means that there are no more answers, as far as Prolog knows (i.e. has been told).

Note that the order in which the answers are given reflects the order Prolog follows in exploring the data base; in fact, it reads the way we do: from left to right and from top to bottom (to get the answers at the terminal we should press the ';' key after each pair).

Premier = clara,
Second = regina ;
Premier = clarae,
Second = reginae ;
Premier = clarum,
Second = fatum ;
Premier = clara,
Second = fata ;
Premier = regina,
Second = clara ;
Premier = reginae,
Second = clarae ;
Premier = fatum,
Second = clarum ;
Premier = fata,
Second = clara ;
false.

UNIFICATION is a simple and powerful mechanism. It accomplishes two things : verifying structural compatibility and retrieving and assigning information. In ALP we make use of a feature-unification algorithm, which turns straight Prolog unification into a less rigid mechanism but is firmly based on standard unification all the same (see Gal et al. 1991).

Feature here is not to be understood as restricted to *atomic binary feature* such as the *singular-plural* pair to capture *number.* A feature as we understand it in ALP has an atomic feature *name* all right (such as *number*) but can take as *value* any structure recognizable by Prolog, i.e. any Prolog *term*. Such structures include *lists* and *trees*, and well-nigh anything the linguist can dream of ever wanting to use.

Let us give a simple example, not related to linguistics or to ALP. Let's build a structure whose functor-name is *suite* and whose argument is a three-element *list*.

Lists are sequences of *elements* enclosed in square brackets. The *elements* can be *atomic*, or themselves be *structures*, *lists*, or any other *Prolog term* (a *term* being anything Prolog recognizes as its own, remember). Examples are:

[albert,bernard,camille,didier, zadig]	(atoms)
[Ten, Nine,One, Two, Three]	(variables)
[first(One,1,Next), [X,Y,z], A,B,C, [zadig], auteur(zadig, voltaire)]	(complex)
[]	(empty list)

Note that lists are explored by means of the operator '|', which divides the list into *Head* and *Tail*.

The *Head* is a list element, or several such elements, separated by commas, as are all list elements; the *Tail* is the remainder of the list, and is itself always a list.

Unification can be used to show '|' at work:

[a,b,C,d,e,f] = [A,B|Queue].

The unification of the two lists succeeds, with the following variable instantiations:
A=a, B=b, Queue=[C,d,e,f].

Examples of our *suite* structure would be

suite([semel(One), bis(Two), ter (Three)]) and
suite([semel(1), F, ter(3)]).

Recall that variables in Prolog open with a capital letter, so that *One*, *Two*, *Three* and *F* are variables. If we use the operator for straight Prolog unification, which we have seen to be nothing else than the equal sign, we can write:

suite([semel(One), bis(Two), ter (Three)]) = suite([semel(1), F, ter(3)])

entering it as a Prolog *query*, i.e. asking Prolog to carry out the unification.

Unification will first check *structural compatibility.* We have two structures here whose functor is *suite* and whose argument is a list, so that they are compatible if the lists themselves are compatible. The *arity* of the lists is the same: 3 (recall that the arity is simply the number of constituents). The two lists will therefore be compatible if each of their elements is compatible. The first is a structure with functor *semel* and a single argument. So far so good, but the arguments must themselves be compatible, i.e. unifiable. This is the case (meaning: they ARE unifiable) because variables (and *One* is a variable) are unifiable with anything, therefore variable *One* is unifiable with the numeric atom found in the corresponding slot, i.e. *1.* The unification succeeds by giving the value *1* to the variable *One* (we say that *One* is *bound* or *instantiated* to *1*). In a similar way, in the second list element, the variable *F* will be bound to the structure *bis(Two),* the variable *Two* continuing *unbound*. In the third list element, *Three* will be bound to *3*. The resulting structure (i.e. the result of the unification process) will be:

suite([semel(1), bis(Two), ter (3)]).

In ALP unification is used to associate morphological variants with lexical items, to retrieve the argument structure of predicates within their entries, to ensure gender, case, number compatibility, and well-nigh everything else. Matching the argument list of a predicate will simply mean going down the list, picking each element and trying to unify it with what we find in the string submitted to analysis. Unification, with the added flexibility of feature unification as implemented in ALP, can deal with the assignment of decorated tree structures to utterances, i.e. parsing. The decoration will be mainly lexical (the words as leaves), but any type of added information (e.g. semantic) can be envisaged, as long as it is computable (for instance, in ALP, the weight assigned to each parse, which leads to the selection of the best parse or parses).

BACKTRACKING plays an important part in the tracking of all the possible solutions to a given problem. Seeing that language (especially with respect to a given grammar) is highly ambiguous, both globally (at sentence level) and locally (within the structure of phrases that will be included at a higher level, where further choices will operate), it is essential for the parser to be able to come up with all the solutions licensed by the grammar it embodies.

Backtracking is the mechanism by which Prolog keeps track of every single choice point in the search tree. Whenever a goal fails, Prolog backtracks to the last choice it made in its attempt to solve the goal, and chooses another branch of the search tree, if there is any that is still unexplored. If all fail, Prolog moves up one step further up the tree, and tries another branch up there. While doing this, it also uninstantiates any variable that got instantiated while Prolog was exploring the branch that led to failure.

In order to find all possible solutions, we can simply store the current solution, and force failure, and thereby force backtracking to occur. As a matter of fact, Prolog itself provides what are known as *second-order predicates* to build a list (including a sorted list) of all solutions. In ALP we use sorting on the weight in order to get the best solutions first, if there are more than one.

One might wonder what kind of profit to expect from a Latin parser. After all, we do not need such a tool as a first step towards machine translation, the texts we are interested in here (classical Latin texts) having been translated and retranslated, commented and over-commented.

The profit we can derive is directly linked to the absence of pressure of any kind. We do not NEED such a parser, so that we can concentrate on what a parser can teach us about language. To be usable as the basis for a parser, a grammar needs a degree of explicitness which forces it to come to grips with a good number of issues that are likely to have been considered irrelevant or to have been relegated to stylistics, i.e. quirks and idiosyncrasies left over to expressive power and the like. A major issue that has to be dealt with is the amount of freedom in word order – what are the limits that need to come to be part of an algorithmic treatment ? How does syntax interact with semantics and pragmatics, in a way that can be shown to improve coverage, i.e. increase the part that can be dealt with algorithmically?

The development of a parser for Latin does not pursue any practical aim. It can be conceived as a contribution to the study of the language, in a spirit of free enquiry, which also means freedom from any pressure that does not directly derive from the subject under scrutiny.

The above considerations militate in favour of a parser whose design keeps grammar and parsing algorithm as separate as possible. We achieve that aim by relying entirely on a *production system* as parsing algorithm. The production system is organized in *passes*. Each pass is implemented as a series of production rules which operate over and over again until they are unable to produce

anything new, in which case they pass control over to the next pass.

The production rules are allowed to build structure on the basis of what is available to them. In the first pass, the lexical pass, the words in the text are paired with the information stored about them in the lexicon (we have seen that this is a matter of straight variable instantiation). When we claimed above that the reading of *fata* as an nominative case would lead to failure, we certainly did not mean that this happened at the lexical look-up stage, where there is absolutely no information available to reject the nominative case or prioritize the accusative. The information is simply stored and made available to the next pass (higher in the structure-building hierarchy).

The grammar passes will likewise proceed from simple structures to more complex ones that need the information provided by the simple ones. Again, all the production rules in a given pass are allowed to produce structure over and over again, until they have nothing to add at their level.

Let's look at a very simple production rule to be found in the first grammar pass, that for building one-word nps such as *rex* in *rex scribit epistulam.*

The comments included in the Prolog program provide basic information about the np building procedures :

```
The core NPS are assembled before the other NPs, for which they can serve as building blocks.
There are indeed two passes for nps: core and finite.
The core NPs are simple nps that do not involve predications, therefore no relatives, no arg-bearing nouns,
just the simple buiding blocks: nouns as nps, names as nps, adj+n as np, and so on...

Each np is associated with an index which refers to the positions it spans in the input string
The index is useful to make sense of gaps, i.e. traces (t or e in syntactic parlance) 'left' by elements
'moved out of place' by 'transformations'. The quotes are meant to show distance with respect to the
syntactic theory underlining such treatment.

But undoubtedly a similar treatment is needed. If the trace cannot be associated with the relative pronoun,
and, via the relative, and more importantly, with the antecedent, all the controls we wish to perform, such as
semantic controls on arg bearers, will prove impossible in relative clauses, to give one example.
```

We then proceed to the production rules for simple nps, and begin with nps consisting of a single noun. We give below the *lex* clause for our word *rex*

```
lex(rex, noun, [pos:noun, txt:rex, lex:rex, case:nom, gender:masc, class:common, number:sing, sem:[hum]]).
```

The relevant production rule is the following:

```
[core,np1] ->
[mapped(noun,[from:A,to:B|FS]),
 constraint([pos:noun,lex:Lex,class:common,sem:Sem,txt:Text,
          number:Nb,gender:G,case:C],FS),
 map(np,[pathlist:[p(A,B)],hp:[p(A,B)],index:i(p(A,B)),distance:[0],
         cat:np,sem:Sem,class:common,lextype:full,
         number:Nb,person:3,gender:G, type:core,lex:Lex,txt:Text,
         case:C,w:1])].
```

Which basically means that if we have in our text (we have it *mapped* by a previous *production*), from position A to position B (remember that capitals are reserved for variables in Prolog – A and B are thus variables), an item that was placed in the noun box and is associated with *feature bundle* FS, then we can use the predicate **constraint** to check or retrieve information from that feature bundle: we need here a common noun, a full lexical item; we retrieve the information contained in the feature bundle regarding *number*, *gender*, *case*, *textual form* and *semantic class*: *sing*, *masc*,

nom(inative), *rex*, *[hum]*. We can then allow the production rule to produce (via the predicate **map**) a record to be put into the np (noun phrase) box.

Such a box will include information about the path covered by the np, the head of such a path, the index in case we need it somewhere down in the structure-building process (for instance if *rex* was to be found to be the antecedent of a relative pronoun), the distance within the path (in case of non-contiguity of the constituents[3]), morphological and lexical information derived from the lexical item. We add person (third person), a type (core np) and a weight (1).

This may seem to be a very heavy procedure just to account for what is dealt with in a couple of rewrite rules in a top-down or bottom-up parser, namely

np->n, n->rex or *rex->n, n->np.*

But in fact all the other information we gather and transport via the production rules based on feature unification will prove to be useful or downright indispensable in any sophisticated parser designed for a nearly free order language such as Latin.

The important design decision is to select a process (such as the one embodied in production rules) that boils down to monotonous incrementation of the available information. This does not prevent us from using specifically designed algorithms for ancillary tasks.

In short, we try to combine a parsing algorithm that is reduced to monotonous structure incrementation through a production system with various procedures that compute the quality rating of the structures licensed by the grammar rules and the information embodied in the lexical items (for instance the argument structure of a predicate, which has predictive power on how the string elements need to be structured into phrases of various levels such as nps and clauses).

A linguist will surely find that there is a huge distance between the type of grammar he is used to writing and the one he is confronted with in ALP. Well, there is a price to pay – an algorithm embodying a grammar cannot be a grammar written without an idea of how it is to be used in parsing. What we can attempt to do is to make the grammar as *declarative* as possible, i.e. as independent as we can make it from issues of control, of how it is to be used, in what order its rules are to be applied, where structures are to be stored, what should be done in case of failure, and the like. As soon as we attempt to go beyond toy systems, we have to dirty our hands a little and think about issues like the degree of freedom there really is in an 'order-free' language such as Latin, and what to do to assess the quality of the parses delivered by the system. The profit we will draw from such an effort is that we will increase explicitness, and have a much better idea of coverage, i.e. how much of the language can be captured by our rules.

Finally, is there nothing to be said *against* ALP as a parser for Latin? Well, there is one negative point, which, if we were parsing anything else but a dead language, would be rather devastating. The production system sketched here is *inefficient*[4] – its very monotony (it does one single thing:

3 In the pretty-printed parses, we give the value of the 'distance' feature if it is greater than zero. It records the number of words that separate elements that are expected to be contiguous. For instance, in the parse of Martial's line *Aestivo serves ubi piscem tempore quaeris?*, *aestivo tempore* gets assigned the function of adjunct of time with a discontiguity value (distance) of 3, on account of the three words (*serves ubi piscem*) that separate the two constituents of the adjunct, *tempore* and its adjective *aestivo*. This information is likely to be of use in word order and stylistic research.

4 It would seem that at present a fifteen word limit should be imposed on strings to be parsed, unless time does not matter AT ALL, which is seldom the case...

increase available information) is at that price.

We should bear in mind the reasons for which one may want to produce a Latin parser – I can see two main reasons only: teaching and research. The quicker the better shouldn't be our motto, surely.

It's time to give a further example of feature unification from the real world, i.e. in our case parsing. We will be looking at ALP at work, and so gain a feeling of sympathy for a system willing to go through so many tricks just to parse a simple sentence.

ALP gets going parsing the simplest (?) ambiguous sentence

Let's see how ALP goes about parsing *Rex venit.*

The sentence in ASCII format (*Rex venit.*) is submitted first to a homemade version of the Prolog linereader, which turns it into the string *rex uenit*, taking care of decapitalization and changing *v* (*venit*) into standard *u* (*uenit*) and removing the final dot :

rex uenit

The WordList maker takes the two words to produce the list

[0/rex,1/uenit,endpos(2)]

at the same time storing the first two list elements into a storage space named *pos* (*pos* stands for positions) with the help of the predefined *recorda* predicate :

recorda(pos,position(Pos,Posand1,NewHead),_),

thus storing in *pos* both *position(0,1,rex)* and *position(1,2,uenit)*

Running the *makelex.pl* program will have given rise to the following *lex* clauses of interest for our little sentence (they are housed in *vocfile*) :

lex(rex, noun, [pos:noun, txt:rex, lex:rex, case:nom, gender:masc, class:common, ab:no, number:sing, sem:[hum]]).

l*ex(uenit, v, [pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:perfect, kind:std, mood:indicative, number:sing, person:3]).*

lex(uenit, v, [pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:present, kind:std, mood:indicative, number:sing, person:3]).

Note that there are two *lex* clauses for the word *uenit*, which is the source of the ambiguity : *uenit* is either *present* or *perfect* tense.

ALP proceeds by allowing different packets of rules to fire according to the stage it has reached in building the parse. The firing of a rule packet is guided by the following clause :

```
run(Pass) :-  [Pass,_] ---> Condition_Action,
        testexec(Condition_Action),
        run(Pass).
```

It should be noted that ALP grammar rules are built according to the format specified above, namely :

[Pass,_] ---> Condition_Action

where *Condition_Action* is a list of Prolog goals. For instance, in the rule which we are going to use in a moment to take care of our two words, *rex* and *uenit*, namely :

```
[lex,words] --->[recorded(pos,position(A,B,Word),_),
                 lex(Word,Box,FS),
                 map(Box,[from:A,to:B|FS])].
```

The *Pass* variable is instantiated to *lex*, and the *Condition_Action* list is instantiated to

```
 [recorded(pos,position(A,B,Word),_),
                 lex(Word,Box,FS),
                 map(Box,[from:A,to:B|FS])].
```

This list is made up of three Prolog goals, which ALP will attempt to satisfy : the *recorded* goal will be satisfied if there is a record in the *pos* box which can be matched with the *position* structure, instantiating the variables *A*, *B* and *Word* ; the *lex* goal will then look for a match leading to the instantiation of *FS*, using the value of the variable *Word* which results from the instantiation just performed by *position(A,B,Word).* The *map* goal will then be attempted with the instantiated variables. Such a goal list will be satisfied as many times as new information gets recorded by the predicate *map*, which is defined as follows :

```
map(Box,FS) :- not(recorded(Box,FS,_)), recorda(Box,FS,_).
    % the not recorded is essential for keeping track of a 'no new solution' situation
            % it enables the various runs to come to an end
```
– see below

In other words, *map* will fail when it has no new information to store.

The *Condition_Action* list is submitted to the *testexec* goal, which simply takes each element of the list in turn and executes it as a Prolog goal :

```
testexec([]).
testexec([First|Rest]) :-
                call(First),
                testexec(Rest).
```

In order to execute something as a Prolog goal, we simply pass it on as the argument of the predefined *call* predicate.

Note that the definition of the *run* predicate given above ends on a recursive call to itself. This ensures that all alternative ways of satisfying the goals executed by *testexec* are attempted ; but if we want the process to come to an end some time, we must ensure that the goal be allowed to fail. The failure is ensured by the condition on *map* : it cannot map what has already been mapped, so it will fail when it has nothing new to store.

Our *runs* begin with the lexical run, i.e. by running the *words* rule in the *lex* run :

```
runs :- run(lex).

[lex,words] --->[recorded(pos,position(A,B,Word),_),
                 lex(Word,Box,FS),
                 map(Box,[from:A,to:B|FS])].
```

A record (a feature bundle) is added to the data base in the box corresponding to the lexical item's POS (Part of Speech). The start and end positions of the item in the string are recorded in the first

two features; the remaining features are read off the lexicon.

We have, in the *pos* box, the records

position(0,1,rex) and *position(1,2,uenit)*

and we have the above three *lex* clauses for *rex* and *uenit.*

The two words are *rex* and *uenit*, the two boxes are *noun* and *v* and the three *FS* (Feature Sets) are :

[pos:noun, txt:rex, lex:rex, case:nom, gender:masc, class:common, ab:no, number:sing, sem: [hum]]

for *rex*

and

[pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:perfect, kind:std, mood:indicative, number:sing, person:3]
and
[pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:present, kind:std, mood:indicative, number:sing, person:3]

for *uenit*

The *words* rule will fire repeatedly, and unification will ensure the 'copy' of the information.
The *map* rule (which is simply *recorda*, i.e. a storing procedure) will fire with the following unifications for *map(Box,[from:A,to:B|FS]) :*

map(n,[from:0,to:1, pos:noun, txt:rex, lex:rex, case:nom, gender:masc, class:common, ab:no, number:sing, sem:[hum]])

map(v,[from:1 , to:2, pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:perfect, kind:std, mood:indicative, number:sing, person:3])

map(v,[from:1 , to:2, pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:present, kind:std, mood:indicative, number:sing, person:3])

Remember that | is simply the list separator, and does not appear when the list is rebuilt.

We now have our two words recorded under their POS (Part of Speech).

uenit, being a *v* (verb) will fire a rule for building the verb group, firing which will happen when running the set of rules marked as *verb ,* whose execution follows on that of *lex.*

runs :- run(verb).

```
[verb,vg1] --->
[ mapped(v,[from:B, to:C|FS1]),
  ifthen(constraint([gender:Gender],FS1),G=Gender), % we have a gender, we record it
                                                    % if we don't, we leave the G var a free var
                                                   % this prevents failure in the case of a gender check
   map(vgpos,[cat:vg,pathlist:[p(B,C)],hp:[p(B,C)], gender:G,w:0|FS1])].
```

Note that using *B* and *C* as variable names instead of whatever else (*A* and *B*, for instance) has no bearing whatsoever on the instantiations : the important thing is that they are variables ; the match with the information mapped by the *words* rule will be as follows :

B will be instantiated to 1 and *C* to 2 and *FS1* to, in turn,

*[pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:**perfect**, kind:std, mood:indicative, number:sing, person:3]*

and

*[pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:**present**, kind:std, mood:indicative, number:sing, person:3]*

Since we don't have a gender feature in *FS1*, the final mapping will be :

*map(vgpos,[cat:vg,pathlist:[p(1,2)],hp:[p(1,2)], gender:G,w:0, pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:**perfect**, kind:std, mood:indicative, number:sing, person:3])]*.

and

*map(vgpos,[cat:vg,pathlist:[p(1,2)],hp:[p(1,2)], gender:G,w:0, pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:**present**, kind:std, mood:indicative, number:sing, person:3])]*.

These mappings will go into the *vgpos* box, to be used by further ALP rules.

Rex, being a noun, will fire the simplest rule *np1* for noun phrase building,

```
runs :- run(core).  % core nps

[core,np1] --->
[mapped(noun,[from:A,to:B|FS]),
 constraint([lex:Lex,class:common,sem:Sem,txt:Text,
          number:Nb,gender:G,case:C],FS),

ifthenelse(constraint([origin:adj],FS),
        ( W is (0.5), Origin=adj),
        ( W is 1,     Origin=noun)
         ),       % nouns out of adjs have lighter weight

 map(np,[pathlist:[p(A,B)],hp:[p(A,B)],index:i(p(A,B)),distance:[0],
        cat:np,sem:Sem,class:common,lextype:full,origin:Origin,
        number:Nb,person:3,gender:G, type:core,lex:Lex,txt:Text,
        case:C,w:W,constituent_structure:Text])].
```

Recall that mapping the noun *rex* has led to A and B being instantiated to 0 and 1 and FS to

```
[pos:noun, txt:rex, lex:rex, case:nom, gender:masc, class:common, ab:no, number:sing, sem:[hum]]
```

The *constraint* predicate will lead to the following instantiations :
lex : rex, *class:common* (this is a check, since the variable is instantiated in the first arg of the *constraint* predicate), *sem :[hum], txt:rex, number:sing, gender:masc, case:nom.*

Since there is no *origin* feature in FS, *W* (the weight) is set to 1 and *Origin* is instantiated to *noun.* The resulting mapping is thus the following :

```
map(np,[pathlist:[p(0,1)],hp:[p(0,1)],index:i(p(0,1)),distance:[0],
        cat:np,sem:[hum],class:common,lextype:full,origin:noun,
        number:sing,person:3,gender:masc, type:core,lex:rex,txt:rex,
        case:nom,w:1,constituent_structure:rex])
```

The mapping is stored in the *np* box, to be used by further ALP rules.

We now come to the building of the whole predication (the whole clause, in this case the whole S as well, the whole sentence). Here (i.e. in this brief presentation) we have to drastically simplify the rule ALP is using, as most of the code does not concern such a simple clause as that embodied in *rex uenit*.

```
[finite,predbase] --->
[
  mapped(vg,FSverb),
  constraint([number:Nsubj,
              person:Psubj,
              type:finite,
              voice:Voice,
              mood:Mood,
              tense:Tense,
              pathlist:PathlistVerb,
              lex:Clex,
              w:WVerb],FSverb),

 lexarg(Clex,arglist:ArgList),
  pick(ws(Lex,_,clause:Clause_Constraints,mwuw:MW,args:Args), ArgList,_),

.....



 match_list(Args,TreeArgs,PathlistArgs,DistanceArgs,
              sujet([number:Nsubj,gender:_,person:Psubj]),
              finite,
              gap:Gap,
              w:Weight,
              Int,
              PathlistVerb),

.....



map(pred,[cat:pred,
              type:finite,
              pathlist:NSortedF,
              distance:[Distance],
              illocutionary_force:Force,
              class:m,
              number:sing,                    % nber, gender, and person of the CLAUSE, not its subject
              person:3,vgperson:Psubj, % needed in imperative clauses of the first and second persons
              gender:neuter,
              mood:Mood,
              tense:Tense,
              polarity:Pol,
              argbound:no,
              gap:Gap,
              w:Wtot,
              add:Add,
              checkint:Interrogative,
              flagint:Flagint,
              constituent_structure:[illocutionary_force:Force,vg:FSverbfull|ST]])]. % ST is the sorted list of arg fillers
```

Let's look at the top third :

```
  mapped(vg,FSverb),
  constraint([number:Nsubj,
          person:Psubj,
          type:finite,
          voice:Voice,
          mood:Mood,
          tense:Tense,
          pathlist:PathlistVerb,
          lex:Clex,
          w:WVerb],FSverb),

 lexarg(Clex,arglist:ArgList),
  pick(ws(Lex,_,clause:Clause_Constraints,mwuw:MW,args:Args), ArgList,_),
```

We have two mapped *FSverb* we can retrieve, and we'll take the first here (the second will be used in a similar fashion when ALP looks for further instantiations) :

*map(vgpos,[cat:vg,pathlist:[p(1,2)],hp:[p(1,2)], gender:G,w:0, pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:**perfect**, kind:std, mood:indicative, number:sing, person:3])].*

The instantiation process carried out by *constraint* will yield the following :

Nsubj = sing, Psubj = 3, Voice = act, Mood = indicative, Tense = perfect,
PathlistVerb = [p(1,2)], Clex= uenire, Wverb = 0.

We now turn to the *lexarg* goal ; it will fire with the instantiation for *Clex* which we have just retrieved :

lexarg(uenire, arglist:ArgList)

The match for such a goal is to be found in *template.pl*, which ALP reads in before starting its dialogue with the user. *template.pl* includes the following *lexarg* clause :

```
lexarg(uenire,
    arglist:[ws(uenio_come,intr,clause:[],mwuw:0,
          args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
              prep_cplt:[type:pp,oblig:no,constraints:[prep:or([ad,in]), case:acc]],
              cplt:[type:np,oblig:no,constraints:[case:acc,sem:[city]]],
              cplt:[type:pred,oblig:no,constraints:[type:nonfinite_i, mood:supine]] ])]).
```

which leads to the instantiation of variable *ArgList* to :

```
[ws(uenio_come,intr,clause:[],mwuw:0,
          args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
              prep_cplt:[type:pp,oblig:no,constraints:[prep:or([ad,in]), case:acc]],
              cplt:[type:np,oblig:no,constraints:[case:acc,sem:[city]]],
              cplt:[type:pred,oblig:no,constraints:[type:nonfinite_i, mood:supine]] ])]
```

Attention should be drawn to the *oblig* feature, which will prove to be important in the parsing of *rex uenit*, because the only arg in the arglist which will turn out instantiated is the subject, the only arg with *yes* as value for the *oblig* feature, the other args being all optional (*oblig:no*) and left uninstantiated in *rex uenit*.

The next goal is a call to the *pick* predicate :

```
pick(ws(Lex,_,clause:Clause_Constraints,mwuw:MW,args:Args), ArgList,_),
```

which selects an arbitrary element in a list (and ends up selecting them all in turn when repeatedly reattempted) :

```
% pick : non-deterministic selection in a list

pick(A,[A|B],B).
pick(A,[B|C],[B|D]) :- pick(A,C,D).
```

Pick selects either the head of the list (*A* in *[A|B]*), returning the remainder of the list (*B*) in its third arg (first clause in the definition of *pick*) or selects an element in the remainder of the list (*C*) (second defining clause).

If the second clause succeeds, the list returned as third argument is made up of the first element of the list passed as second argument (*B*, left untouched) and of the tail of the list, to wit *D*, i.e. what remains of *C* when *A* has been *picked* out of it.

In our case, *pick* has a single element that it can pick, the head of the list, the tail being the empty list (there is a single *ws* structure in *ArgList*). The instantiations will therefore be :

```
Lex =  uenio_come,
Clause_Constraints = [],
MW = 0,
Args = [subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                prep_cplt:[type:pp,oblig:no,constraints:[prep:or([ad,in]), case:acc]],
                cplt:[type:np,oblig:no,constraints:[case:acc,sem:[city]]],
                cplt:[type:pred,oblig:no,constraints:[type:nonfinite_i, mood:supine]] ]
```

Our next goal belongs to the second block of the *predbase* rule, namely :

```
match_list(Args,TreeArgs,PathlistArgs,DistanceArgs,
          sujet([number:Nsubj,gender:_,person:Psubj]),
          finite,
          gap:Gap,
          w:Weight,
          Int,
          PathlistVerb),
```

a call to the *match_list* predicate with a number of arguments already instantiated, and others to be instantiated by executing the *match_list* goal.

The instantiated arguments (instantiations carried out by the predicate *constraint* – see above) are the following :

Nsubj = sing, Psubj = 3, PathlistVerb = [p(1,2)]

and of course the instantiation of *Args* which we have just seen :

```
Args = [subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                prep_cplt:[type:pp,oblig:no,constraints:[prep:or([ad,in]), case:acc]],
                cplt:[type:np,oblig:no,constraints:[case:acc,sem:[city]]],
                cplt:[type:pred,oblig:no,constraints:[type:nonfinite_i, mood:supine]] ].
```

The call to *match_list* will therefore be :

```
match_list([subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
            prep_cplt:[type:pp,oblig:no,constraints:[prep:or([ad,in]), case:acc]],
            cplt:[type:np,oblig:no,constraints:[case:acc,sem:[city]]],
            cplt:[type:pred,oblig:no,constraints:[type:nonfinite_i, mood:supine]] ],
           TreeArgs,
           PathlistArgs,
           DistanceArgs,
           sujet([number:sing,gender:_,person:3]),
           finite,
           gap:Gap,
           w:Weight,
           Int,
           [p(1,2)]),
```

We can now look at the *match_list* clause definition :

```
match_list([Function:Specs|RemainingArgs],
        [ArgTree|ArgTrees],
        [PathlistArg|PathlistArgs],
        [DistanceArg|DistanceArgs],
        sujet(SConstraints),
        ClauseType,
        gap:Gap,
        w:BW,
        Int,
        PathlistVerb) :-

  match(Function:Specs,ArgTree,SConstraints,PathlistArg,DistanceArg,ClauseType,gap:Gap1,w:W1,Int,PathlistVerb),
  match_list(RemainingArgs,ArgTrees,PathlistArgs,DistanceArgs,sujet(SConstraints),
         ClauseType,gap:Gap2,w:W,Int,PathlistVerb),
                                                    % match all the other args on the arglist

  % BW is W+W1,                    % sum the weights
  myplus(W,W1,BW),
```

Match_list performs a call to *match* and then a recursive call to itself. Note that in the definition of *match_list* the first argument is presented as a list whose head has been isolated by unification :

```
[Function:Specs|RemainingArgs] =

[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
            prep_cplt:[type:pp,oblig:no,constraints:[prep:or([ad,in]), case:acc]],
            cplt:[type:np,oblig:no,constraints:[case:acc,sem:[city]]],
            cplt:[type:pred,oblig:no,constraints:[type:nonfinite_i, mood:supine]] ]
```

will yield

```
Function : Specs = subject:[type:np,oblig:yes,constraints:[sem:[hum]]]
```

and `RemainingArgs = [prep_cplt:[type:pp,oblig:no,constraints:[prep:or([ad,in]), case:acc]],`
`cplt:[type:np,oblig:no,constraints:[case:acc,sem:[city]]],`
`cplt:[type:pred,oblig:no,constraints:[type:nonfinite_i, mood:supine]]`
(remember that the tail of a list is always a list)

so that the call to *match* will be :

```
match(subject:[type:np, oblig:yes, constraints:[sem:[hum]]] ,
                ArgTree,
                sujet([number:sing,gender:_,person:3]),
                PathlistArg,
                DistanceArg,
                ClauseType,
                gap:Gap1,
                w:W1,
                Int,
                 [p(1,2)]).
```

There are a number of definitions for the *match* predicate. The one for a subject of a finite clause is the one that will be triggered here, on account of the first arg passed to *match*, which specifies that it is interested in the subject arg. I give it here in a simplified version :

```
match(subject:Specs,
      subject:OutSpecs,
      [number:Nsubj,gender:Gendersubj,person:Psubj], % information on the subject taken over from the verb
      PathlistArg,
      DistanceArg,
      finite,          % finiteness implies that subject is in the nominative case
      gap:[],
      w:W,
      Int,
      PathlistVerb ) :-

constraint([type:T],Specs),                    % looking at the nature of the required filler : pred, np, etc.
constraint([constraints:Cs],Specs),            % fishing out the constraints on the arg

mapped(T,FSSpecs),                             % check that we have the type of phrase needed for the arg

constraint([number:NumberinSpecs,gender:GenderinSpecs,case:nom,person:Psubj],FSSpecs),
                                  % retrieving gender and number from the proposed subject and requiring nominative

constraint(Cs,OutSpecs),                                          % applying all other relevant constraints
constraint([pathlist:PathlistArg, distance:DistanceArg,w:W],OutSpecs).   % adding to OutSpecs
```

Unification yields : *Specs = [type:np, oblig:yes, constraints:[sem:[hum]]]*

The first call to *constraint* will instantiate *T* to *np,* the second call *Cs* to *[sem:[hum]]*. ALP will call on *mapped* to find what is stored in the *T* , i.e. *np,* box (where *rex* is stored) and will be able to instantiate *FSSpecs* to

```
[pathlist:[p(0,1)],hp:[p(0,1)],index:i(p(0,1)),distance:[0],
      cat:np,sem:[hum],class:common,lextype:full,origin:noun,
      number:sing,person:3,gender:masc, type:core,lex:rex,txt:rex,
      case:nom,w:1,constituent_structure:rex]
```

The call to *constraint* that follows will unify *NumberinSpecs* to *sing, GenderinSpecs* to *masc* and *Psubj* to *3*.

The *Cs* constraints boil down to *[sem:[hum]]*, as we have just seen. The next call to *constraint* projects it into the variable *OutSpecs*, and the last call to *constraint* adds the triplet *[pathlist:PathlistArg, distance:DistanceArg,w:W],* so that *OutSpecs* is unified to *[sem:[hum],pathlist:PathlistArg, distance:DistanceArg,w:W]*.

But note that the various features constitutive of *FSSpecs* have all been instantiated in the call to

mapped, and are therefore available : *PathlistArg* is instantiated to *[p(0,1)]*, *DistanceArg* to *[0]* and *W* to *1*.
So *OutSpecs* is now instantiated to : *[sem:[hum],pathlist:[p(0,1)], distance:[0],w:1].* This list of features is passed on to the caller of *match*, i.e. *match_list*, in the pair *subject:OutSpecs* as the value of the variable *ArgTree :*

subject :[sem:[hum],pathlist:[p(0,1)], distance:[0],w:1]

ALP will use this information in buiding the resulting parse tree.

In the meantime, *match_list* will go on applying to the remaining args specified in the arglist of *uenire*, namely :

```
prep_cplt:[type:pp,oblig:no,constraints:[prep:or([ad,in]), case:acc]],
 cplt:[type:np,oblig:no,constraints:[case:acc,sem:[city]]],
 cplt:[type:pred,oblig:no,constraints:[type:nonfinite_i, mood:supine]]
```

We have already pointed out that they are all optional (*oblig*:*no*) and can therefore be satisfied without being instantiated at all :

```
% all arguments consumed or only optional args left unsatisfied

match_list([_:Specs|RemainingArgs],
        ArgTrees,
        PathlistArgs,
        DistanceArgs,
        sujet(SConstraints),
        ClauseType,
        gap:Gap,
        w:W,
        Int,PathlistVerb) :-
   constraint([oblig:no], Specs),
   match_list(RemainingArgs,
      ArgTrees,
      PathlistArgs,
      DistanceArgs,
      sujet(SConstraints),
      ClauseType,
      gap:Gap,
      w:W,
      Int,
      PathlistVerb).
```

match_list applies to each argument in turn, each time reducing the length of the arg list it still has to deal with. When the list is empty, the following clause will apply :

```
match_list([],[],[],[0],_,_,_,w:0,_,_).
```

match_list is satisfied : nothing to match, nothing to return as instantiation.

We come to our final block in the *predbase* rule, the mapping of the whole clause :

```
map(pred,[cat:pred,
        type:finite,
        pathlist:NSortedF,
        distance:[Distance],
        illocutionary_force:Force,
        class:m,
        number:sing,                % nber, gender, and person of the CLAUSE, not its subject
        person:3,vgperson:Psubj, % needed in imperative clauses of the first and second persons
        gender:neuter,
        mood:Mood,
        tense:Tense,
        polarity:Pol,
        argbound:no,
        gap:Gap,
        w:Wtot,
        add:Add,
        checkint:Interrogative,
        flagint:Flagint,
        constituent_structure:[illocutionary_force:Force,vg:FSverbfull|ST]])]. % ST is the sorted list of arg fillers
```

Lots of explanations and a good deal of code would be necessary to explain how all the features come to be instantiated. It is the value of the *constituent_structure* feature that will get printed as raw parse and then as pretty-printed parse by ALP. The arg we have been looking at, the subject arg filled by *rex,* will be the only element constitutive of *ST*, the (here vacuously) sorted list of args.

The two parses produced by ALP for *rex uenit* are given below. The cpu time is that taken by the whole parsing procedure. The parses produced have equal weight (1). The only difference is in the tense assigned to *uenit* : *present* or *perfect*

```
[0/rex,1/uenit,endpos(2)]
cputime : 0.015625

1--->
[illocutionary_force:statement,

vg:[selected_reading:uenio_come, polarity:pos, cat:vg, pathlist:[p(1,2)], hp[p(1,2)],
gender:_36098, w:0, pos:v, class:intr, type:finite, lex:uenire, voice:act, txt:uenit, tense:present,
kind:std, mood:indicative, number:sing, person:3],

subject:[number:sing, gender:masc, pathlist:[p(0,1)],hp:[p(0,1)],index:i(p(0,1)), distance[0],
cat:np, sem:[hum], class:common, lextype:full, origin:noun, person:3, type:core, lex:rex, txt:rex,
case:nom, w:1, constituent_structure:rex]]

   illocutionary_force:statement
   vg
      selected_reading:uenio_come
      polarity:pos
      cat:vg
      * uenit *
      pos:v
      lex:uenire
      voice:act
      tense:present
      mood:indicative
      number:sing
      person:3
```

subject
number:sing
gender:masc
** rex **
index:i(p(0,1))
cat:np
sem:[hum]
person:3
lex:rex
case:nom
constituent_structure
rex

1--->
....

illocutionary_force:statement
vg
selected_reading:uenio_come
polarity:pos
cat:vg
** uenit **
pos:v
lex:uenire
voice:act
*tense:***perfect**
mood:indicative
number:sing
person:3
subject
number:sing
gender:masc
** rex **
index:i(p(0,1))
cat:np
sem:[hum]
person:3
lex:rex
case:nom
constituent_structure
rex

I hope the reader will refrain from exclaiming : *Tout ça pour ça !* Reader, that is what true parsing entails ...

Lexical look-up

We come back to the *lexical look-up* process, which associates information to be found in the lexicon with the words encountered in the text:

```
[lex,words] ->[recorded(pos,position(A,B,Word),_),
               lex(Word,Box,FS),
               map(Box,[from:A,to:B|FS])].
```

In this unusual construction, where left of the arrow (→) we find information that does not lead to any action, we accordingly need to concentrate our attention on what follows the arrow.

Structurally, it is a list. The elements it contains (three) are actions to be performed, i.e. calls to Prolog predicates defined somewhere in the program (facts or rules).

The first such call (to the *recorded* clause) specifies that a feature (the second argument of the *recorded* clause) must have been stored in the data base in the box specified as first argument (*pos* here, a *box* being a named bundle of records), corresponding, unsuprisingly, to the lexical item's *pos* (position).

The feature in question (`position(A,B,Word)`) is a three-arg structure: the start and end positions of the item in the sentence are recorded in the first two arguments, the variables *A* and *B* pointing to the beginning and end positions of the word in the wordlist corresponding to the sentence. The *Word* variable refers to the word found between these two positions in the wordlist representing the string.

For instance, if the sentence is '*Habent sua fata libelli.*' (the first example sentence in our test file, credits to Terentianus Maurus), the corresponding wordlist will be *[habent,sua,fata,libelli]* and *fata*, for instance, will be found to occupy the third position, i.e. from 2 to 3 (the count beginning at 0). As a matter of fact, the production of the wordlist corresponding to the input string will already have inserted the positions in the resulting wordlist (as well as specifying the end position by means of the *endpos* feature):
[0/habent,1/sua,2/fata,3/libelli,endpos(4)]

In this instance, the positions of the word and the morphological variant will have been stored in a record belonging to the *pos* box ('position' records), yielding here, by instantiation:
position(2,3,fata) (variable A being instantiated to 2, B to 3, and Word to *fata*).

We then retrieve information on *fata* from the lexicon (`lex(Word,Box,FS)` (i.e. the set of *lex* clauses). The lookup (carried out by straight Prolog unification) yields two such Prolog clauses, one for nominative and one for accusative:

```
lex(fata, noun, [pos:noun, txt:fata, lex:fatum, case:nom, gender:neuter, class:common, number:pl, sem:[abstract]]).
lex(fata, noun, [pos:noun, txt:fata, lex:fatum, case:acc, gender:neuter, class:common, number:pl, sem:[abstract]]).
```

the pattern being *lex(Textual_form,Record_Box_in_db,Feature_List)*

The lex clauses have arity 3, i.e. three arguments: the first is the morphological variant itself (*fata*), the second the POS (Part of Speech, this time) and the third is a list of features, with information on part of speech, text form, lexeme, case, gender, number, class, and a list of semantic features, a single one in this case, namely 'abstract'.

The three arguments will instantiate the variables Word, Box and FS:

Word = *fata*, Box = noun, and FS will be instantiated to the first feature list (that for *fata* as nominative: [pos:noun, txt:fata, lex:fatum, case:nom, gender:neuter, class:common, number:pl, sem:[abstract]])

Note that such an assignment will eventually lead to failure (its fate...), *fata* being an accusative in our sentence; as will be explained in the next section, backtracking will occur, and the second feature list will come to bind the variable FS.

We then record the information (ending the look-up procedure) in a box (i.e. a collection of records) whose name is that of the POS, i.e. in this case 'noun' : map(Box,[from:A,to:B|FS]).

The feature bundle FS ([pos:noun, txt:fata, lex:fatum, case:nom, gender:neuter, class:common, number:pl, sem:[abstract]]) will come to be included in a list whose first two elements will be the positions within which the item was found in the user's text. Variable binding will therefore yield:

map(noun, [from:2, to:3, [pos:noun, txt:fata, lex:fatum, case:nom, gender:neuter, class:common, number:pl, sem:[abstract]]]

The thing is obvious, but worth bearing in mind : lexical lookup has no prescience as to what the parser will retain and build into a parse and what it will reject as unusable for the parsing of a given sentence. Consider the very simple sentence *hunc librum legit*. Lexical lookup will pull out not three but five lex clauses, namely :

lex(hunc, adj, [pos:adj, txt:hunc, lex:hic, gender:masc, type:tool, case:acc, number:sing]).
lex(hunc, prdem, [pos:prdem, txt:hunc, lex:prdemaccmascsing, gender:masc, person:3, sem:_, case:acc, number:sing, function:[object, prep_cplt, subject]]).

lex(librum, noun, [pos:noun, txt:librum, lex:liber, case:acc, gender:masc, class:common, ab:no, number:sing, sem: [thing, abstract]]).

lex(legit, v, [pos:v, class:tr_cod, type:finite, lex:legere, voice:act, txt:legit, tense:perfect, kind:std, mood:indicative, number:sing, person:3]).
lex(legit, v, [pos:v, class:tr_cod, type:finite, lex:legere, voice:act, txt:legit, tense:present, kind:std, mood:indicative, number:sing, person:3]).

Only one *hunc* will survive, the one where *hunc* is assigned *adj* as part of speech, because the *prdem* (demonstrative pronoun) reading will prove to be unusable in parsing the sentence. The two lex clauses for *legit* will both be retained, giving rise to two parses for the sentence, which will therefore be recognized as ambiguous on account of the double tense assignment (present vs perfect).

If the template for *legere* (templates are responsible for argument assignment, more on that below) had offered the possibility of assigning both a direct object and a complement of the object, we would have had a parse assigning the object function to the pronoun *hunc* and the function of complement of the object to *librum,* giving rise to an interpretation along the lines of *he reads/read this one as a book* (as if it were a book?). But the template for the sense of *legere* we are interested in here (the *legere* meaning *read* not *choose*) is simply :

lexarg(legere,
arglist:[ws(lego_read,tr_cod,clause:[],mwuw:0,
args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
object:[type:np,oblig:no,constraints:[case:acc]]])]).

In ALP unification is thus used to associate morphological variants with lexical items, to retrieve the argument structure of predicates within their entries, to ensure gender, case, number compatibility, and well-nigh everything else. Matching the argument list of a predicate will simply mean going down the list, picking each element and trying to unify it with what we find in the string submitted to analysis. Unification, with the added flexibility of feature unification as implemented in ALP, can deal with the assignment of decorated tree structures to utterances, i.e. parsing. The decoration will be mainly lexical (the words as leaves), but any type of added information (e.g. semantic) can be envisaged, as long as it is computable (for instance, in ALP, the weight assigned to each parse, which leads to the selection of the best parse or parses).

BACKTRACKING plays an important part in the tracking of all the possible solutions to a given problem. Seeing that language (especially with respect to a given grammar) is highly ambiguous, both globally (at sentence level) and locally (within the structure of phrases that will be included at a higher level, where further choices will operate), it is essential for the parser to be able to come up with all the solutions licensed by the grammar it embodies.

Backtracking is the mechanism by which Prolog keeps track of every single choice point in the search tree. Whenever a goal fails, Prolog backtracks to the last choice it made in its attempt to solve the goal, and chooses another branch of the search tree, if there is any that is still unexplored. If all fail, Prolog moves up one step further up the tree, and tries another branch up there. While doing this, it also uninstantiates any variable that got instantiated while Prolog was exploring the branch that led to failure.

In order to find all possible solutions, we can simply store the current solution, and force failure, and thereby force backtracking to occur. As a matter of fact, Prolog itself provides what are known as *second-order predicates* to build a list (including a sorted list) of all solutions. In ALP we use sorting on the weight in order to get the best solutions first, if there are more than one.

One might wonder what kind of profit to expect from a Latin parser. After all, we do not need such a tool as a first step towards machine translation, the texts we are interested in here (classical Latin texts) having been translated and retranslated, commented and over-commented.

The profit we can derive is directly linked to the absence of pressure of any kind. We do not NEED such a parser, so that we can concentrate on what a parser can teach us about language. To be usable as the basis for a parser, a grammar needs a degree of explicitness which forces it to come to grips with a good number of issues that are likely to have been considered irrelevant or to have been relegated to stylistics, i.e. quirks and idiosyncrasies left over to expressive power and the like. A major issue that has to be dealt with is the amount of freedom in word order – what are the limits that need to come to be part of an algorithmic treatment ? How does syntax interact with semantics and pragmatics, in a way that can be shown to improve coverage, i.e. increase the part that can be dealt with algorithmically?

The development of a parser for Latin does not pursue any practical aim. It can be conceived as a contribution to the study of the language, in a spirit of free enquiry, which also means freedom from any pressure that does not directly derive from the subject under scrutiny.

The above considerations militate in favour of a parser whose design keeps grammar and parsing algorithm as separate as possible. We achieve that aim by relying entirely on a *production system* as parsing algorithm. The production system is organized in *passes*. Each pass is implemented as a series of production rules which operate over and over again until they are unable to produce

anything new, in which case they pass control over to the next pass.

The production rules are allowed to build structure on the basis of what is available to them. In the first pass, the lexical pass, the words in the text are paired with the information stored about them in the lexicon (we have seen that this is a matter of straight variable instantiation). When we claimed above that the reading of *fata* as an nominative case would lead to failure, we certainly did not mean that this happened at the lexical look-up stage, where there is absolutely no information available to reject the nominative case or prioritize the accusative. The information is simply stored and made available to the next pass (higher in the structure-building hierarchy).

The grammar passes will likewise proceed from simple structures to more complex ones that need the information provided by the simple ones. Again, all the production rules in a given pass are allowed to produce structure over and over again, until they have nothing to add at their level.

Let's look at a very simple production rule to be found in the first grammar pass, that for building one-word nps such as *rex* in *rex scribit epistulam.*

The comments included in the Prolog program provide basic information about the np building procedures :

```
The core NPS are assembled before the other NPs, for which they can serve as building blocks.
There are indeed two passes for nps: core and finite.
The core NPs are simple nps that do not involve predications, therefore no relatives, no arg-bearing nouns,
just the simple buiding blocks: nouns as nps, names as nps, adj+n as np, and so on...

Each np is associated with an index which refers to the positions it spans in the input string
The index is useful to make sense of gaps, i.e. traces (t or e in syntactic parlance) 'left' by elements
'moved out of place' by 'transformations'. The quotes are meant to show distance with respect to the
syntactic theory underlining such treatment.

But undoubtedly a similar treatment is needed. If the trace cannot be associated with the relative pronoun,
and, via the relative, and more importantly, with the antecedent, all the controls we wish to perform, such as
semantic controls on arg bearers, will prove impossible in relative clauses, to give one example.
```

We then proceed to the production rules for simple nps, and begin with nps consisting of a single noun. We give below the *lex* clause for our word *rex*

```
lex(rex, noun, [pos:noun, txt:rex, lex:rex, case:nom, gender:masc, class:common, number:sing, sem:[hum]]).
```

The relevant production rule is the following:

```
[core,np1] ->
[mapped(noun,[from:A,to:B|FS]),
 constraint([pos:noun,lex:Lex,class:common,sem:Sem,txt:Text,
          number:Nb,gender:G,case:C],FS),
 map(np,[pathlist:[p(A,B)],hp:[p(A,B)],index:i(p(A,B)),distance:[0],
         cat:np,sem:Sem,class:common,lextype:full,
         number:Nb,person:3,gender:G, type:core,lex:Lex,txt:Text,
         case:C,w:1])].
```

Which basically means that if we have in our text (we have it *mapped* by a previous *production*), from position A to position B (remember that capitals are reserved for variables in Prolog – A and B are thus variables), an item that was placed in the noun box and is associated with *feature bundle* FS, then we can use the predicate **constraint** to check or retrieve information from that feature bundle: we need here a common noun, a full lexical item; we retrieve the information contained in the feature bundle regarding *number*, *gender*, *case*, *textual form* and *semantic class*: *sing*, *masc*,

nom(inative), *rex*, *[hum]*. We can then allow the production rule to produce (via the predicate **map**) a record to be put into the np (noun phrase) box.

Such a box will include information about the path covered by the np, the head of such a path, the index in case we need it somewhere down in the structure-building process (for instance if *rex* was to be found to be the antecedent of a relative pronoun), the distance within the path (in case of non-contiguity of the constituents[5]), morphological and lexical information derived from the lexical item. We add person (third person), a type (core np) and a weight (1).

This may seem to be a very heavy procedure just to account for what is dealt with in a couple of rewrite rules in a top-down or bottom-up parser, namely

np->n, n->rex or *rex->n, n->np.*

But in fact all the other information we gather and transport via the production rules based on feature unification will prove to be useful or downright indispensable in any sophisticated parser designed for a nearly free order language such as Latin.

The important design decision is to select a process (such as the one embodied in production rules) that boils down to monotonous incrementation of the available information. This does not prevent us from using specifically designed algorithms for ancillary tasks.

In short, we try to combine a parsing algorithm that is reduced to monotonous structure incrementation through a production system with various procedures that compute the quality rating of the structures licensed by the grammar rules and the information embodied in the lexical items (for instance the argument structure of a predicate, which has predictive power on how the string elements need to be structured into phrases of various levels such as nps and clauses).

A linguist will surely find that there is a huge distance between the type of grammar he is used to writing and the one he is confronted with in ALP. Well, there is a price to pay – an algorithm embodying a grammar cannot be a grammar written without an idea of how it is to be used in parsing. What we can attempt to do is to make the grammar as *declarative* as possible, i.e. as independent as we can make it from issues of control, of how it is to be used, in what order its rules are to be applied, where structures are to be stored, what should be done in case of failure, and the like. As soon as we attempt to go beyond toy systems, we have to dirty our hands a little and think about issues like the degree of freedom there really is in an 'order-free' language such as Latin, and what to do to assess the quality of the parses delivered by the system. The profit we will draw from such an effort is that we will increase explicitness, and have a much better idea of coverage, i.e. how much of the language can be captured by our rules.

Finally, is there nothing to be said *against* ALP as a parser for Latin? Well, there is one negative point, which, if we were parsing anything else but a dead language, would be rather devastating. The production system sketched here is *inefficient*[6] – its very monotony (it does one single thing:

5 In the pretty-printed parses, we give the value of the 'distance' feature if it is greater than zero. It records the number of words that separate elements that are expected to be contiguous. For instance, in the parse of Martial's line *Aestivo serves ubi piscem tempore quaeris?*, *aestivo tempore* gets assigned the function of adjunct of time with a discontiguity value (distance) of 3, on account of the three words (*serves ubi piscem*) that separate the two constituents of the adjunct, *tempore* and its adjective *aestivo*. This information is likely to be of use in word order and stylistic research.

6 It would seem that at present a fifteen word limit should be imposed on strings to be parsed, unless time does not matter AT ALL, which is seldom the case...

increase available information) is at that price.

We should bear in mind the reasons for which one may want to produce a Latin parser – I can see two main reasons only: teaching and research. The quicker the better shouldn't be our motto, surely.

Parsing Issues

Grammatical Sketch

The heart of the grammar implemented in ALP revolves around the *predicate* and its *arguments* building a *clause* (a grammatical clause, not to be confused with a Prolog clause). The clauses can be *finite* or *non-finite* (infinitive, participial) and can contain clauses as constituents. The arguments taking the form of *phrases* (adjective phrases, noun phrases, prepositional phrases) can also contain clauses, for instance under the guise of relative clauses attached to noun phrases. Recursivity is also to be found at the level of the phrases themselves, since a prepositional phrase is best defined as a preposition governing a noun phrase, and a noun phrase itself can contain prepositional phrases. The ease with which Prolog handles recursivity is a major pluspoint in its use in the implementation of a grammar for a natural language such as Latin.

The association of a predicate and its arguments is a matter for the lexicon to handle, but the description of the structural make-up of the arguments and the constraints imposed on them (e.g. semantic) must be such that the grammar can tackle them – the interaction between grammar and lexicon must be total, with no piece of information in either that the other cannot 'understand', i.e. register or make use of.

We will attempt to show this by looking at two lexical entries for verbs and the requirements that they impose on a grammar capable to match them.

The lexical entries are those for *obliuiscor* and *timeo*. We are not concerned here with the mechanisms building all their morphological variants, but with their argument list, which are housed in the relevant *lexarg* clauses:

```
% OBLIUISCI
% Non obliviscar sermones tuos - Pascal, Mémorial.
% Oblita est periculi ancilla fortior dominis multis.
% Obliuiscitur rex reginam longas epistulas scripsisse ancillae Marci.

lexarg(obliuisci,
      arglist:[ws(obliuiscor_forget,tr_cod,clause:[],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:np,oblig:no,constraints:[case:or([acc,gen])]]]),

             ws(obliuiscor_forget_that,tr_inf,clause:[],mwuw:0,
             args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                   object:[type:pred,oblig:yes,constraints:[type:nonfinite]]])]).
```

The arglist opens with an indication of the word sense being described; here we have two word senses, *forget* and *forget_that*. The information that follows concerns the *category* (transitive verb with np object, transitive verb with clausal object), the *constraints* it imposes on the clause in which it fits (here, the empty list indicates that there are no such constraints), its *weight* as a multi-word unit (0, since it is not a mwu), and then the *arglist* proper, the feature whose name is *args* and whose value is a list of arguments.

We should emphasize that the members of the arglist need not be found in the string to be parsed in the order in which they appear in the arglist (a canonical order used for easier maintenance of the

lexicon). As a matter of fact, in our second example for *obliuiscor* the genitive object precedes the subject, and in the first example (the one drawn from Pascal's Mémorial), the subject does not appear in the sentence at all, but is projected from the verb phrase (first person subject).

In the first wordsense recorded here, the args are subject and object, the first obligatory and the second optional (in which case it can be argued that it is mostly context-retrievable), and both structurally nps. The constraints on the subject are semantic (the subject must bear the feature +HUM), and the constraints on the object are case-related. The default cases are of course nominative for subject and accusative for object. But the object of *obliuiscor* can also be in the genitive case, so we need an OR-value for case: either accusative (as in the example from Pascal's Mémorial) or genitive (as in our second example).

In the second wordsense, the arglist specifies a clausal object, a non-finite clause (accusative cum infinitive), as in our third example.

We see thus that our grammar must be able to structurally characterize nps and assign them functions within the clause they operate in. They must also receive a semantic description, and have been assigned a case. The grammar must also deal with clauses in arg position, both finite and non-finite. For an example of a finite clause as arg we can turn to the entry for *timeo*:

```
% TIMERE
% Timeo Danaos etiam dona ferentes.
% Timeo amicis meis.
% Timeo ne veniant ad urbem capiendam.

lexarg(timere,
    arglist:[ws(timeo_fear,tr_cod,clause:[],mwuw:0,
        args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
            object:[type:np,oblig:yes,constraints:[case:acc]]]),

        ws(timeo_fear_for,tr_cod,clause:[],mwuw:0,
        args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
            object:[type:np,oblig:yes,constraints:[case:dat]]]),

        ws(timeo_fear_that,tr_cod,clause:[],mwuw:0,
         args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
              object:[type:pred,oblig:yes,
                   constraints:[type:finite,mood:subjunctive, argbound:yes,subordinator:or([ne,ut])]]])]).
```

In the case of *timeo*, we have three distinct word senses, each with its own arglist; we have a general *fear*, a *fear for* and a *fear that*. The constraints on the args have already been discussed, except for the new type of arg associated with *timeo* as *fear that*. The object is again a whole clause (type:*pred*), with its own list of constraints: it must be a *finite* clause, in the *subjunctive* mood, bound to that argument and opened with subordinator *ne* or *ut*. All requirements that the grammar must handle properly for the word sense to be captured.

The above examples give a very rough idea of the granularity problems that arise in the pairing of a lexicon (whose granularity should extend to the wordsense) with a grammar.

Dealing with Multi-Word Units

Another relevant example is the treatment of multi-word units. First, it should be clear that the description we give in the lexicon must allow their insertion into the grammatical framework we are implementing. Second, if their behaviour is not constrained in any way that the grammar is able to capture, their being read as mwu's must be given priority over the readings where they are just standard grammatical strings, although the latter readings cannot be excluded.

Consider a simple mwu such as *res nouae* (revolution). Lewis and Short :

4. Novae res, *new things, novelties:* nihil te ad me postea scripsisse demiror, praesertim tam novis rebus, Cic. Fam. 7, 18, 4.
Also *subst.:* **nŏvum,** i, *n., a new thing, a novelty; news:* novum attulerint, quod fit nusquam gentium, Plaut. Cas. prol. 70: num quidnam inquit novi? Cic. de Or. 2, 3, 13: si quid novi vel sero invenissem, Quint. 2, 5, 3.
Plur.: novorum interpositione priora confundere, Quint. 10, 3, 32; 8, 3, 60.
But, in gen., novae res signifies *political innovations, a revolution:* Q. Servilius Ahala Sp. Maelium novis rebus studentem manu suā occidit, Cic. Cat. 1, 1, 3: rerum novarum causam quaerere, id. Agr. 2, 33, 91: plebes novarum rerum cupida, Sall. C. 28, 4: cuncta plebes novarum rerum studio Catilinae incepta probabat, id. ib. 37, 1: novarum rerum avidi, id. J. 19, 1.
In a double sense: Segulium neglegamus, qui res novas quaerit: non quo veterem comederit—nullam enim habuit—sed hanc ipsam recentem novam devorārit, *innovations* and *new wealth,* Cic. Fam. 11, 21, 2.

We enter it in the lexicon as follows:

```
% RES NOVAE (revolution)
% cupiditate regni adductus novis rebus studebat (Caesar, De Bello Gallico, 1.9.3)
[core,np2aii] --->
[mapped(noun,[from:A, to:B|FSnoun]),
 mapped(adj,[from:X, to:Y|FSadj]),
 constraint([number:pl,gender:fem,case:Case,lex:res],FSnoun),  % plural needed, of course
 constraint([number:pl,gender:fem,case:Case,lex:nouus],FSadj),
 adjacent([p(A,B)],[p(X,Y)]), % adjacency required res nouae or nouae res
 append([p(A,B)],[p(X,Y)],Path),
 msort(Path, Sorted),
 map(np,[pathlist:Sorted,hp:[p(A,B)],index:i(p(A,B)),distance:[0],cat:np,class:common,sem:[abstract],
        number:pl,person:3,gender:fem,type:core,lex:res_nouae,lextype:full,
        case:Case,w:3])].
```

This entry specifies that the noun *res* and the adjective *nouae* should be adjacent (as opposed to the the adjective-noun nexus, where the two elements can be separated from each other: *res inuenit nouas*, 'he found new things'). The number is not free either: it must be plural, as opposed to the unspecified number of a standard adjective-noun nexus: *noua res*, 'a new thing'). Third, of course, the lexemes are specific: the noun must be *res* and the adjective must be *nouus*. If the relevant constraints are satisfied, we build a standard np, to which we assign a specific semantics (standard *res* can be sem:[thing]) and a specific lexical value (*res_nouae*). And of course we increase the weight assigned to the np. An np made up of a noun and an adjective will have weight 2 and the weight we assign to *res_nouae* is 3.

Let's now consider a more complex mwu, i.e. one with wider, less local constraints.

```
% ALIQUEM/QUOD (NON) PILI FACERE
% Praetor non amabat milites nec faciebat pili cohortem.

lexarg(facere,
      arglist:[ws(mwu_non_pili_facio_not_give_a_damn,tr_cod_cplt,clause:[[polarity:neg]],mwuw:2,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:np,oblig:yes,constraints:[case:acc]],
                  object_cplt:[type:phrase,frozen:yes,oblig:yes,constraints:[lex:pili]]]),
```

Remember Catullus, 10, 12-14? *praesertim quibus esset irrumator / praetor, nec faceret pili cohortem.* (nòt least whén said praetor was a fuckface / *and* didn't give a shit for his poor staffers. - translation Peter Green).

Pili facere needs to be inserted in a non-affirmative context (provided here by *nec*, but there are other ways of making a context non-affirmative, such as *quid obstat quominus*/*quid est causae quin*). We therefore introduce a clause-level constraint, on clause polarity: clause:[[polarity:neg]]. We give a bonus weight to the verb (2). The object complement is a phrase, quite frozen, and whose lexical specification goes down to the word-level: we need *pili*, nothing else will do (the lexicon has an entry for the relevant phrase, namely *string(phrase,[pili],[lex:pili,w:1]).*

Linearity

Let us consider the *contiguity check* algorithm as an example. In Latin, elements that belong together need not be found side by side in the string. A standard example is the Vergilian *patulae recubans sub tegmine fagi*, where the adjective *patulae* belongs to the noun *fagi* (both singular, both feminine, both genitive – the agreement triplet that we require to hold on the adjective-noun nexus). If we allow elements belonging together to be dispersed in the string, we won't find it too difficult to account for the np, precisely on the basis of the triple agreement required on the adj+n group. But notice that there is nothing in the intervening material (*recubans sub tegmine*) that would offer a better link for the adjective *patulae*. Just as *fagi* is justifiable as genitive attached to *tegmine*, and *tegmine* as governed by *sub*, and *sub tegmine* as place adjunct for *recubans*, *patulae* is justifiable as adjective attached to *fagi* – there is no other attachment to compete with the one that involves non-contiguity between adjective and noun.

In the case of competing links we need to assess the weight to be attached to each link and select the heavier of two or the heaviest among more than two. How do we proceed?

The first thing to notice is that first and foremost we need to keep track of where each string element is to be found in the string. We have seen how to do this with the algorithm computing positions while turning the string into a *wordlist*.

Now, to be able to use that information the way we should, we need to keep track of the position of the material covered in the case of any structure that gets superimposed on lexical material. We store this information in the *path* feature. If, in *bonus rex scripsit epistulas malas*, we build a np to cover *bonus rex* and another one to cover *epistulas malas*, we need to record in the first np a path extending from position 0 (in front of *bonus*) to 2 (at the end of *rex*), and in the second np a path extending from 3 to 5.

We also need to be able to say where the head of a structure is to be found. In the case of our two nps, the head is the noun and we record positions 0 to 1 for the first head (*rex*) and 3 to 4 for the second (*epistulas*).

We can now discuss the algorithms relating to path, distance and contiguity. We go straight to the Prolog program, where the comments are supposed to do part of the job of explaining how the algorithms work. We add a few more comments to make the procedure as explicit as we can.

```
% these procedures examine which parts of the string are covered by various elements
% they are meant to measure properties like adjacency, contiguousness and distance

% the path is a list of p(X,Y) structures, where X and Y stand for positions in the string,
% as computed when the string is entered in and processed.
```

FINDING A PATH THAT COVERS THE WHOLE SENTENCE

```
% all the words of a string must be used up for a parse to be considered valid for the string
% no gap left and the end of the sentence must have been reached

% Begin and End are the extremities of the pathlist (0 and whatever fin(Fin) records)
```

The *fin(Fin)* predicate stores in the variable *Fin* the last position in the string. It is taken care of by the process turning the string into a wordlist.

```
path(Begin,End,Pathlist):-
        pick(p(Begin,Next),Pathlist,RPaths),
        path(Next,End,RPaths).

path(E,E,[]).
```

The procedure applies recursively to its third argument. The second clause for the predicate (i.e. path(E,E,[]) gets us out of recursion: when all the elements have been picked out, we are left with an empty list ([]) whose beginning and end are the same (single variable E, the end of the path). More on this below.

As for *pick*, it is a three-arg procedure. It arbitrarily picks up an element (first arg) in a list (second arg) and returns what remains of the list once the selected element has been taken out (third arg). It can be defined as follows:

```
pick(H,[H|T],T).
pick(X,[H|T],[H|T1]) :- pick(X,T,T1).
```

In both *path* and *pick* use is made of the data structure *List* and the core of the procedure is recursive, i.e. calls on itself in its very definition. This is quite a standard way to proceed in Prolog. Lists are explorable by means of the operator '|', whose second argument, remember, is always a list. We can recur on that list until we are left with the empty list, which we use in a defining clause which gets us out of recursion. Consider *pick* in that light. We begin with the simplest case one can think of: picking an element out of a list is achieved by selecting the element which is easiest to grasp, i.e. the head of the list, the element to the left of the operator '|'. The element picked is thus H, and the remainder of the list, which is a list, is T (the tail of the list). But the element to be picked can be anywhere in the list; we can select it by picking it out of the remainder of the list, which is what the second defining clause says: leave the head(H) well alone, and pick your X element in the remainder of the list, T. The list you will get as a result of picking out element X out of tail T we will refer to with variable T1. Therefore the list that should be returned as result of the picking is head H and tail T1, that is to say [H|T1]. Here the end of recursion clause does not refer to the empty list, because there is nothing to pick in an empty list – we need at leat one element, and the pattern [H|T] therefore applies to the list in the clause that gets us out of recursion, in this case the first one.

In *path*, however, we need to explore the whole length of the pathlist. We need to pick elements until we reach the very last position referred to as *End* in the first clause. At that juncture the Pathlist must be empty, and we get out of recursion by means of the second defining clause for *path*. The Begin and End point must be the same, since we then start from the end-position of the very last element in the path. We stay put, having reached our goal. The path is really a path because we cannot pick a new element (a new p-structure) unless its beginning point corresponds to the endpoint of the preceding p-structure. Backtracking ensures that a path will be found if there is a path to be found, i.e. all the p-stuctures can be joined by sharing a position (endpoint of one is start of the following).

ADJACENCY

% see nps with genitive np as subconstituent for an example of the relevance of such a procedure

% **strict**

% one pair in the first path has an end which corresponds to the beginning of a pair
% belonging to the second path, or the other way round

% [p(3,5), p(2,3] and [p(5,6), p(6,8), p(8,9)] for instance

```
adjacent(PL1,PL2):- member(p(_,Y),PL1), member(p(Y,_),PL2),!.
adjacent(PL1,PL2):- member(p(_,Y),PL2), member(p(Y,_),PL1).
```

The member predicate is self-explanatory: member(Element, List) succeeds if Element is a member of List. It can be defined as follows:

```
member(H,[H|T]).                    % the Head of a List is a member of that List
member(H,[_|T]) :- member(H,T).      % if it's not the Head, it should be a member of the tail T
```

% **relaxed adjacency**

% a distance of 1 or 2 (in the case of relaxadjacent2) is allowed between the two corresponding pairs

```
relaxadjacent(PL1,PL2):- member(p(_,Y),PL1), member(p(X,_),PL2), succ(Y,X),!.
relaxadjacent(PL1,PL2):- member(p(_,Y),PL2), member(p(X,_),PL1), succ(Y,X).

relaxadjacent2(PL1,PL2):- member(p(_,A),PL1),
                          member(p(C,_),PL2),
                          succ(A,B),
                          succ(B,C),!.

relaxadjacent2(PL1,PL2):- member(p(_,A),PL2),
                          member(p(C,_),PL1),
                          succ(A,B),
                          succ(B,C).
```

The *succ* predicate gives the successor of an element in the list [0,1,2,3,4.....n], e.g.
succ(3,4) succeeds while succ(4,3) and succ(3,5) fail.

Note that *succ* is a *pre-defined* predicate, i.e. it belongs to the Prolog programming language and need not be defined by the user, i.e. the Prolog programmer. We therefore give it no definition in these notes.

% relaxed adjacency with control on intervening elements

% with respect to POS:noun

% sometimes we have to check that no noun occurs in an interval
% as when we wish to relate the heads of nps linked by the cplt noun relation
% involving a genitive phrase
% *Marci servas amicos*
% *Marci* preferably linked with *servas* rather than with *amicos*:
% *putabas Marci servas amicos reginae amasse*

```
relaxedadjacent1_n(PL1,PL2,n):- member(p(_,Y),PL1), member(p(X,_),PL2),
                                succ(Y,X),
                                \+ mapped(noun,[from:Y,to:X|FSnoun]).

relaxedadjacent2_n(PL1,PL2,n):- member(p(_,A),PL1), member(p(C,_),PL2),
                                succ(A,B),succ(B,C),
                                \+ mapped(noun,[from:A,to:B|FSnoun1]),
                                \+ mapped(noun,[from:B,to:C|FSnoun2]).
```

Notice the \+, which is used in Prolog for negation. \+mapped is used to indicate that no element of the box noun exists that covers the intervening element, i.e. that the intervening element is not a noun.

```
% with respect to CASE GENDER and NUMBER


% when we try to relate adj and noun
% we are not likely to be allowed to jump a noun with all the right properties in terms of
% case gender and number:
% putabas malas servas amicas reginae fuisse
% malas is not likely to link with amicas by 'jumping' servas
% in the code below we use the cut-fail pair ('!',fail). The cut ('!') prevents backtracking, so that the fail
% that follows cannot be undone; the predicate being defined fails if a noun is found where it shouldn't be.
% Both the cut and the predicate fail are pre-defined

relaxedadjacent1_cgn(PL1,PL2,Case,Gender,Nb):- member(p(_,Y),PL1), member(p(X,_),PL2),
                                succ(Y,X),
                                mapped(noun,[from:Y,to:X|FSnoun]),
                                constraint([case:Case,gender:Gender,number:Nb],FSnoun),
                                !, fail.

relaxedadjacent1_cgn(PL1,PL2,Case,Gender,Nb).

relaxedadjacent2_cgn(PL1,PL2,Case,Gender,Nb):- member(p(_,A),PL1), member(p(C,_),PL2),
                                succ(A,B),succ(B,C),
                                ( (mapped(noun,[from:A,to:B|FSnoun1]),
                                  constraint([case:Case,gender:Gender,number:Nb],FSnoun1)) ;
                                  (mapped(noun,[from:B,to:C|FSnoun2]),
                                  constraint([case:Case,gender:Gender,number:Nb],FSnoun2))),
                                !, fail.

relaxedadjacent2_cgn(PL1,PL2,Case,Gender,Nb).

relaxedadjacent3_cgn(PL1,PL2,Case,Gender,Nb):- member(p(_,A),PL1), member(p(D,_),PL2),
                                succ(A,B),succ(B,C),succ(C,D),
                                ( (mapped(noun,[from:A,to:B|FSnoun1]),
                                  constraint([case:Case,gender:Gender,number:Nb],FSnoun1)) ;
                                        ( (mapped(noun,[from:B,to:C|FSnoun2]),
                                          constraint([case:Case,gender:Gender,number:Nb],FSnoun2));
                                          (mapped(noun,[from:C,to:D|FSnoun3]),
                                          constraint([case:Case,gender:Gender,number:Nb],FSnoun3)))),
                                !, fail.

relaxedadjacent3_cgn(PL1,PL2,Case,Gender,Nb).
```

PATH CONTIGUITY

```
% the various elements follow each other without leaving a gap

contiguous([]).
contiguous([One]).

contiguous([p(X,Y),p(Y,Z)|Tail]):- contiguous([p(Y,Z)|Tail]).
```

In words : an empty list is contiguous, i.e. does not feature non-contiguity...
A one-element list does not feature non-contiguity either
If the first two elements in a list are contiguous (the extremity of the first being the start of the second), then if the list made up of the second element and the tail of the list (all the remaining elements) is also contiguous, then the whole list is contiguous.

```
% in quasicontiguous we allow one element to be out of place

quasicontiguous(L):- contiguous(L), !.                    % Qui peut le plus...

quasicontiguous(L):- pick(El,L,L1), contiguous(L1).
```

The *quasicontiguous* predicate can be applied as a check on non-finite clause constituency when dealing with poetry. The structures building such a clause must be found together, with the exception of a single word. This relaxed check on contiguity allows the parsing of the Horatian *Me tabula sacer votiva paries indicat uvida suspendisse potenti vestimenta maris deo*, where *Me* belongs to the non-finite complement clause of *indicat*: *Me … uvida suspendisse potenti vestimenta maris deo.*

DISTANCE BETWEEN TWO PATHS

```
% we first determine the end points of the two paths
% we determine the order in which they appear
% and then the distance between extremity of the first one and start of the second

distance(Path1,Path2,Distance):- extremity(Path1,Ext1), extremity(Path2,Ext2),
                        start(Path1,St1), start(Path2, St2),
                        ifthenelse(Ext1 =< St2,   % IF
                                Distance is St2 – Ext1, % THEN
                                Distance is St1 – Ext2). % ELSE

% extremity: last position in pathlist
% we select the very last position registered, i.e. the second element of the p(X,Y) structure that ends the
% path

extremity(PathList, Ex):- last(PathList,p(_,Ex)).

                % (last(List,Last) is true if Last is the last element of List)

% last, although pre-defined (i.e. part of the Prolog language) can be re(?)defined as follows:

last([Last],Last).                % Last is the last element of a list which does not contain anything else

last([_|Tail],Last):- last(Tail,Last). % If there is more than a single element in the list, then Last is the
                                % last element of the Tail of the list


% start: first position in a pathlist

% the first element of a list is easy to find by simple unification:
% we select the first element of the relevant p structure

start([p(Start,_)|_],Start).


% precedes(Path1,Path2)

precedes(P1,P2):- extremity(P1,Extremity), start(P2,Start), Extremity =< Start.
```

Consider the following line from Martial, 2.78:

Aestivo serves ubi piscem tempore quaeris? (You want to know where to keep fish in summertime?)

```
[0/aestiuo,1/serues,2/ubi,3/piscem,4/tempore,5/quaeris,endpos(6)]
cputime : 0.203125

   illocutionary_force:question
   vg
      selected_reading:quaero_ask
      polarity:pos
      cat:vg
      * quaeris *
      pos:v
      lex:quaerere
      voice:act
      tense:present
      mood:indicative
      number:sing
      person:2
   subject
      source:context_retrievable
      number:sing
      gender:or([masc,fem])
      person:2
      cat:np
      index:i(0,0)
      constraints_to_be_met:[sem:[hum]]
      case:nom
   object
      cat:pred
      * serues * * ubi * * piscem *
      illocutionary_force:question
      number:sing
      person:3
      vgperson:2
      gender:neuter
      mood:subjunctive
      tense:present
      polarity:pos
      add:no
      checkint:yes
      constituent_structure
         illocutionary_force:question
         vg
            selected_reading:servo_keep
            polarity:pos
            cat:vg
            * serues *
            pos:v
            lex:seruare
            voice:act
            tense:present
            mood:subjunctive
            number:sing
            person:2
         subject
            source:context_retrievable
            number:sing
            gender:or([masc,fem])
            person:2
            cat:np
            index:i(0,0)
            constraints_to_be_met:[sem:[hum]]
            case:nom
         object
            * piscem *
            index:i(p(3,4))
            cat:np
            sem:[thing]
            number:sing
            person:3
            gender:masc
            lex:piscis
            case:acc
            constituent_structure
            piscem
         adjunct
            * ubi *
            cat:advp
            value:place
            lex:ubi
            constituent_structure
               lex:ubi
               sem:location
               cat:advp
   adjunct
      * aestiuo * * tempore *
      distance:3
      cat:np
      value:time
      number:sing
      person:3
      gender:neuter
      lex:tempus
      case:abl
      constituent_structure
         head
            lex:tempus
            sem:time_when
            cat:np
            number:sing
            gender:neuter
            case:acc
            index:i(p(4,5))
      adj:aestiuus
```

Such a sentence seems to exhibit a completely free word order. But it suffices to run it through a string generator to come to realize that this is far from the case. A six-word sentence generates 6!, i.e. 720 strings, most of which are totally ungrammatical:

aestivo piscem quaeris serves tempore ubi .
aestivo piscem quaeris serves ubi tempore .
aestivo piscem quaeris tempore serves ubi .
aestivo piscem quaeris tempore ubi serves .
aestivo piscem quaeris ubi serves tempore .
…
aestivo tempore quaeris ubi serves piscem .
aestivo tempore serves piscem quaeris ubi .
...
piscem quaeris tempore ubi aestivo serves .
piscem quaeris tempore ubi serves aestivo .
piscem quaeris ubi aestivo serves tempore .
piscem quaeris ubi aestivo tempore serves .
…
quaeris piscem ubi aestivo serves tempore .
quaeris piscem ubi aestivo tempore serves .
quaeris piscem ubi serves aestivo tempore .
...
serves aestivo piscem quaeris tempore ubi .
serves aestivo piscem quaeris ubi tempore .
serves aestivo piscem tempore quaeris ubi .
...
tempore quaeris piscem aestivo ubi serves .
tempore quaeris piscem serves aestivo ubi .
tempore quaeris piscem serves ubi aestivo .
...
ubi aestivo quaeris piscem serves tempore .
ubi aestivo quaeris piscem tempore serves .
...
ubi tempore serves quaeris aestivo piscem .
ubi tempore serves quaeris piscem aestivo .

So, the task does not boil down to letting the 'anything goes' principle make havoc of Latin word order, but to open up the range of possible grammatical strings by relaxing the contiguity we expect structures to exhibit, while taking care not to allow the production of strings that would turn out to be impossibly ambiguous. The various path procedures enable us to contain freedom within reasonable (i.e. grammatical) limits.

Producing, Storing and Retrieving Information

A few words may be in order about the *data bases* ALP uses. A first data base consists of the Prolog program itself, made up of clauses embodying both facts and rules.

This data base is increased by running the **makelex** program, which involves expanding its macro-clauses. This process yields new lexical clauses, resulting from the generation of the morphological variants for regular lexical items such as adjectives, nouns and verbs. The irregular or invariant forms are entered directly as *lex* clauses. Let us look at examples of both:

a) directly entered as lex clauses:

semper (invariable):
```
lex(semper,adv,[lex:semper,pos:adv,type:clausal, sem:time]).
```

simus (irregular)
```
lex(simus,v,[pos:v,class:v_esse,type:finite,lex:esse,
        voice:act,txt:simus ,tense:present,kind:std,mood:subjunctive,
        number:pl,person:1]).
```

b) generated on the basis of a macro-clause:

```
lex(rogabis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabis, tense:future, kind:std, mood:indicative, number:sing, person:2]).
lex(rogabit, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabit, tense:future, kind:std, mood:indicative, number:sing, person:3]).
lex(rogabitis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabitis, tense:future, kind:std, mood:indicative, number:pl, person:2]).
```

If we go to the macro-clause itself, we see that it uses the root provided by a *verb* clause (in this case for the verb *rogo*) and the list of endings suitable for that verb, to generate the morphological variants. Each variant is then turned into the first argument of a *lex* clause, the remaining arguments being the Part of Speech and a list of features (tense, mood, person, etc.) to be associated with that particular variant. The resulting new bunch of *lex* clauses (such as the three above) are then asserted by the macro-clause, i.e. added to the Prolog data base.

The verb clause for *rogo* (entered as such as part of the Prolog program) reads:

```
verb([v(rogare,1,rog,rogau,rogat)],tr_cod,std).
% the v functor encompasses infinitive, conjugation and the three roots. We then have the verb class, and
% the indication that the verb behaves 'standardly' with respect to the production of morphological variants
```

The macro-clause involved is much too long for it to be given in full in this introduction. Suffice it to say that it needs access to the relevant roots and endings, and performs atom-concatenation to produce the morphological variants. The atoms it concatenates (i.e. chains together in the order specified) are simply the relevant roots and endings.

The process results in the production of 213 *lex* clauses for the morphological variants of *rogo,* among which the three given above. The full list is to be found at the end of this document, just before the Appendices

As for the predicate-argument structure associated with the verb, it is the object of a specific clause for each verb. Such are the *lexarg* clauses, whose first argument is the arg-bearing element (in this case the verb *rogare*) and the second the *arglist feature*, whose value is a list of *word senses* accompanied by the arg structure they require.

```
% ROGARE

% Examples of the word senses accounted for in ALP
% Eo auxilium rogatum.                                  Ask for
% Rogebant quae fortuna exercitus esset.                Ask

lexarg(rogare,
      arglist:[ws(rogo_ask_for,tr_cod,clause:[],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:np,oblig:yes,constraints:[case:acc]]]),

            ws(rogo_ask,tr_cod,clause:[],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:pred,oblig:yes,
                   constraints:[type:finite,mood:subjunctive, checkint:yes]]])]).
```

In the example for *rogare*, we distinguish between two wordsenses, to capture the distinction between *rogo* meaning *ask* and *rogo* meaning *ask for. Rogo* meaning *ask for* has an np as object, whereas *rogo* meaning *ask* governs a clause, with its own constraints: it must be finite, its mood must be subjunctive, and it must be interrogative. The other requirements we hope do not need further explanation.

We now move on to discuss the second type of data base used by ALP.

This data base has a limited life-span. It is specific to each sentence being parsed, and is erased as soon as the sentence has been parsed. It houses all the productions that the production rules embodying the parser have generated all through the parsing process of that sentence, from working out the positions of the items within the string, down to the structures corresponding to a full parse of the input string.

It should be clear that while processing a sentence, no information yielded by the production rules is ever erased. The process is strictly incremental, producing bits of structure that will or will not contribute to the final parse or parses. That will be decided by the higher levels materialized by higher passes, the structures retained as valid parses having to span the whole string.

In such a system the linguist *describes* rather than sets about specifying the way the information he gives in the grammar and lexicon should be used (in what order, under what conditions, etc.). As soon as the coverage is to be more than strictly minimal, it makes a hell of a difference for the linguist's job if he hasn't to turn into a programmer each time he wants to add new lexical items, and, above all, new constructions.

The predicates for adding information to, and removing information from, that second type of data base, are specific to it. Instead of the *assert/retract* pair (used for the permanent db), we have the *recorda/erase* pair. No confusion is possible.

Recorda works with as many storage areas (called *Boxes* in ALP) as one wishes, and can store any Prolog term, i.e. anything ALP is ever likely to produce, from tiny lexical bits up to whole parses.

We use *recorda* both to store information and to retrieve recorded information.

Information gets stored in this second type of data base if it is not already there. The situation in which the parser can no longer record new information signals the end of the parsing process. We use two predicates, *map* and *mapped*, the first to store, the second to retrieve information. They are defined as follows :

```
map(Box,FS) :- not(recorded(Box,FS,_)),
                recorda(Box,FS,_).
        % the not recorded condition is essential for keeping track of a 'no new solution' situation
        % which enables the various parser runs to come to an end
         % in most cases FS is a list of features

mapped(Box,X) :- recorded(Box,X,_).
```

Note that we do not bother about the third argument of either *recorda* or *recorded*. Remember that using the anonymous variable (_) is the standard way of indicating that one is not interested in the information stored there. As a matter of fact, we could have used the two-arg versions of *recorda* and *recorded*, also part of SWI-Prolog, which simply leave out the information we are not interested in.

Mapping

To illustrate this process, we can look at the treatment of simple np groups made up of a noun and an ajective phrase as in *agricola doctior Petro*. We take up the analysis at a point where the adjective phrase has already been built and needs to be attached to the noun to build the resulting np.

We give the commented Prolog code, adding some more comments in an attempt to persuade the reader to stay with us a few pages more...

```
% adj phrase following the noun


[finite,np2a] ->  % category and name of the rule

[mapped(noun,[from:A, to:B|FSnoun]),      % we have a noun in the word list corresponding to the input string
                                          % it extends from position A to position B
                                          % its feature bundle has been read off the lexicon
        % in the case of agricola (taking it as a nominative), it would be
        %[pos:noun, txt:agricola, lex:agricola, case:nom, gender:masc, class:common, number:sing, sem:[hum]]

 mapped(adjp,FSadj),
 % we have an adjective phrase (here covering doctior Petro) with its own feature bundle, sth along the
 % lines of

% [cat:adjp,pathlist:Sorted,distance:Distance,hp:[p(A,B)],
%                 case:Case,number:N, gender:G,lex:Lex,type:Type,w:Weight,
 %                c_str:[Lex,comp_cplt:FSnp]]

% with the variables duly instantiated


 constraint([number:Nb,gender:Gender,case:Case,class:Class,sem:Sem,lex:LexNoun],FSnoun),

% we select from the feature bundle associated with the noun
% the values for number, gender, case, etc.

 constraint([number:Nb,gender:Gender,case:Case,lex:LexAdj,type:Type,w:W],FSadj),

% we select values from the feature bundle associated with the adjective phrase,
% leaving unification to check that the agreement triplet Number, Gender, Case holds

 Type \= int,                % interrogative adjectives dealt with separately - they need heavy weight

 constraint([pathlist:Padj],FSadj),
% we retrieve the pathlist of the adjective phrase

 append([p(A,B)],Padj,Pnp),
% we append i.e. concatenate the path for the noun and the path for the adjective phrase

 extremity(Padj,Ext),              % we select the endpoint of the path covered by the adjective

 Ext > B,                      % here the adj phrase follows the noun, B being the endpoint of the noun

 distance([p(A,B)],Padj,Distance),    % the distance between the noun and the path of the adjective
                          % determines the straining factor as well as helping to decide
                                  % whether noun and adj DO belong together
                          % the straining factor will contribute negatively to the weight assigned
                          % to the parse

 ifthen(LexAdj=is, Distance=0),    % is/ea/id adjacent - this requirement is probably too strong


 msort(Pnp, Sorted),  % merge sort
 \+dup(Sorted),          % no duplicates – recall that \+ is the negation operator in Prolog

% we standardly apply this couple of procedures to paths
% first, we sort them ; second, we check that they do not contain duplicates


 Distance < 4,                % 3 is thus the maximum distance
                                  % between adj and noun
                          % a Prolog call that acts as a barrier – if it fails, the whole thing fails
```

```
 % we still have to exclude the occurrence, within the gap, of nouns to which the adjective could be
 % attached with priority, because they agree in the well-known agreement triplet
% we use relaxedadjacentN_cgn
%  (where N=1,N=2,N=3, and cgn means that case gender and number are checked for agreement)


 ifthen(Distance=3, relaxedadjacent3_cgn([p(A,B)],Padj,Case,Gender,Nb)),
                        % three in between, neither of them a noun with relevant triplet
 ifthen(Distance=2, relaxedadjacent2_cgn([p(A,B)],Padj,Case,Gender,Nb)),
                        % two in between, neither of them a noun with relevant triplet
 ifthen(Distance=1, relaxedadjacent1_cgn([p(A,B)],Padj,Case,Gender,Nb)),
                        % one in between, not a noun with same [gender,number,case] triplet


  % Weight is W+1,
  % we increase the weight of the adjective phrase (as computed when the phrase was parsed)
  % with the weight assigned to a single noun, i.e. 1
  % myplus is the same as plus, but does not fail if it meets with a variable instead of a number
  myplus(W,1,Weight),  % means W+1=Weight, a formulation that would lead to disaster in Prolog,
                          % since the equal sign (=) is used for unification, not addition !!!
                          % standard Prolog requires Weight is W+1


  % we can now build the resulting NP
  % note that the head of the NP is the N,
%  which also yields the index reference used for binding traces (as in relative clauses)

  map(np,[pathlist:Sorted,hp:[p(A,B)],index:i(p(A,B)),distance:[Distance],   % distance is recorded as
                                                                              % straining factor
          cat:np,class:Class,sem:Sem,
          number:Nb,person:3,gender:Gender,type:core,lex:LexNoun,lextype:full,
          case:Case,w:Weight,
          c_str:[head:FSnoun,adjp:FSadj]])].
```

the *c_str* is the constituent as it appears in the parse tree : (we use the parse produced by ALP for the sentence : [0/agricola,1/doctior,2/petro,3/misit,4/reginae,5/epistulam,endpos(6)], selecting the bit assigned to the subject (*agricola doctior Petro) :*

```
subject:[pathlist:[p(0,1),p(1,2),p(2,3)],hp:[p(0,1)],index:i(p(0,1)),distance:[0],
          cat:np,class:common,sem:[hum],
          number:sing,person:3,gender:masc,
          type:core,lex:agricola,lextype:full,case:nom,w:3,

c_str:[head:[pos:noun,txt:agricola,lex:agricola,case:nom,gender:masc,class:common,number:sing,sem:
[hum]],

           adjp:[cat:adjp,pathlist:[p(1,2),p(2,3)],distance:[0],hp:[p(1,2)],
                 case:nom,number:sing,gender:masc,
                 lex:doctus,type:std,w:2,

c_str:[doctus,
         comp_cplt:[pathlist:[p(2,3)],hp:[p(2,3)],index:i(p(2,3)),distance:[0],
                    cat:np,sem[hum],class:proper,
                    lex:petrus,lextype:full,number:sing,person:3,gender:masc,
                    type:core,case:abl,w:1]]]]]
```

which pretty-prints as:

```
subject
    index:i(p(0,1))
    cat:np
    sem:[hum]
    number:sing
    person:3
    gender:masc
    lex:agricola
    case:nom
    c_str
      head
        pos:noun
        lex:agricola
        case:nom
        gender:masc
        number:sing
        sem:[hum]
      adjp
        cat:adjp
        case:nom
        number:sing
        gender:masc
        lex:doctus
        c_str
          doctus
          comp_cplt
            index:i(p(2,3))
            cat:np
            sem:[hum]
            lex:petrus
            number:sing
            person:3
            gender:masc
            case:abl
```

Matching

Since we have been looking at *arglists*, we'll now say a few words about the process by which the argument requirements are satisfied, i.e. matched with structures to be found in the string to be parsed. We have already pointed out that the args do not have to be found in the canonical order in which they appear in the *arglist*. It also stands to reason that the args marked as optional need not instantiated, but if they are, they contribute to the weight assigned to the predicate-arg nexus.

Consider the matching of the *subject* arg. We have already seen that the subject could be projected from the verb group, which is the standard case when the subject is first or second person, but a third person subject may also be textually retrievable.

Voice will affect the arglist. In the passive voice, the object arg will be assigned the subject function, and the subject arg will be demoted to a prepositional phrase status (ab+ablative) or will be assigned the ablative case, and will in all cases be optional. Such transformations to the arglist must be accomplished as soon as we have ascertained the predicate's voice, which should be early enough in the parsing process (but remember a very important property of the production system: rules fire automatically when the material they need is ready, i.e. has been made available by lexical look-up or the previous firing of grammatical rules and their production of the required structures. It is NOT the linguist's task to worry about sequence in the application of rules. Considering the very highly recursive nature of grammar, this is a key property of production systems).

We'll now look at the code for updating the args to be matched in the case of a passive voice being found in the arg-bearer, i.e. the predicate, in a non-finite clause (*puto reginam ab ancilla marci amari*).

```
 mapped(vg,FSverb),
  constraint([type:nonfinite,mood:Mood,voice:Voice,tense:Tense,       % nonfinite verb form (in our case: amari)
          pathlist:PathlistVerb,lex:Clex, w:WVerb],FSverb),

lexarg(Clex,arglist:ArgList),                                  % connection with the args via lexarg
                                                               % the args are those for amo:
                                        lexarg(amare,
                                        arglist:[
                                        ws(amo_love,tr_cod,clause:[],mwuw:0,
                                        args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                                        object:[type:np,oblig:no,constraints:[case:acc]]])]).

pick(ws(Lex,Class,clause:Clause_Constraints,mwuw:MW,args:Args), ArgList,_),
                    % picking a word sense to see if it is appropriate ...
                    % remember that in general there will be more than one
                    % ws (i.e. word sense) for a given arg-bearer

ifthenelse(Voice=pass,                % outer THEN              % PASSIVE, as here amari

          % selecting the object to remove it from the arglist and turn it into a subject

           (pick(object:ObjectSpecs,Args,Args1),  % note that there must be an object if a passive was produced !!
            pick(oblig:Oblig, ObjectSpecs,OS1),    % but perhaps it was not obligatory, as in the case of amare
            pick(constraints:Oconstraints,OS1,OS2), % and we must also update the constraints
            pick(case:Caseobj,Oconstraints,Oconstraints1),
                                                    % recall that the predicate pick selects an element and
                                                    % removes it from the list
                                                    % case is buried within the constraints associated with the arg
                                                    % we pick it out

            append([case:acc],Oconstraints1,NewOconstraints), % subjects are accusatives in nonfinite clauses !!
                                                                % here reginam
            append([constraints:NewOconstraints],OS2,NOS),
            append([oblig:yes],NOS,NewObjectSpecs),          % there must be a subject if a passive is used !!!


          % turning the subject into an optional (a+) abl pp arg
           pick(subject:SubjectSpecs,Args1,Args2),
           pick(constraints:Sconstraints,SubjectSpecs,SS1),

           ifthenelse( Sconstraints=[],
                    % no constraint on subj: both types of agent are OK        IF-CLAUSE

                ( NewArg=[type:pp,oblig:no,constraints:[prep:ab,sem:[hum]]] ;
                                                    % THEN-CLAUSE note the OR operator (;)
                  NewArg=[type:np,oblig:no,constraints:[case:abl,sem:[thing]]])  ,

                                                    % ELSE-CLAUSE:
            % ab+hum vs simple abl for non-hum

            ifthenelse(   constraint([sem:[hum]],Sconstraints),   % IF2-clause       case of amare
                                                                   % there are constraints: we act accordingly
                        NewArg=[type:pp,oblig:no,constraints:[prep:ab,sem:[hum]]],    % THEN2-clause
                        NewArg=[type:np,oblig:no,constraints:[case:abl]])),          % ELSE2-clause

            append([agent:NewArg],Args2,At),
            append([subject:NewObjectSpecs],At,Argstomatch) ),  % reconstructing the arglist

          Argstomatch=Args),                  % outer ELSE:  ACTIVE : leave the args as they are in the arg specs


         % the remainder of this bit of code concerns the matching of the new arglist
         % and is not discussed here
```

A Voice in the Middle...

Since we have been discussing voice, this might be the right place to point out that ALP works with a *middle* voice, by the side of the active and the passive.

Morphologically, we generate forms that look like passives, but concern third person singular of intransitive verbs, e.g. *insanitur*, *insaniatur*, *insaniebatur* and *insanietur* for *insanio*, for instance. This middle voice is much nearer to active than to passive. The passive touch is found in the generality and impersonality of the process, which can convey a sense of inevitability: *Ibatur in caedes...*

```
vg
    selected_reading:eo_go                  % Ibatur
    polarity:pos
    cat:vg
    pos:v
    lex:ire
    voice:middle
    tense:imperfect
    mood:indicative
    number:sing
    person:3
  prep_cplt                                 % in
    case:acc
    prep:in
    sem:[abstract]
    lex:caedes                              % caedes
    index:i(p(2,3))
    cat:pp
    c_str
       prep:in
       head
          index:i(p(2,3))
          cat:np
          sem:[abstract]
          number:pl
          person:3
          gender:fem
          lex:caedes
          case:acc
```

We also register *middle* voice in two other cases:

```
% 1. pugnatum est

[verb,vg5amiddle] --->
[ mapped(v,[from:X, to:Y|FSverb]),
  constraint([lex:esse,type:Type, person:3, number:sing, tense:Tense,mood:Mood],FSverb),      % esse
  mapped(v,[from:C, to:D|Supine]),                                    % supine verb form
  adjacent([p(X,Y)],[p(C,D)]),              % adjacency, but either order
  append([p(X,Y)],[p(C,D)],Path),
  msort(Path, Sorted),
  constraint([type:supine,lex:Lex,kind:std],Supine),              % does not apply to deponent verbs
  ifthen(Tense=present,Tenseout=perfect),                         % working out tense assignment on the basis
                                                                  % of what we find in the 'esse'
  ifthen(Tense=imperfect,Tenseout=pluperfect),
  ifthen(Tense=future,Tenseout=future_perfect),

  map(vgpos,[cat:vg,type:Type,pathlist:Path,hp:[p(C,D)],lex:Lex,
             person:3,mood:Mood,tense:Tenseout,
             voice:middle,number:sing,gender:neuter,w:3])].          % middle voice

% with gerunds
% 2. insaniendum est (also with deponent verbs: hortandum est)

[verb,vg5amiddle1] --->
[ mapped(v,[from:X, to:Y|FSverb]),
  constraint([lex:esse,type:Type, person:3, number:sing,tense:Tense,mood:Mood],FSverb),      % esse
  mapped(v,[from:C, to:D|Gerund]),                                    % gerund
  adjacent([p(X,Y)],[p(C,D)]),
  append([p(X,Y)],[p(C,D)],Path),
  msort(Path, Sorted),
  constraint([case:acc,type:gerund,lex:Lex,kind:Kind],Gerund),         % accusative form of the gerund
  map(vgpos,[cat:vg,type:Type,pathlist:Path,hp:[p(C,D)],lex:Lex,
             person:3,mood:Mood,tense:Tense,
             voice:middle,number:sing,gender:neuter,              % middle voice
             value:obligation,w:3])].                          % semantic force : obligation
```

Relative Clauses

We deal with relative clauses in a way that may prove somewhat surprising to a linguist not used to working with *indices* and *gaps*. Consider a relative clause from which the relative pronoun has been removed:

(qui relinquit reginam / quem relinquit rex) → *relinquit reginam / relinquit rex*

we can look at such structures as a *pred-arg nexus missing an argument*, subject in the first case, object in the second (this is not the only reading, it's the reading that we give these structures when we know that they are incomplete, that they miss something). The missing arg is of course the relative pronoun, but the relative pronoun on its own does not give sufficient information to guarantee that all the constraints on the missing arg are met. The relative pronoun must be put into relation with its antecedent, and then we can complete the checking of a number of constraints, for instance of a semantic nature. The relation between antecedent and pronoun, and thereby between the antecedent and the gap in the incomplete predication discussed above, is implemented by *index sharing*. The index is no more than a pointer and can be coded in Prolog by a one argument structure, *i(Index)*, where *Index* is a variable to be shared by all instances of an index pointing to the same thing. The variable *Index* would then be shared by the triplet *missing arg/relative pronoun/antecedent*. We can increase the readability of our parses if we use a pair of values instead of the *Index* variable. This pair of values can be assigned as index each time we posit a noun phrase head: the values will be the start and end positions of the NP head in the string. To give an undoubtedly welcome example, in the sentence

rex qui relinquit reginam malus est

rex will be assigned *i(0,1)* as an index, which will come to be shared by *qui* and the missing arg of the *relinquit*-arg nexus (where it is often known as a *trace* or a *gap*). If *relinquit* needs a human subject (it does!), the constraint will be placed on the gap, passed on to the relative and to its antecedent, where it will be found to be satisfied.

The assigned parse follows:

vg
selected_reading:sum_be EST
polarity:pos
cat:vg
pos:v
lex:esse
voice:act
tense:present
mood:indicative
number:sing
person:3
subject
cat:np
index:i(p(0,1)) **REX**
number:sing
gender:masc
sem:[hum]
person:3
case:nom
lex:**rex**
c_str
head
rex
rel_clause
index:i(p(0,1)) **QUI**
number:or([sing,pl])
gender:masc
case:nom
mood:indicative
tense:present
c_str
vg
selected_reading:relinquo_leave RELINQUIT
polarity:pos
cat:vg
pos:v
lex:relinquere
voice:act
tense:present
mood:indicative
number:sing
person:3
subject **pointer to REX**
index:i(p(0,1))
object
index:i(p(3,4))
cat:np
sem:[hum]
number:sing
person:3
gender:fem
lex:regina REGINAM
case:acc
predicative
cat:adjp
case:nom
number:sing
gender:masc
lex:malus MALUS

We will briefly go into the index assignment and index sharing procedure, a procedure considerably facilitated by Prolog unification.

First, whenever we build an np, we assign an index corresponding to the path of its head, as in (the simplest possible case of an np built out of a single noun):

```
[core,np1] --->
[mapped(noun,[from:A,to:B|FS]),
 constraint([pos:noun,lex:Lex,class:common,sem:Sem,txt:Text,
          number:Nb,gender:G,case:C],FS),
 map(np,[pathlist:[p(A,B)],hp:[p(A,B)],index:i(p(A,B)),distance:[0],
         cat:np,sem:Sem,class:common,lextype:full,
         number:Nb,person:3,gender:G, type:core,lex:Lex,txt:Text,
         case:C,w:1])].
```

Second, we map relative clauses as consisting of a relative pronoun followed by a predication which misses an argument, and we relate the missing argument to the relative (in the code given below, the relative pronoun is subject, as in *uir qui epistulas ad Marcum misit*)

```
% Recall that the function of the np is independent from its function in the relative clause:
% "liber quem  rex legit ..." : liber is subject in the main clause and quem is object in the relative
% The index is shared; it reports the positions spanned by the np.
% A relative clause is an S displaying a [gap:Gap] feature corresponding to the antecedent:
% same category (np, pp) and shared index
% The gap site can specify any type of constraints on the constituent structure of the antecedent NP;
% this power is necessary to deal with mwus where the deletion site
% can point to an NP that must be lexically described,
% not just in terms of features such as number and broad semantic category


%% with a relative pronoun filling an np slot
% "(vir) qui  epistulas ad Marcum misit" ; "(librum) quem ancilla legit"

% subject

[finite,rel_clause_1] --->
[mapped(relative,[from:X,to:Y|FS1]),                                  % a relative pronoun

 constraint([number:Nb, case:Case,gender:Gender, function:Functions],FS1),

 mapped(pred,FS2),                                                    % a clause

 constraint([type:finite,                                             % relative clauses are finite
          mood:Mood, tense:Tense,pathlist:Pathlist,distance:Distance,w:Weight],FS2),

 msort(Pathlist,Sorted),
 start(Sorted,Y),
 contiguous(Sorted),                  % the relative clause cannot bind structures outside of itself
                                      % this restriction is important
                                      % strict contiguity seems to be called for

 constraint([gap:GAPARG,c_str:C_str],FS2),     % there MUST be a gap in the clause

 nonvar(GAPARG),                              % the built-in predicate nonvar checks that its argument
                                              % is an instantiated (bound) variable when the call to
                                              % nonvar is made
                                              % without such a test the danger is that the following test
                                              % be no test at all, but should merely result in unification
                                              % of the GAPARG variable with whatever comes its way
```

```
GAPARG=[gap:[type:np,                    % GAP specifies type
        index:Index,                % opens a place for the Index of the antecedent to fill
        function:subject,            % function must be compatible with relative pronoun
        subject:[e:Index],           % info for the parse tree - e (empty, trace) followed by the Index
        constraints:Constraints        % we put the constraints to be checked on the antecedent in this box
                ]],


 member(subject,Functions),         % remember that the acc rel pronouns must bear the 'subject' function as well as the
                                    % object one

 append([p(X,Y)],Pathlist,Pnew),              % appending the relative pronoun to the path

 map(relative_clause,[pathlist:Pnew,
                distance:Distance,
                gap:GAPARG,            % gap info carried by the clause
                index:Index,
                number:Nb,
                gender:Gender,
                case:Case,
                type:finite,
                mood:Mood,
                tense:Tense,
                constraints:Constraints,
                w:Weight,
                c_str:C_str]) ].

% similar treatment for the other functions (object, etc.)
```

We should now look at the building of a gapped predication. Whenever we are trying to match an argument with a phrase in the string to be parsed, we allow that argument to be matched with nothing at all in the string, with the proviso that it create a GAP, a structure which houses the requirements that the arg would have filled if it had been found in the string. Again, we look at the subject in finite clauses:

```
% in finite clauses
% the subject is the one of the potentially main clause, i.e. the 'up' one
% rex qui // [subject-gap] amat reginam

match(subject:Specs,
      subject:[index:Indexup],
      [number:Nsubj,gender:Gendersubj,person:Psubj,index:Indexup],
      [],
      [0],
      finite,                                                    % important !
      gap:[gap:[type:Type,
            index:Indexup,
            function:subject,
            subject:[e:Indexup],
            constraints:GapConstraints]],
      w:1,
      Int) :-

 constraint([type:Type,constraints:Constraints],Specs),
 Type \= dummy,
 append([number:Nsubj,gender:Gendersubj,person:Psubj,case:nom,index:Indexup],Constraints,GapConstraints).

% remember that a gapped constituent simply puts its constraints in the Gap feature,
% to be satisfied when the pred is connected to the antecedent
```

It remains for us to look at how the junction between the relative clause and its antecedent is accomplished. The index of the np is projected into the relative clause; the constraints housed in the relative clause should be met, with the exception of case, which need not be shared between antecedent and relative pronoun (and thereby the GAP structure of the relative).

```
% NP WITH RELATIVE CLAUSE
[finite,np6] --->

[                                                    % the NP
 mapped(np,FS1),
 constraint([pathlist:PL1,hp:HL1,distance:[Distnp1]],FS1),
 constraint([cat:np,index:Index,number:Nb,gender:G,
          sem:SemNP,lex:Lex, case:Case],FS1),

                                                     % the relative clause
 mapped(relative_clause,FS2),
 constraint([number:Nb,gender:G,pathlist:PL2,distance:[Distrel], constraints:Constraints,w:W],FS2),

 constraint([index:Index],FS2), % index sharing with the NP - essential to link relative and antecedent

 cleanc(Constraints,CC),
                  % the Constraints should not include Case (to be removed - other constraints to be kept)
 ifthen(CC\=[],constraint(CC,FS1)),
                  % apply the constraints, e.g. semantic constraints passed on to the antecedent noun

 msort(PL1,PLnpSorted),
 extremity(PLnpSorted,X),          % contiguity test np and rel clause
 succ(X,Xplus),                    % successor function (succ(X,Xplus) is equivalent to Xplus is X+1
 (start(PL2,X); start(PL2,Xplus)), % room for only one word to fit between antecedent and relative clause

 append(PLnpSorted,PL2,PL),
 msort(PL, Sorted),
  \+dup(Sorted),
 % contiguous(Sorted),         % not applicable on account of possible non-contiguity in the NP constituents
                                % although at first sight the restriction looks reasonable... but:
                                % imperatores timeo qui a pace abhorrent

                               % Distance is Distnp1+Distrel,
                              % Weight is W+1,
 myplus(Distnp1,Distrel,Distance),
 myplus(W,1,Weight),

 map(np,[pathlist:Sorted, hp:HL1,distance:[Distance],
         cat:np,type:full,class:common,lextype:full,
         index:Index,number:Nb,gender:G,sem:SemNP,person:3,case:Case,
         lex:Lex,w:Weight,c_str:[head:Lex,rel_clause:FS2]])
].
```

Prioritizing Subject-object Order in Accusative-cum-infinitive Clauses

In nonfinite dependent clauses, namely in accusative-cum-infinitive clauses, we might encounter accusative pairs, and even accusative triplets in the case of double-accusative verbs of the *doceo*-type:

Putabas ***reginam regem*** *amare.*
Putabas ***ancillam pueros grammaticam*** *docere.*

The accusatives can play the parts of subject, object, and even indirect object in the argument structure of *doceo*-type verbs.

Suppose we wish to prioritize the standard order of args in the clause, with the subject preceding the object(s), without rejecting readings in which the subject follows either or both of the objects:

Putabas reginam regem amare:
Preferred reading: *reginam* subject, *regem* object:
You thought the queen loved the king.
Deprioritized but possible reading: *regem* subject, *reginam* object:
You thought the king loved the queen.

Putabas ancillam pueros grammaticam docere.
Preferred reading: *ancillam* subject, *pueros* indirect object, *grammaticam* object:
You thought the servant was teaching the kids grammar.

Deprioritized but possible reading:
ancillam indirect object, *pueros* subject, *grammaticam* object:
You thought the kids were teaching the servant grammar.

Rejected readings (on semantic grounds: both the subject and indirect object must bear the feature +HUM):
grammaticam subject, *pueros* indirect object, *ancillam* direct object:
** You thought the grammar was teaching the kids the servant.*
grammaticam subject, *pueros* object, *ancillam* indirect object:
**You thought the grammar was teaching the servant the kids.*

The parser should come up, and does come up, with the following ranking of the two retained parses:

[0/putabas,1/ancillam,2/pueros,3/grammaticam,4/docuisse,endpos(5)]

5--->
lex:puto_think_that
cat:vg
polarity:pos
pos:v
lex:putare
tense:imperfect
mood:indicative
number:sing
person:2
subject
source:context_retrievable
number:sing
person:2
index:i(_G3839)
constraints_to_be_met:[sem:[hum]]
case:nom
object
cat:pred
number:sing
case:or([nom,acc])
person:3
i_ss:i(p(1,2))
c_str
lex:doceo_teach
cat:vg
polarity:pos
pos:v
lex:docere
tense:past
mood:infinitive
subject
index:i(p(1,2))
cat:np
sem:[hum]
number:sing
person:3
gender:fem
lex:ancilla
case:acc
c_str
head
pos:noun
lex:ancilla
case:acc
gender:fem
number:sing
sem:[hum]
object
index:i(p(3,4))
cat:np
sem:[abstract]
number:sing
person:3
gender:fem
lex:grammatica
case:acc

c_str
head
pos:noun
lex:grammatica
case:acc
gender:fem
number:sing
sem:[abstract]
i_object
index:i(p(2,3))
cat:np
sem:[hum]
number:pl
person:3
gender:masc
lex:puer
case:acc
c_str
head
pos:noun
lex:puer
case:acc
gender:masc
number:pl
sem:[hum]

4--->

- lex:puto_think_that
  - cat:vg
  - polarity:pos
  - pos:v
  - lex:putare
  - tense:imperfect
  - mood:indicative
  - number:sing
  - person:2
  - subject
    - source:context_retrievable
    - number:sing
    - person:2
    - index:i(_G4775)
    - constraints_to_be_met:[sem:[hum]]
    - case:nom
  - object
    - cat:pred
    - number:sing
    - case:or([nom,acc])
    - person:3
    - i_ss:i(p(2,3))
    - c_str
      - lex:doceo_teach
        - cat:vg
        - polarity:pos
        - pos:v
        - lex:docere
        - tense:past
        - mood:infinitive
        - subject
          - index:i(p(2,3))
          - cat:np
          - sem:[hum]
          - number:pl
          - person:3
          - gender:masc
          - lex:puer
          - case:acc
          - c_str
            - head
              - pos:noun
              - lex:puer
              - case:acc
              - gender:masc
              - number:pl
              - sem:[hum]
        - object
          - index:i(p(3,4))
          - cat:np
          - sem:[abstract]
          - number:sing
          - person:3
          - gender:fem
          - lex:grammatica
          - case:acc
          - c_str
            - head
              - pos:noun
              - lex:grammatica
              - case:acc
              - gender:fem

number:sing
sem:[abstract]
i_object
index:i(p(1,2))
cat:np
sem:[hum]
number:sing
person:3
gender:fem
lex:ancilla
case:acc
c_str
head
pos:noun
lex:ancilla
case:acc
gender:fem
number:sing
sem:[hum]

It should be borne in mind that the accusative-cum-infinitive construction might be found inside a relative clause: in such cases, a correct parsing procedure must identify the nature of the trace (gap) in the relative clause, and ensure the proper indexing of it by index-sharing between antecedent and trace.

Consider:
Amas grammaticam quam putabas ancillam pueros docuisse.

In the accusative-cum-infinitive clause [*ancillam pueros docuisse*] inside the relative clause [*quam putabas ancillam pueros docuisse*] we need to posit a gap (symbolized in the parse tree with an *e* for *empty*) for the object of *docuisse* and associate that gap, by co-indexing, with the antecedent of the relative, i.e. *grammaticam*.

We do this by assigning an index referring to the position occupied by the antecedent in the word list derived from the input string:

[0/amas,**1/grammaticam,2**/quam,3/putabas,4/ancillam,5/pueros,6/docuisse,endpos(7)]

The antecedent *grammaticam* spans from 1 to 2: *p(1,2).*
The index for the missing arg within the infinitive clause (the object arg) must therefore bear the index *p(1,2)*, as it does in the parses returned by the parser. Once again, the parse tree where the order subject-object is maintained is prioritized

5--->
lex:amo_love
cat:vg
polarity:pos
pos:v
lex:amare
tense:present
mood:indicative
number:sing
person:2
subject
source:context_retrievable
number:sing
person:2
index:i(_G1737)
constraints_to_be_met:[sem:[hum]]
case:nom
object
cat:np
index:i(p(1,2))
number:sing
gender:fem
sem:[abstract]
person:3
case:acc
lex:grammatica
c_str
head
head
pos:noun
lex:grammatica
case:acc
gender:fem
number:sing
sem:[abstract]
rel_clause
index:i(p(1,2))
number:sing
gender:fem
case:acc
c_str
lex:puto_think_that
cat:vg
polarity:pos
pos:v
lex:putare
tense:imperfect
mood:indicative
number:sing
person:2
subject
source:context_retrievable
number:sing
person:2
index:i(_G2272)
constraints_to_be_met:[sem:[hum]]
case:nom
object
cat:pred
number:sing
case:or([nom,acc])

person:3
i_ss:i(p(4,5))
c_str
  lex:doceo_teach
    cat:vg
    polarity:pos
    pos:v
    lex:docere
    tense:past
    mood:infinitive
    subject
      index:i(p(4,5))
      cat:np
      sem:[hum]
      number:sing
      person:3
      gender:fem
      lex:ancilla
      case:acc
      c_str
        head
          pos:noun
          lex:ancilla
          case:acc
          gender:fem
          number:sing
          sem:[hum]
    object
      e:i(p(1,2))
    i_object
      index:i(p(5,6))
      cat:np
      sem:[hum]
      number:pl
      person:3
      gender:masc
      lex:puer
      case:acc
      c_str
        head
          pos:noun
          lex:puer
          case:acc
          gender:masc
          number:pl
          sem:[hum]

4--->

```
lex:amo_love
   cat:vg
   polarity:pos
   pos:v
   lex:amare
   tense:present
   mood:indicative
   number:sing
   person:2
   subject
      source:context_retrievable
      number:sing
      person:2
      index:i(_G445)
      constraints_to_be_met:[sem:[hum]]
      case:nom
   object
      cat:np
      index:i(p(1,2))
      number:sing
      gender:fem
      sem:[abstract]
      person:3
      case:acc
      lex:grammatica
      c_str
         head
            head
               pos:noun
               lex:grammatica
               case:acc
               gender:fem
               number:sing
               sem:[abstract]
         rel_clause
            index:i(p(1,2))
            number:sing
            gender:fem
            case:acc
            c_str
               lex:puto_think_that
                  cat:vg
                  polarity:pos
                  pos:v
                  lex:putare
                  tense:imperfect
                  mood:indicative
                  number:sing
                  person:2
                  subject
                     source:context_retrievable
                     number:sing
                     person:2
                     index:i(_G980)
                     constraints_to_be_met:[sem:[hum]]
                     case:nom
                  object
                     cat:pred
                     number:sing
                     case:or([nom,acc])
                     person:3
```

i_ss:i(p(5,6))
c_str
  lex:doceo_teach
    cat:vg
    polarity:pos
    pos:v
    lex:docere
    tense:past
    mood:infinitive
    subject
      index:i(p(5,6))
      cat:np
      sem:[hum]
      number:pl
      person:3
      gender:masc
      lex:puer
      case:acc
      c_str
        head
          pos:noun
          lex:puer
          case:acc
          gender:masc
          number:pl
          sem:[hum]
    object
      e:i(p(1,2))
    i_object
      index:i(p(4,5))
      cat:np
      sem:[hum]
      number:sing
      person:3
      gender:fem
      lex:ancilla
      case:acc
      c_str
        head
          pos:noun
          lex:ancilla
          case:acc
          gender:fem
          number:sing
          sem:[hum]

It might be worthwhile taking a quick look at the way the Prolog program embodying the parser deals with the subject-object order prioritization in the accusative-cum-infinitive construction:

```
% prioritize normal subject-object order

 ifthenelse(constraint([subject:[hp:Pathsubj]],ST),
                                        % we have a non-gapped subject, we record its head path
          true,                         % and abstain from doing anything else
          Pathsubj=[p(0,0)]),
                    % otherwise the subject is higher up and therefore necessarily precedes

 ifthenelse(constraint([object:[hp:Pathobj]],ST),      % IF-CLAUSE we have an object
        ifthenelse(precedes(Pathsubj,Pathobj), NW is Weight+1, NW=Weight),
                                                % THEN-CLAUSE
        NW=Weight),                                     % ELSE-CLAUSE

 ifthenelse(constraint([i_object:[hp:Pathiobj]],ST),          %  we have an indirect object
        ifthenelse(precedes(Pathsubj,Pathiobj), NW1 is NW+1, NW1=NW),
        NW1=NW),
```

To understand this bit of code, one needs to know that the **hp** (HeadPathList) feature above records the positions spanned by the head of the noun phrase filling in the arg position of the parent feature: subject, object, or indirect object.

Remember that if the subject is gapped, it does not have a **hp** feature since it does not occur in the clause: there is no subject present in [*epistulam misisse Marco*] as object of *putas* in [*putas epistulam misisse Marco*], itself included in the relative clause [*quem putas epistulam misisse Marco*] appended to the antecedent *rex* in [*Rex quem putas epistulam misisse Marco*] which functions as subject of the whole S(entence): *[Rex quem putas epistulam misisse Marco] amat ancillam reginae.*

In the case of such a gapped subject we assign it a dummy headpathlist, namely [p(0,0)], which is sure to precede the pathlist of any arg found in the clause.
Otherwise we record the value of the *hp* feature and we can then compare it with the values of the headpathlists for the object and indirect object, if available, and proceed to the prioritizing by increasing the current Weight assigned to the parse tree being built.

We have seen that the *precedes* predicate is trivially simple to code; we know that the *constraint* predicate implements feature unification. It is used here to retrieve values for features possibly instanciated in the feature bundle *ST*, which records the structure and properties of the arguments filling in the arg slots opened up by the predicate.

Binding SE

The issue is well-known. The SE family (*se*, *sui*, *sibi*) offers the possibility of multiple binding: a local binding in the clause of which SE is a constituent, and outer bindings in higher clauses, i.e. clauses having the SE-bearing clause as an argument or as the argument of one of their arguments, and so on climbing up the parse tree.

We wish to insist on one point: reference belongs to discourse. The only thing grammar can do is to suggest candidates and provide limits to the exploration of referents.

Remember that in ALP each *NP* is associated with an *index* which records the *string position* of its *head*. In the string we are going to parse, namely

Romani sciunt
regem credere
reginam putare
se a Venere amatum iri.

The np heads *Romani*, *regem*, *reginam* et *Venere* will be assigned an index. Since the process turning the string into a wordlist yields

[0/romani,1/sciunt,2/regem,3/credere,4/reginam,5/putare,6/se,7/a,8/uenere,9/amatum,10/iri],

the indices will be the following:
romani i(0,1) *regem* i(2,3) *reginam* i(4,5) *uenere* i(8,9).

The reflexive pronoun SE (and members of its family such as *sibi*), instead of having an index reflecting its position in the string (here, it would be i(6,7)), is assigned an index waiting to be bound: *index:Index*.

The binding of this index is meant to reflect the assignment of referents to the reflexive pronoun. The binding will result in the production of competing parses, where the index is bound locally, i.e. within the clause of which SE is a constituent, or externally (coming to be bound with a constituent higher up in the hierarchy).

The binding cannot take place until we have available a complete parse. We then examine its structure and proceed to the binding(s).

In our example string, the possible referents are *romani*, *regem* and *reginam* (the first is nominative, and the other two accusative; *venere* is not a subject, and is therefore excluded). The three candidates pass a structural test (nphood) and a semantic test (+HUM) and are 'positionally' OK, i.e. are at the right level and have been matched as candidate binding arguments.

Looking at the parses delivered by ALP, we note that SE is subject of the infinitive (future and passive voice) and is assigned different indices in the top parsings: i(p(4,5)),i(p(2,3)),i(p(0,1)) pointing to *reginam, regem* and *romani* as candidates for the referent of the subject. The parse is to be found below.

Once again, parses should not be rejected on the basis of the candidates they put forward for reference. We need other tools to tackle the issue, which is only partly a matter for linguistics to handle, and certainly not one to be exclusively assigned to the syntactic component that is the core business of parsers.

vg
selected_reading:scio_know_that
polarity:pos
cat:vg
pos:v
lex:scire
voice:act
tense:present
mood:indicative
number:pl
person:3
subject
index:i(p(0,1))
cat:np
sem:[hum]
number:pl
person:3
gender:masc
lex:romanus
case:nom
object
cat:pred
mood:infinitive
tense:present
number:sing
gender:neuter
case:or([nom,acc])
person:3
polarity:pos
argbound:no
flagint:no
c_str
vg
selected_reading:credo_believe_that
polarity:pos
cat:vg
gender:masc
pos:v
lex:credere
voice:act
tense:present
mood:infinitive
subject
index:i(p(2,3))
cat:np
sem:[hum]
number:sing
person:3
gender:masc
lex:rex
case:acc
object
cat:pred
mood:infinitive
tense:present
number:sing
gender:neuter
case:or([nom,acc])
person:3
polarity:pos
argbound:no
flagint:no
c_str
vg
selected_reading:puto_think_that
polarity:pos
cat:vg
gender:fem
pos:v

lex:putare
voice:act
tense:present
mood:infinitive
subject
index:i(p(4,5))
cat:np
sem:[hum]
number:sing
person:3
gender:fem
lex:regina
case:acc
object
cat:pred
mood:infinitive
tense:future
number:sing
gender:neuter
case:or([nom,acc])
person:3
polarity:pos
argbound:no
flagint:no
c_str
vg
selected_reading:amo_love
polarity:pos
cat:vg
lex:amare
mood:infinitive
tense:future
voice:pass
gender:or([masc,fem])
subject
index:i(p(4,5)) **(/ index:i(p(2,3)) / index:i(p(0,1))** in the other 2 top parses**)**
cat:np
sem:[hum]
lex:pp3refl
number:or([sing,pl])
person:3
gender:or([masc,fem])
case:acc
agent
index:i(p(8,9))
case:abl
prep:ab
sem:[hum]
lex:uenus
cat:pp
c_str
prep:ab
head
index:i(p(8,9))
cat:np
sem:[hum]
lex:uenus
number:sing
person:3
gender:fem
case:abl

We should add that in the current version of ALP we use a somewhat more sober approach: we bind SE locally, unless there is a governing clause above the one in which SE occurs, in which case we bind 'one up', i.e. to the subject of the governing clause. In *laudant se reges*, we bind *se* to *reges*; in *putat rex reginam se laudare*, we bind *se* to *rex* (preferably as object, due to the heavier weight given to Subject-Object order):

Laudant se reges:

vg
selected_reading:laudo_praise
polarity:pos
cat:vg
pos:v
lex:laudare
voice:act
tense:present
mood:indicative
number:pl
person:3
subject
number:pl
gender:masc
index:i(p(2,3))
cat:np
sem:[hum]
person:3
lex:rex
case:nom
object
number:**pl**
gender:**masc**
index:i(p(2,3))
cat:np
sem:[hum]
lex:pp3refl
person:3
case:acc

Rex putat reginam se laudare:

vg
selected_reading:puto_think_that
polarity:pos
cat:vg
pos:v
lex:putare
voice:act
tense:present
mood:indicative
number:sing
person:3
subject
number:sing
gender:masc
index:i(p(0,1))
cat:np
sem:[hum]
person:3
lex:rex
case:nom
object
cat:pred
mood:infinitive
tense:present
number:sing
gender:neuter
case:or([nom,acc])
person:3
polarity:pos
argbound:no
flagint:no
c_str
vg
selected_reading:laudo_praise
polarity:pos
cat:vg
gender:fem
pos:v
lex:laudare
voice:act
tense:present
mood:infinitive
number:sing
subject
number:sing
gender:fem
index:i(p(2,3))
cat:np
sem:[hum]
person:3
lex:regina
case:acc
object
index:i(p(0,1))
cat:np
sem:[hum]
lex:pp3refl
number:or([sing,pl])
person:3
gender:or([masc,fem])
case:acc

In the case of

rex putat reginam se ipsam laudare

we get the following subject-object pairing

subject
number:sing
gender:fem
index:i(p(2,3))
cat:np
sem:[hum]
person:3
lex:regina
case:acc
object
index:i(p(2,3))
cat:np
sem:[hum]
lex:pp3refl
emphasis:yes
number:sing
person:3
gender:fem
case:acc

where the emphasis due to the *ipsam* prevented the binding with masculine *rex.*

Weighting

We wish to stress that a weighting process is absolutely necessary for parsers. The parser is first and foremost a tool that yields structural descriptions of strings on the basis of what the linguist has specified as grammatical. If we do not build a weighting procedure on top, we will be presented with multiple parses of strings, all of them grammatical with respect to our grammar, although some of them may look very far-fetched, and, to put it bluntly, totally unacceptable as plausible readings of the string submitted to parsing. Consider the following example : *Amo magistros cupidos legendae historiae*, *I like teachers who are eager to read history (books)*.

The parse which gets the highest ranking in ALP is the natural one, in fact the only one that comes to mind when we read the Latin sentence (and the only one we expect the learner to work out): the sentence is made up of a predicate, *amo*, with first-person subject immediately derivable from the verb form; the predicate is transitive *amo*, which in the sentence has as object the noun phrase *magistros cupidos legendae historiae*, which is made up of a head, *magistros*, in the accusative as it should be, and an adjective phrase attached to it, namely *cupidos legendae historiae*, whose head, *cupidos*, is in its turn in the right case, gender and number. *Cupidus* is an argument-bearing adjective, its argument being a genitive phrase, noun phrase or gerund(ive) clause, as is the case here, the gerundive clause being *legendae historiae*, made up of a predicate, the gerundive *legendae*, and its argument, a noun phrase in the genitive case, *historiae*, *lego* being transitive just like the *amo* of two minutes ago. We have reached the end of the gerundive clause, the end of the argument of the adjective, the end of the noun phrase of which the adjective phrase is a part, the end of the argument of the main verb, the end of the sentence, the predicate having the two arguments it needs, a subject hidden in the verb form, and an object covering all the words except the prediate itself. Nothing could be simpler, there is no way of getting it wrong, and ALP certainly does not.

```
vg
    selected_reading:amo_love
    polarity:pos
    cat:vg
    pos:v
    lex:amare
    voice:act
    tense:present
    mood:indicative
    number:sing
    person:1
  subject
    source:context_retrievable
    number:sing
    gender:or([masc,fem])
    person:1
    cat:np
    index:i(0,0)
    constraints_to_be_met:[sem:[hum]]
    case:nom
  object
    index:i(p(1,2))
    cat:np
    sem:[hum]
    number:pl
    person:3
    gender:masc
    lex:magister
    case:acc
    c_str
      head
        pos:noun
        lex:magister
        case:acc
        gender:masc
        number:pl
```

```
    sem:[hum]
  adjp
    cat:adjp
    number:pl
    gender:masc
    lex:cupidus
    case:acc
    c_str
      cupidus
        object
          cat:pred
          subtype:gerundive
          mood:gerund
          local_case:gen
          number:sing
          person:3
          gender:neuter
          c_str
            vg
              selected_reading:lego_read
              pos:gdiv
              case:gen
              gender:fem
              number:sing
              lex:legere
              mood:gerund
              person:3
            object
              index:i(p(4,5))
              cat:np
              sem:[abstract]
              number:sing
              person:3
              gender:fem
              lex:historia
              case:gen
```

End of story? Well, there is Livy with *Pacis petendae oratores ad consulem miserunt* and Tacitus with *Germanicus Aegyptum proficiscitur cognoscendae antiquitatis*. And if we wish to account for the usage of our two historians, we need to make room for an adjunct of purpose built around a gerundive clause. And we run the risk of parsing our very simple sentence as meaning something along the lines of *I love greedy teachers in order to read history*.

In fact, there is no way of preventing the 'wrong' parse to come out, in so far as it is not a wrong parse at all – it is correct with respect to a grammar that is itself correct. What we can do to avoid the parse coming up to the surface is to deprioritize it, rank it down, or, what amounts to the same, prioritize what we regard as the natural parse, the one we have just shown to be the top choice of ALP.

The weighting procedure in ALP is based on two principles:

1) prefer tight links (such as that between a predicate and its arguments) over loose ones (an adjunct at clause level)
2) assign penalties to distorsions of the underlying word order (where the subject precedes the object) and, first and foremost, to strains due to the distance separating elements which are naturally found together, such as an adjective or genitive np and the noun functioning as head of the resulting noun phrase.

The above strategies need to be put to work with a certain amount of care, so that they should cooperate rather than compete. We have also seen that we need the path algorithms studied above to put the second of them into practice.

Features in ALP : a brief and partial survey

Lexical Features

Verbs

verb([v(conscire,4,consci,consciu,conscit)],tr_cod,std).
verb([v(cadere,3,cad,cecid,cas)],intr,std).
verb([v(canere,3,can,cecin,cant)],tr_cod,std).
verb([v(uti,3,ut,us)],tr_cod_abl,dep).

verb([v(Infinitive, Conjugation Class, RootPres, RootPerf, RootSup)], Class, Type)

tr_cod applies to transitives with direct object, which can undergo passivisation and form -ndum gerunds (*canitur a X*, *canendum*)

Type is *std* (standard) or *dep* (deponent)

Adjectives

adj(beatus,beat,1,std,n:hpl,[beat,beatissim],[]).
adj(cupidus,cupid,1,std,n:hpl,[cupid,cupidissim],[hum]).
adj(facilis,facil,2,std,n:no,[facil,facillim],[]).

adj(Nominative masc sg, Root, Class, n feature, [Root Comp, Root Sup], [Semantics])

The *n:Value* feature is meant to capture the availability of the adj as a noun :

n:no - cannot be read as a noun e.g. *aureus*
n:yes - can always be read as noun e.g. *clarus*
n:n - only neuter gender readable as noun: *uerum* / *uera*
n:hpl - readable as a noun if both plural and with human reference : *boni*

Nouns

noun(1,fem,copiae,copi,class:common, ab:mm, sem:[hum],[nb:pl]). % pluralia tantum
noun(1,fem,ecclesia,ecclesi,class:common, ab:no, sem:[hum, thing,loc],[]).
noun(2, masc, morbus, morb, class:common, ab:mm, sem:[abstract, thing],[]).
noun(2, neuter, beneficium, benefici,class:common, ab:mm, sem:[abstract, thing],[]).
noun(3,masc,orator,orator,um,class:common, ab:no, sem:[hum],[]).
noun(4,masc,metus,met,class:common, ab:mm, sem:[abstract],[]).
noun(1,masc,catilina,catilin,class:proper,ab:no, sem:[male],[nb:sg]).

noun(Declension class, gender, nominative, root, class, ab feature, semantics, restriction on number.

The *ab* feature specifies use as adjunct in the abl without prep (*mm* : manner_means : *morbo*).
Names have got *proper* as value for the class feature, standard nouns have *common*

Pronouns (and related adjectives)

lex(nos, prpers,[pos:prpers,txt:nos,lex:pp1pl,
number:pl, person:1, gender:or([masc,fem]),
case:acc, sem:[hum]]).

lex(Textual form, POS, Feature List)

The Feature List is an open structure, being a list (i.e. it does not have to have the same number of elements throughout) It includes features for POS, Textual Form, lex value (compacted info on features, such as here *pp1pl*, personal pronoun first person plural), number, person, gender, case, semantics, etc.

Examples :

QUAE, an overloaded relative pronoun:

lex(quae, relative, [pos:relative, txt:quae, lex:relnomfemsing,gender:fem,
case:nom, number:sing, function:[subject]]).
lex(quae, relative, [pos:relative, txt:quae, lex:relnomfempl,gender:fem,
case:nom, number:pl, function:[subject]]).
lex(quae, relative, [pos:relative, txt:quae, lex:relnomneuterpl,gender:neuter,
case:nom, number:pl, function:[subject]]).
lex(quae, relative, [pos:relative, txt:quae, lex:relaccneuterpl,gender:neuter,
case:acc, number:pl, function:[object]]).
lex(quae, relative, [pos:relative, txt:quae, lex:relaccneuterpl,gender:neuter,
case:acc, number:pl, function:[subject]]).

UTRUMQUE

lex(utrumque, adj, [pos:adj, txt:utrumque, lex:uterque,gender:neuter,type:tool,
case:nom, number:sing]).
lex(utrumque, adj, [pos:adj, txt:utrumque, lex:uterque,gender:neuter,type:tool,
case:acc, number:sing]).
lex(utrumque, adj, [pos:adj, txt:utrumque, lex:uterque,gender:masc,type:tool,
case:acc, number:sing]).

MULTI, MULTAE, MULTA as pronoun

l*ex(multi, prindef, [pos:prindef, txt:multi, lex:multi,gender:masc,person:3,sem:[hum],*
case:nom, number:pl, function:[subject]]).
lex(multae, prindef, [pos:prindef, txt:multae, lex:multi,gender:fem,person:3,sem:[hum],
case:nom, number:pl, function:[subject]]).
lex(multa, prindef, [pos:prindef, txt:multa, lex:multi,gender:neuter,person:3,sem:[thing],
case:nom, number:pl, function:[subject]]).
lex(multa, prindef, [pos:prindef, txt:multa, lex:multi,gender:neuter,person:3,sem:[thing],
case:acc, number:pl, function:[subject]]).
lex(multa, prindef, [pos:prindef, txt:multa, lex:multi,gender:neuter,person:3,sem:[thing],
case:acc, number:pl, function:[object]]).

QUIBUS as interrogative pronoun

lex(quibus, print, [pos:print, txt:quibus, lex:quis,gender:or([masc,fem,neuter]),
case:dat, number:pl,person:3,sem:_,
function:[i_object]]).
lex(quibus, print, [pos:print, txt:quibus, lex:quis,gender:or([masc,fem,neuter]),
case:abl, number:pl,person:3,sem:_,
function:[prep_cplt]]).

Note the use of the OR operator in feature values: or([List_of_Possible_Values]) and of the *anonymous variable (_)* to indicate that any value is to be accepted.

Prepositions

lex(a, prep,[lex:ab, pos:prep, requires:abl, gerund:no, type:pre]).
lex(ab, prep,[lex:ab, pos:prep, requires:abl, gerund:no, type:pre]).
lex(abs, prep,[lex:ab, pos:prep, requires:abl, gerund:no, type:pre]).
lex(ad, prep,[lex:ad, pos:prep, requires:acc, gerund:yes,type:pre]).

lex(Textual Form, POS, List of Features (includes lexeme, POS, Required Case, availability for gerunds, type (*pre* or *post* governed np : think of *causa*))

Adverbs

lex(ceterum, adv,[lex:ceterum, pos:adv,type:clausal, sem:discourse]).
lex(cotidie, adv,[lex:cotidie, pos:adv,type:vpbound, sem:time]).
lex(cras, adv,[lex:cras, pos:adv,type:vpbound, sem:time]).

lex(Text Form, POS, Feature List (includes preferred attachment – vp or whole clause – and semantics))

Subordinators

lex(quoniam, sub,[lex:quoniam, pos:sub, argbound:no, mood:_, value:reason]).
lex(quia, sub,[lex:quia, pos:sub, argbound:no, mood:_, value:reason]).
lex(quod, sub,[lex:quod, pos:sub, argbound:no, mood:_, value:reason]).
lex(quando, sub,[lex:quando, pos:sub, argbound:no, mood:_, value:reason]).
lex(quin, sub,[lex:quin, pos:sub, argbound:yes, mood:subjunctive]).
lex(quominus, sub,[lex:quominus, pos:sub, argbound:yes, mood:subjunctive]).

Note the *argbound* feature, with values *yes*/*no*, to specify whether the subordinate clause can be read as argument. *Mood* and *value* are straightforward. As the feature list is a list, it can include features that are sometimes specified and sometimes not. The *value* feature is a case in point. The *mood* feature has to be present (for use by the *constraint* predicate) but the value can be left open by means of the *anonymous variable*.

Particles

lex(ne_int, part, [lex:ne_int, type:int, value:open_orientation, clausetype:_]).
lex(num, part, [lex:num, type:int, value:open_orientation, clausetype:sub]). % in indirect questions
lex(num, part, [lex:num, type:int, value:negative_orientation, clausetype:main]). % in direct questions
lex(nonne, part, [lex:num, type:int, value:positive_orientation, clausetype:_]). % in direct questions

Note the *ne_int* as textual form ; it is produced by the *word list maker,* which removes it from the element it is attached to (the process makes sure that any word ending in *-ne* is preserved as an alternative reading (*agmine*, *sine*, etc, etc. - see also the treatment of *que* and *ue*).

The *value* and *clausetype* features are straightforward.

Verb templates

The *lexarg* structures are essential to the working of ALP. Whenever an *arg-bearer* (all verbs, but also some nouns and adjectives) is encountered, ALP gets hold of its *arglist*, which is made up of *ws* (word sense) structures. Each word sense is specified by a short description in the guise of a rough English equivalent, followed by its transitivity class, restrictions that apply on the clause it is inserted in (most of the time the restriction list is empty, but see *dubitare* below), and its weight as potential multi-word unit. There follows a list of arguments as value of the *args* feature. Each argument is specified as to its grammatical function, its obligatory or optional character, with a list of constraints that the constituent building the argument must meet : the *type* is one such constraint, but there is also a specific *constraints list* which, being a list, is open as to number and nature of the individual constraints.

```
% REDDERE
lexarg(reddere,
      arglist:[ws(reddo_give_back,tr_cod,clause:[],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:np,oblig:yes,constraints:[case:acc]]]),

            ws(reddo_make,tr_cod_cplt,clause:[],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:np,oblig:yes,constraints:[case:acc,number:N,gender:G]],
                  object_cplt:[type:adjp,oblig:yes,constraints:[case:acc, number:N,gender:G]]])]).

% REGNARE
% Tres annos regnavit.
lexarg(regnare,
      arglist:[ws(regno_reign,intr,clause:[],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]]])]).


% UESPERASCERE
lexarg(uesperascere,
      arglist:[ws(dies_uesperascit_it_is_getting_dark,intr,clause:[],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[lex:dies]]])]).
```

```
% UOCARE
% Nauta rationes puellae in dubium uocat.
% Vocavit matrem eius et non uenit.
lexarg(uocare,
      arglist:[ws(uoco_call,tr_cod,clause:[],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:np,oblig:yes,constraints:[case:acc]],
                  object_cplt:[type:np,oblig:no,constraints:[case:acc]]]),

            ws(uoco_call,tr_cod,clause:[],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:np,oblig:yes,constraints:[case:acc]],
                  prep_cplt:[type:pp,oblig:yes,constraints:[prep:ad, case:acc]]]),


            ws(uoco_call,tr_cod,clause:[],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:np,oblig:yes,constraints:[sem:[hum],case:acc]]]),

       % mwu IN IUS VOCARE
          ws(mwu_in_ius_uoco_bring_to_court_IDIOM,tr_cod_cplt,clause:[],mwuw:2,
          args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
               object:[type:np,oblig:yes,constraints:[sem:[hum],case:acc]],
               object_cplt:[type:phrase,frozen:yes,oblig:yes,constraints:[lex:in_ius]]]),

       % mwu IN DUBIUM VOCARE
           ws(mwu_in_dubium_uoco_call_into_doubt_IDIOM,tr_cod_cplt,clause:[],mwuw:2,
           args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                object:[type:np,oblig:yes,constraints:[sem:[abstract],case:acc]],
                object_cplt:[type:phrase,frozen:yes,oblig:yes,constraints:[lex:in_dubium]]])]).


% DUBITARE
lexarg(dubitare,
      arglist:[ws(dubito_doubt,tr,clause:[[polarity:neg]],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:pred,oblig:yes,constraints:[type:finite,gap:[],mood:subjunctive,
                                    argbound:yes, subordinator:quin]]]),
            ws(dubito_hesitate,tr,clause:[[polarity:neg]],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:pred,oblig:yes,constraints:[type:nonfinite_i]]]),

            ws(dubito_doubt,tr,clause:[[illocutionary_force:question]],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:pred,oblig:yes,constraints:[type:finite,gap:[],mood:subjunctive,
                                    argbound:yes, subordinator:quin]]]),
            ws(dubito_hesitate,tr,clause:[[illocutionary_force:question]],mwuw:0,
            args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                  object:[type:pred,oblig:yes,constraints:[type:nonfinite_i]]])]).
```

General Features

Pathlist : the word-list maker, the Prolog subprocedure that turns an ASCII string into a wordlist, produces and records (with recorda) a number of *position* structures (*position* being a functor with arity three, i.e. with three arguments) in a storage space (a box) named *pos* :

recorda(pos,position(Pos,Posand1,NewHead),_),

It should be noted that in the above the small-letter *pos* is an atom (short for *position*), to be carefully distinguished from the variable *Pos*, which is meant to store the position reached in the string by the word-list maker (1, 2, 3, etc.)

A general lexical rule will use the records so produced to store the words under their respective POS (Part of Speech, this time), using the lexicon's *lex* clauses (mainly produced by *makelex.pl*), to get at the relevant POS.

In the *vocfile* (produced by *makelex*) we find such a clause as :

lex(ambularent, v, [pos:v, class:intr, type:finite, lex:ambulare, voice:act, txt:ambularent, tense:imperfect, kind:std, mood:subjunctive, number:pl, person:3]).

Using such clauses we map the words in the relevant Part of Speech boxes :

```
[lex,words] --->[recorded(pos,position(A,B,Word),_),
                 % we have the word form Word in the string, e.g. ambularent
                 lex(Word,Box,FS),
                 % we have a lex for the Word (ambularent) to put in the relevant Box (the v Box)
                 % accompanied by the Feature List (FS) stored in the lexicon
          map(Box,[from:A,to:B|FS])].

                  % we store its position there accompanied by the feature list FS (third arg of lex)
```

We can then use the stored record to build the *pathlist* ; in the simple case of a one word structure we use the *from:P1* and *to:P2* features to build a pathlist containing a single element : *pathlist: [p(P1,P2)]* as in the mapping of an adjective phrase made up of a single adjective :

lex(formosam, adj, [pos:adj, txt:formosam, lex:formosus, type:std, n:no, case:acc, gender:fem, number:sing, degree:pos, sem:[]]).

```
[adj,adjp1] --->
[mapped(adj,[from:A,to:B|FS]),
 constraint([case:C, number:N, gender:G, type:Type, lex:Lex],FS),
 map(adjp,[cat:adjp,pathlist:[p(A,B)],distance:[0],hp:[p(A,B)],
        case:C,number:N, gender:G,lex:Lex,type:Type,w:1,constituent_structure:Lex])].
```

When we build larger structures we join the pathlists, and perform the necessary checks on the resulting longer paths : computing distance, checking adjacency or near-adjacency, or conditioned near-adjacency, etc.

We also work out where the *head* lies and store the path pertaining to the head as value of the *hp* (head phrase) feature. In the case of a one word structure, *pathlist* and *hp* store the same value.

Note how the *constraint* predicate is used in the rule above to retrieve values for features stored in the lexicon : case, number, gender, etc are read off the lexical entry and can then be stored in the new structure to be mapped, namely the adjective phrase. We use *constraint* in order not to have to bother about the order the feature-value pairs are stored, nor about there being a fixed number of elements in the FS list. Using the straightforward unification clause :

FS = [pos:_, txt:_, lex:Lex, type:Type, n:_, case:C, gender:G, number:N, degree:_, sem:_]

is considerably more error-prone and the program should have to be revised when a feature is added to a standard feature list or deleted from it.

To see the pathlists at work we can look at the link between adj and noun ; or between two noun phrases, as is the case of an np resulting from a core np to which is joined an np in the gentive acting as 'complément déterminatif' to use the terminology prevalent in secondary school classes when I was a pupil struggling with Caesar and, later, Virgil.

Consider

% 115 Tacitus Hist V 2 1.
Quoniam famosae urbis supremum diem tradituri sumus congruens videtur primordia eius aperire.

It seems obvious that *famosae urbis* should be attached to *supremum diem* and *eius* to *primordia*, rather than, for instance, the other way round. A check on adjacency should be sufficient here, with the order 'cplt followed by np head' (*famosae urbis supremum diem)* in the first case, and the reverse order in the second (*primordia eius).* But can we be sure that no element can come and get stuck between the two nps ? Of course, we can't, as any reader of Latin poetry is well aware. There has to be a check on the number and nature of the intervening elements. If we have available the pathlists of the two nps and the pathlists of their heads, we can specify conditions such as those specified in the np25 rule :

```
[finite,np25] --->
[mapped(np,Fnp1), % the head
 mapped(np,Fnp2), % the noun cplt

 Fnp1 \= Fnp2,  % not the same NP

 constraint([pathlist:PL1,hp:[p(H1,H2)],distance:[Distnp1],sem:Sem,w:W1,
        number:Nb,person:3,gender:G,case:C,class:common,lex:LexNoun], Fnp1),

 constraint([pathlist:PL2, hp:[p(G1,G2)],distance:[Distnp2],w:W2,case:gen],Fnp2),
                                                        % case must be genitive

distancev(PL1,PL2,Strain),                  Strain < 2,
                                              % distance between paths    one word max !!!
distancev([p(H1,H2)],[p(G1,G2)],Strain2),   Strain2 < 3,
                                            % distance between heads    two words max !!!

....
```

Distance and *weight (w)*

The *distance* feature houses the distance (in terms of number of intervening words) between two constituents. If the constituents are expected to be adjacent, the distance factor is a strain which renders computability of the intended construct lower, i.e. the constituent will be harder to parse (for the human reader, not ALP!). Such strain decreases the probability of the parse in the competition with other candidate parses. It is the negative counterpart of *weight.* The value of the weight feature increases with the postulated links posited by the parser.

Constituent_structure

The value of this feature is a bit of the parse tree, or the entire parse tree once we reach the top level, i.e. have a complete parse for a whole sentence. The different bits are linked to one another by inclusion or contiguity. One should remember thath the value assigned to a feature can display any level of complexity. This is why the raw parses and their pretty-printed counterparts can be the values assigned to the *constituent_structure* feature at sentence level.

Cat

The *cat*(egory) feature is easy to understand and of course very handy to retrieve from the mapped structures the ones we are interested in (an np, a pp, a subordinate clause, etc)

Class and Type

These features allow for a subclassification of the main constituents according to the needs of the parsing procedure. The assignments are likely to be modified or added to as the parser develops.

Grammatical Features

Adjective Phrases

Example :

map(adjp,[cat:adjp,pathlist:Sorted,distance:Distance,hp:[p(A,B)],
case:Case,number:N, gender:G,degree:comp,lex:Lex,type:Type,w:Weight,
constituent_structure:[Lex,comp_cplt:FSnp]])].

The feature list shouldn't come as a surprise at this stage of our presentation. The feature *degree* is restricted to adjectives and adverbs

example : *grande* in *dedimus grande patientiae documentum*

adjp:[cat:adjp,pathlist:[p(1,2)],
distance:[0],hp:[p(1,2)],case:acc,number:sing,gender:neuter,lex:grandis,
type:std,w:1,constituent_structure:grandis]]

Remember that the word count in the word list starts at zero, so that the rank of the word is given by the second element of the p structure : *grande* is the second word so that we have *p(1,2)* as constituent of its pathlist.

Noun Phrases

Example

map(np,[pathlist:Sorted,hp:[p(A,B)],index:i(p(A,B)),distance:[Distance],
cat:np,class:Class,sem:Sem,
number:Nb,person:3,gender:Gender,type:core,lex:LexNoun,lextype:full,
case:CaseOut,w:Weight,
constituent_structure:[head:FSnounp,adjp:FSadj]])].

We ought to say a word about the *index* feature, which we observe to coincide with the path assigned to the head of the noun phrase. We have *hp:[p(A,B)]* and *index:i(p(A,B)).*The index points to the position of the head in the np and will be used to associate a gap with its filler. In *rex quem regina amat*, the gap in the clause *regina amat*, i.e. the direct object of *amat* is associated with *rex* by way of relative *quem*.

The gap disappears when the entire sentence is parsed. In *rex quem regina amat amat ancillam*, the gap gets filled by its association with *quem* and the link between *quem* and *rex* is licensed by syntactic rules and semantic restrictions.

Example of np : n and adj
(*populi Romani* in *Non debes temere consulem populi Romani saltatorem vocare*)

noun_cplt:[pathlist:[p(4,5),p(5,6)],hp:[p(4,5)],
index:i(p(4,5)),distance:[0],cat:np,
class:common,
sem:[hum],number:sing,person:3,
gender:masc,type:core,lex:populus,
lextype:full,case:gen,w:2,
constituent_structure:[head:[pathlist:[p(4,5)],pos:noun,
txt:populi, lex:populus, case:gen, gender:masc, class:common,
ab:no, number:sing,sem:[hum]],
adjp:[cat:adjp,pathlist:[p(5,6)], distance:[0],
hp [p(5,6)],
case:gen,number:sing,gender:masc,lex:romanus,
type:std,w:1,
constituent_structure:romanus]]]]]

Prepositional Phrases

Example

map(pp,[pathlist:Sorted,hp:[p(Begin,End)],distance:Dist,index:Inp,
case:Case,prep:Lexprep, sem:Sem,lex:LexNoun,
w:Weight,
type:Type,
% the type is the NP type, not the prep's
% it will be available for checks on the NP within the PP
% for instance in adjuncts
% (Type registers whether the NP is interrogative)
cat:pp,constituent_structure:[prep:Lexprep,head:Fnp]])].

Example (pretty-printed) : (*cum voce* in *Memoriam quoque ipsam cum voce perdidissemus si tam in nostra potestate esset oblivisci quam tacere.)*

prep_phrase_adjunct_1
** cum * * uoce **
index:i(p(3,4))
case:abl
prep:cum
sem:[abstract]
lex:uox
cat:pp
constituent_structure
prep:cum
head
** uoce **
index:i(p(3,4))
cat:np
sem:[abstract]
number:sing
person:3
gender:fem
lex:uox
case:abl
constituent_structure
uoce

Adjuncts

Example

map(adjunct,[pathlist:[p(X,Y)],hp:[p(X,Y)],distance:[0],
cat:np,class:adjunct,value:place,
number:Nb,person:3,gender:Gender,lex:Lexnoun,lextype:full,type:place,
case:gen,w:2,
constituent_structure:[head:[lex:Lexnoun,
sem:location,cat:np,number:Nb,gender:Gender,case:gen,index:i(p(X,Y))]]])].

Example of an np adjunct in the genitive (locative) and with *type*:*place*

Sum Lugdini.

adjunct:[pathlist:[p(1,2)],hp:[p(1,2)],distance:[0],cat:np,
class:adjunct,value:place,number:sing,person:3,gender:neuter,lex:
lugdunum,lextype:full,type:place,case:gen,w:2,constituent_structure:
[head:[lex:lugdunum,sem:location,cat:np,number:sing,gender:neuter,case:
gen,index:i(p(1,2))]]]

Verb Groups

Examples

map(vgpos,[cat:vg,type:finite,pathlist:Sorted,hp:[p(C,D)],lex:Lex,
person:P,mood:Mood,tense:Tenseout, voice:act,
number:Nb,gender:G,w:5])].

map(vgpos,[cat:vg,type:nonfinite,pathlist:Sorted,hp:[p(C,D)],lex:Lex,
person:_,mood:infinitive,tense:perfect,
voice:act,number:Nb,gender:G,w:3])].

Example : *venisses* in *si venisses laetus fuissem*

vg:[selected_reading:uenio_come,polarity:pos,cat:vg,pathlist:[p(1,2)],hp:
[p(1,2)],gender:_85492,w:0,pos:v,class:intr,type:finite,lex:uenire,kind:
std,mood:subjunctive,voice:act,txt:uenisses,tense:pluperfect,number:sing,
person:2]

Clauses

ALP parses finite and non-finite clauses (infinitives, gerundial, supine-based, participial), main and subordinate clauses, with positive or negative polarity, declarative and interrogative. It also parses relative clauses and links them with the antecedent (when there is one).

The *argbound* feature (with values *yes* or *no*) records whether a clause fills in the arg of a predicate (as in *timeo ne* + Clause, *non dubito quin* + Clause, *puto* + accusative-cum-infinitive,...) or is free (*cum se diceret indicaturum de coniuratione*, *ubi paulatim licentia creuit*,...)

Examples

```
map(pred,[cat:pred,type:nonfinite_i,
        mood:supine, tense:present,class:m,
        pathlist:Sorted, distance:[Distance],
        number:sing,gender:neuter,case:or([nom,acc]),
        person:3,
        gap:[],w:Wtotal,
        constituent_structure:[vg:FSverbfull|ST]])].


 map(pred,[cat:pred,
        type:finite,
        pathlist:NSortedF,
        distance:[Distance],
        illocutionary_force:Force,
        class:m,
        number:sing,                % nber, gender, and person of the CLAUSE, not its subject
        person:3,vgperson:Psubj, % needed in imperative clauses of the first and second persons
        gender:neuter,
        mood:Mood,
        tense:Tense,
        polarity:Pol,
        argbound:no,
        gap:Gap,
        w:Wtot,
        add:Add,
        checkint:Interrogative,
        flagint:Flagint,
        constituent_structure:[illocutionary_force:Force,vg:FSverbfull|ST]])].
                                % ST is the sorted list of arg fillers
```

```
map(relative_clause,[pathlist:PL,
                     distance:Distance,
                     gap:GAPARG,            % gap info carried by the clause
                     index:Index,
                     number:Nb,
                     gender:Gender,
                     case:Case,
                     person:PersonRel,
                     reltype:Rel_type,
                     type:finite,
                     mood:Mood,
                     tense:Tense,
                     constraints:Constraints,
                     w:Weight,
                     constituent_structure:C_str]) ].
```

We have already discussed the gap feature. The *polarity* feature is recorded to satisfy constraints on the arg (*haud magni facere*, etc.). *Mood* is essential to the description of both subordinate and main clauses (think of conditionals).

Examples are provided by the parses produced by ALP on its test files.

Using the recorded feature values

The *finiteness* constraint on the *pred* enables us to show only complete parses, i.e. parses for the sentence submitted that make of it an independent gapless (finite) clause.

The 'type:finite' constraint should be lifted in parsing bits of indirect discourse, where a lot of verbs are likely to be infinitives.

```
constraint([cat:pred,
            argbound:no,
            class:m,                    % the whole pred is main and not arg-bound
            pathlist:Pathlist,
            type:finite,                % the finiteness constraint
            distance:[Distance],        % Distance and Weight used in ranking
            gap[],                      % no gap left
            w:Weight,
            constituent_structure:Parse],  % the value of the constituent_structure feature is the
                                           % parse shown to the user, in raw and pretty-printed format
           Pred),  % all the above constraints apply to a candidate clause
```

Test File

% 1 Terentianus Maurus De Syll 1286.

Habent sua fata libelli.

% 2 Cicero Ad Fam VIII 16 4.

Me secum in Hispaniam ducit.

% 3 Tacitus Agricola 2 1 1.

Dedimus profecto grande patientiae documentum.

% 4 Cicero Pro Murena VI 13.

Saltatorem appellat Murenam Cato.

% 5 adapted from Cicero Pro Murena VI 13.

Non debes temere consulem populi Romani saltatorem vocare.

% 6 Terentius Heauton Timorumenos I 77.

Humani nil a me alienum puto.

% 7 Tacitus Hist Lib I LXXVI 9.

Carthaginem ceterae civitates secutae.

% 8 Tacitus Hist Lib I 19 2.

Censuerant patres mittendos ad Germanicum exercitum legatos.

% 9 Ovidius Ars Amatoria 1 42.

Elige cui dicas : tu mihi sola places.

% 10 Horatius Carmina 1 5 1 - notice that the whole thing does not sport a single toolword.

Me tabula sacer votiva paries indicat uvida suspendisse potenti vestimenta maris deo.

% 11 Martial, 2 78.

Aestivo serves ubi piscem tempore quaeris?

% 12 Caesar De Bello Gallico I 9 3 4.

Cupiditate regni adductus nouis rebus studebat.

% 13 Tacitus Hist Lib I XXX 3.

Falluntur quibus luxuria specie liberalitatis imponit.

% 14 'dubitare quin' in Non-affirmative Contexts (from Cicero, Paradoxa, 6 48 1).

Quis igitur dubitet quin in virtute divitiae sint?

% 15 Cicero Att 4 12 1 5 fragmented main clause.

Id tu quoniam Macronem tanti facis ignoscas mihi uelim.

% 16 adapted from Tacitus Hist I 12 2.

Maturavit ea res consilium Galbae iampridem de adoptione cum proximis agitantis.

% 17 adapted from Tacitus Hist Lib I 74 3.

Otho revocatis quos Galba miserat legatis ad utrumque Germanicum exercitum copias misit.

% 17 bis.

Revocatis quos Galba miserat legatos.

% 18 adapted from Cicero Epistulae ad Familiares 9 14 1 1.

Non dubito quin tu meis praeceptis et consiliis obtemperans praestantissimum te civem et singularem consulem praebeas.

% 19 adapted from Cicero Pro Murena XXIII.

Nostrae artes admirabilem utilitatem possident quae nobis studia populi Romani conciliat.

% 20 adapted from Titus Livius Ab Urbe Condita Liber XXI 8.

Quies inter labores renovavit corpora animosque ad omnia de integro patienda.

% 21 Ovidius Ars Amatoria 1 307.

Crede tamen speculo quod te negat esse iuvencam.

% 22 adapted from Cicero Phil I I 1.

Graecum etiam verbum usurpavi quo in sedandis discordiis utitur civitas illa.

% 23 Tacitus Hist Lib I 39 2.

Agitasse Laco ignaro Galba de occidendo Vinio dicitur.

% 24 Tacitus Agricola 2 4 1.

Memoriam quoque ipsam cum voce perdidissemus si tam in nostra potestate esset oblivisci quam tacere.

% 25 Cicero De Amicitia 10 1.

Suis autem incommodis graviter angi non amicum sed se ipsum amantis est.

% 26 Cicero Pro Murena XV.

Omnibus regibus quibuscum populus Romanus bellum gessit hunc regem nimirum antepones.

% 27 Cicero Pro Murena XVII 35.

Totam opinionem parva nonnumquam commutat aura rumoris.

% 28 Catullus XIII 13-14.

Quod tu cum olfacies, deos rogabis totum ut te faciant nasum.

% 29 Cicero Pro Murena XXXIV 85.

Versabitur in rostris furor, in curia timor, in foro coniuratio, in Campo exercitus, in agris vastitas.

% 30 Cicero Pro Murena III 7.

Huic ego satis facere cupio uosque adhibere arbitros.

% 31 Cicero Pro Murena XVII 36.

Nihil est incertius uolgo, nihil obscurius uoluntate hominum.

% 32 Cicero Pro Murena XVIII 37.

Munus amplissimum quod petitio praeturae desiderarat praetura restituit.

% 33 adapted from Cicero Pro Murena XXXVIII 16.

Cogita nonnullorum amicorum studia minui in eos a quibus provincias contemni intellegunt.

% 34 Cic Ad Fam 1 1 2 10.

Is ceteris in rebus se acerrimum tui defensorem fore ostendit.

% 35 Tacitus Ann 12 8 1.

Die nuptiarum Silanus mortem sibi conscivit delecto die augendam ad invidiam.

% 36 adapted from Cicero Cat I 10.

Exclusi eos quos tu salutatum miseras.

% 37 adapted from Cicero Cat I 10.

Illi venerunt quos ad me venturos esse praedixeram.

% 38 adapted from Tacitus Hist Lib I 16 3.

Mihi ac tibi providendum est ne etiam a bonis Nero desideretur.

% 39 adapted from Tacitus Hist Lib I 44 1.

Nullam caedem Otho maiore laetitia excepisse dicitur.

% 40 adapted from Tacitus Hist Lib I 79 1.

Conversis ad civile bellum animis externa sine cura habebantur.

% 41 Tacitus Ann 16 34 1 a.

Arriam tentantem exemplum matris sequi monet retinere vitam.

% 41 Tacitus Ann 16 34 1 b.

Arriam monet filiae communi subsidium unicum non adimere.

% 42 Tacitus Hist Lib I 85 9.

Vitellianos milites venisse in urbem ad studia partium noscenda plerique credebant.

% 43 Tacitus Hist Lib I 46 1.

Omnia deinde arbitrio militum acta.

% 44 Tacitus Hist Lib I 46 1.

Praetorii praefectos sibi ipsi legere.

% 45 Martialis Epigrammata 3 9 2.

Non scribit cuius carmina nemo legit.

% 46 Ovidius Ars Amatoria 2 658.

Nominibus mollire licet mala.

% 47 Tacitus Ann 14 53 4.

Una defensio occurrit quod muneribus tuis obniti non debui.

% 48 Tacitus Ann 14 10 1.

A Caesare perfecto demum scelere magnitudo eius intellecta est.

% 49 Horatius Ep 1 xiii 16.

Ne volgo narres te sudavisse ferendo carmina.

% 50 Cicero Ad fam xiv 4 5.

Non vitium nostrum sed virtus nostra nos afflixit.

% 51 Horatius Sat I 9 59-60.

Nil sine magno vita labore dedit mortalibus.

% 52 Horatius Sat II 6 79-81.

Olim rusticus urbanum murem mus paupere fertur accepisse cavo.

% 53 Tacitus Ann 11 22 2.

Dolabella censuit spectaculum gladiatorum celebrandum pecunia eorum qui quaesturam adipiscerentur.

% 54 adapted from Horatius Ep 2 ii 65-66.

Romaene me poemata censes scribere posse inter tot curas?

% 55 Hieronymus Ep LX 19.

Cotidie morimur et aeternos nos esse credimus.

% 56 Hieronymus Ep LX 19.

Cum quo loqui non possumus de eo loqui numquam desinamus.

% 57 Iuuenalis Sat 8 244.

Roma patrem patriae Ciceronem libera dixit.

% 58 Vergilius Georgica 1 463.

Solem quis dicere falsum audeat?

% 59 Ouidius Amores I 6 34.

Solus eram si non saevus adesset Amor.

% 60 Horatius Carm II 3 1-2.

Aequam memento rebus in arduis servare mentem.

% 61 Ouidius Ars Am 1 132.

Haec mihi si dederis commoda miles ero.

% 62 Peruigilium Veneris 1.

Cras amet qui numquam amavit.

% 63 Sallustius Bellum Iugurthinum V 1.

Superbiae nobilitatis obviam itum est.

% 64 Sallustius Bellum Iugurthinum XX 5.

Legatos ad Iugurtham de iniuriis questum misit.

% 65 Sallustius Bellum Iugurthinum XIV 8.

Ego eis finibus eiectus sum quos maioribus meis populus Romanus dedit.

% 66 adapted from Sallustius Bellum Iugurthinum V 1.

Bellum scripsi quod populus Romanus cum rege Numidarum gessit.

% 67 Sallustius Bellum Iugurthinum LXXXV 31.

Ipsa se uirtus satis ostendit.

% 68 Sallustius Bellum Iugurthinum LXXXV 31.

Illis artificio opus est ut turpia facta oratione tegant.

% 69 Sallustius Bellum Iugurthinum LXXXV 14.

Contemnunt novitatem meam, ego illorum ignaviam.

% 70 Sallustius Bellum Iugurthinum LXXXV 14.

Mihi fortuna, illis probra obiectantur.

% 71 Cicero Pro Murena 84 1.

Mihi credite, iudices.

% 72 Vergilius Eclogae 1 46.

Fortunate senex, ergo tua rura manebunt.

% 73 Vergilius Eclogae 1 6.

O Meliboee, deus nobis haec otia fecit.

% 74 Vergilius Eclogae 1 13.

Hanc etiam vix, Tityre, duco.

% 75 Vergilius Eclogae 4 60.

Incipe, parve puer, risu cognoscere matrem.

% 76 Vergilius Eclogae 9 43.

Insani feriant sine litora fluctus.

% 77 Cicero De Legibus II 1 3.

Ille sapientissimus vir Ithacam ut videret immortalitatem scribitur repudiasse.

% 78 Seneca Dialogi 6 26 3 4.

Cur in domo nostra diutissime lugetur qui felicissime moritur?

% 79 adapted from Cicero De Legibus II 4 6.

Vestigia adsunt eorum quos diligimus.

% 80 Horatius Ep II 2 102.

Multa fero ut placem genus irritabile vatum.

% 81 Tacitus Ann XIII 5.

Ita specie pietatis obviam itum dedecori.

% 82 Tacitus Ann XIII 15.

Nero intellecta invidia odium intendit pararique venenum iubet.

% 83 Tacitus Ann XV 52 1.

Coniuratis tamen metu permotis placitum maturare caedem.

% 84 Sallustius Catilinae Coniuratio 1 1 2.

Omnes homines niti decet ne uitam silentio transeant.

% 85 Tacitus Hist Lib I 62 1.

Mira inter exercitum imperatoremque diversitas.

% 86 Tacitus Hist Lib I 10 1.

Oriens adhuc immotus.

% 87 Tacitus Hist Lib I 9 7.

In Britannico exercitu nihil irarum.

% 88 Tacitus Hist Lib I 7 13.

Venalia cuncta.

% 89 Tacitus Hist Lib I 7 15.

Eadem novae aulae mala.

% 90 Tacitus Hist Lib I 2 13.

Plenum exiliis mare.

% 91 PS 27 1.

Dominus illuminatio mea.

% 92 Vergilius Georgica 2 458.

Fortunatos nimium sua si bona norint agricolas.

% 93 Catullus Carmina 61 184.

Iam licet venias, marite.

% 94 Catullus Carmina 61 94-95.

Vide ut faces aureas quatiunt comas.

% 95 Catullus Carmina 72 1.

Dicebas quondam solum te nosse Catullum.

% 96 Catullus Carmina 46 6.

Ad claras Asiae volemus urbes.

% 97 adapted from Cicero In Cat 1 17 2.

Servi mei si me metuerent ut te metuunt cives tui domum meam relinquendam putarem.

% 98 Cicero Att VII 3 4.

De sua potentia dimicant homines hoc tempore periculo civitatis.

% 99 Ovidius Tristia 1 9 5.

Donec eris sospes multos numerabis amicos.

% 100 Ovidius Tristia 1 9 6.

Tempora si fuerint nubila solus eris.

% 101 Seneca Epistulae Morales ad Lucilium 7 1 4.

Mane leonibus et ursis homines, meridie spectatoribus suis obiiciuntur.

% 102 Cicero Pro Rege Deiotaro 30 14-15.

Nulli parietes nostram salutem, nullae leges, nulla iura custodient.

% 103 Tacitus Ann 16 34 1.

Tum ad Thraseam in hortis agentem quaestor consulis missus vesperascente iam die.

% 104 Tacitus Hist I 75 1.

Insidiatores ab Othone in Germaniam, a Vitellio in urbem missi.

% 105 Tacitus Hist I 11 1.

Aegyptum copiasque quibus coerceretur equites Romani obtinent loco regum.

% 106 Martialis Epigr Lib I 3 1-2.

Argiletanas mauis habitare tabernas cum tibi, parue liber, scrinia nostra vacent.

% 107 Martialis Epigr Lib I 117 8.

Quod quaeris propius petas licebit.

% 108 Propertius El III 25 17.

Has tibi fatalis cecinit mea pagina diras.

% 109 Propertius El III 25 18.

Euentum formae disce timere tuae.

% 110 adapted from Propertius El III 20 1-2.

Credis iam tuae meminisse figurae quem uidisti a lecto dare uela tuo.

% 111 Tacitus Hist IV 1 1.

Interfecto Vitellio bellum magis desierat quam pax coeperat.

% 112 Tacitus Ann 13 21 20.

Vivere ego Britannico potiente rerum poteram?

% 113 Tacitus Hist I 75 1.

Ita promissis simul ac minis tentabantur.

% 114 Tacitus Hist IV 16 1.

Civilis dolo grassandum ratus incusavit ultro praefectos quod castella deseruissent.

% 115 Tacitus Hist V 2 1.

Quoniam famosae urbis supremum diem traduturi sumus congruens videtur primordia eius aperire.

% 116 Tacitus Hist II 2 1.

Fuerunt qui accensum desiderio reginae vertisse iter crederent.

% 117 Tacitus Hist IV 58 2.

Bellum cum populo Romano vestris se manibus gesturum Classicus sperat.

% 118 Tacitus Dialogus 27 3 2.

Cum de antiquis loquaris utere antiqua libertate a qua vel magis degenerauimus quam ab eloquentia.

% 119 Lucius Annaeus Seneca senior Controversiae 8 1 10.

I ad illum quem magis amas quam patrem.

% 120 (20-27 build up Tacitus Hist IV 64 3-4).

ut amicitia societasque nostra in aeternum rata sint /

% 121.

postulamus a uobis muros coloniae munimenta servitii detrahatis.

% 122.

etiam fera animalia si clausa teneas virtutis obliviscuntur.

% 123.

Romanos omnis in finibus uestris trucidetis.

% 124.

haud facile libertas et domini miscentur.

% 125.

bona interfectorum in medium cedant.

% 126.

ne quis occulere quicquam aut segregare causam suam possit /

% 127.

liceat nobis vobisque utramque ripam colere.

% 128 (adapted from Tac ANN III 31 4).

Memorabantur exempla maiorum qui iuventutis irreverentiam gravibus decretis notavissent.

% 129 Tac ANN I 42 4.

Hunc ego nuntium patri laeta omnia aliis e provinciis audienti feram.

% 130 and 131 Caesar BG 1 XL 1.

Ariovistum se consule cupidissime populi Romani amicitiam appetisse.

% 131.

cur hunc tam temere quisquam ab officio discessurum iudicaret?

% 132 and 133 (Cicero Pro Milone 38 104).

hunc sua quisquam sententia ex hac urbe expellet /

% 133.

quem omnes urbes expulsum a vobis ad se vocabunt.

stop.

Test sentences which used to be used in teaching Latin in French schools

From USUS

Accepi litteras a patre.
Age quod agis.
Marcus, cum Ciceronem interfecisset, magnitudinem facinoris perspexit.
Ambulat in horto.
Amo patrem.
Amor a patre.
Angebat Hamilcarem amissa Sicilia.
Angebant ingentis spiritus virum Sicilia Sardiniaque amissae.
Beneficiorum memini.
Credit se esse beatum.
Cum amico cenabam.
Amo magistros cupidos legendi.
Amo magistros cupidos legendi historiam.
Amo magistros cupidos legendae historiae.
Cicerone consule omnes magistri insanivere.
Dicunt Homerum caecum fuisse.
Doceo pueros grammaticam.
Est doctior Petro.
Est doctior quam Petrus.
Eo lusum.
Eo Lutetiam.
Errare humanum est.
Est hominis rationem sequi.
Haec est invidia.
Homerus dicitur caecus fuisse.
Ibam forte Via Sacra.
Iter feci per Galliam.
Legat librum Petri.
Litterae quas scripsisti mihi iucundissimae fuerunt.
Magna voce clamat.
Me paenitet erroris mei.
Mihi colenda est uirtus.
Mihi est libellus impudicus.
Misit legatos qui pacem peterent.
Ne hoc faciamus.
Ne hoc feceris.
Ne mortem timueritis.
Noli hoc facere.
Nonne amicus meus es ?
Num insanis ?
Orat te pater ut ad se venias.
Orat te mater ut filio ignoscas suo.
Partibus factis verba facit leo.
Pater est bonus.
Pater et mater sunt boni.
Est Marcus peritus belli.
Pugnandum est.

Pugnatur.
Quaero num pater tuus venerit.
Quaero ueneritne pater tuus.
Quaero quis uenerit.
Scio uitam esse breuem.
Scripturus sum.
Si hunc librum leges, laetus ero.
Si hunc librum legeris, laetus ero.
Si venias, laetus sum.
Si venires, laetus essem.
Si venisses, laetus fuissem.
Sum Lugduni.
Timeo ne non veniat.
Timeo ne veniat.
Tres annos regnavit.
Urbem captam hostis diripuit.
Urbem Romam reges habuere.
Utinam illum diem videam !
Utinam dives essem !
Utinam omnes Marcus servare potuisset !
Utor memoria.
Venit in hortum.
Victi sunt consules apud Cannas.
Vidistine Romam ?
stop.

Alp189 timing :
TOTAL TIME : 17 sec

A Few Example Parses

[0/carthaginem,1/ceterae,2/ciuitates,3/secutae,endpos(4)]

```
illocutionary_force:statement
vg
  selected_reading:sequor_follow
  polarity:pos
  cat:vg
  * secutae *
  lex:sequi_2
  person:3
  mood:indicative
  tense:perfect
  voice:act
  number:pl
  gender:fem
  subject
    number:pl
    gender:fem
    * ceterae * * ciuitates *
    index:i(p(2,3))
    cat:np
    sem:[hum,loc]
    person:3
    lex:ciuitas
    case:nom
    c_str
      head
        * ciuitates *
        pos:noun
        lex:ciuitas
        case:nom
        gender:fem
        number:pl
        sem:[hum,loc]
      adjp
        cat:adjp
        * ceterae *
        case:nom
        number:pl
        gender:fem
        lex:ceterus
  object
    * carthaginem *
    index:i(p(0,1))
    cat:np
    sem:[city,thing,abstract]
    number:sing
    person:3
    gender:fem
    lex:carthago
    case:acc
    c_str
    carthaginem
```

[0/id,1/tu,2/quoniam,3/macronem,4/tanti,5/facis,6/ignoscas,7/mihi,8/uelim,endpos(9)]

```
 subordinator
   lex:quoniam
   pos:sub
   argbound:no
   mood:indicative
   value:reason
subordinate_clause
   illocutionary_force:statement
   vg
      selected_reading:mwu_tanti_facio_appreciate
      polarity:pos
      cat:vg
      * facis *
      pos:v
      lex:facere
      voice:act
      tense:present
      mood:indicative
      number:sing
      person:2
      subject
         source:context_retrievable
         number:sing
         gender:or([masc,fem])
         person:2
         cat:np
         index:i(0,0)
         constraints_to_be_met:[sem:[hum]]
         case:nom
      object
         * macronem *
         index:i(p(3,4))
         cat:np
         sem:[hum]
         lex:macro
         number:sing
         person:3
         gender:masc
         case:acc
main_clause
   illocutionary_force:statement
   vg
      selected_reading:uolo_want_sbd_to
      polarity:pos
      cat:vg
      * uelim *
      pos:v
      lex:uelle
      voice:act
      tense:present
      mood:subjunctive
      number:sing
      person:1
      subject
         source:context_retrievable
         number:sing
         gender:or([masc,fem])
         person:1
         cat:np
         index:i(0,0)
         constraints_to_be_met:[sem:[hum]]
         case:nom
      object
         cat:pred
         * id * * tu * * ignoscas * * mihi *
         illocutionary_force:statement
         number:sing
         person:3
         vgperson:2
         gender:neuter
         mood:subjunctive
         tense:present
         polarity:pos
         argbound:no
```

```
add:no
c_str
   illocutionary_force:statement
   vg
      selected_reading:ignosco_pardon
      polarity:pos
      cat:vg
      * ignoscas *
      pos:v
      lex:ignoscere
      voice:act
      tense:present
      mood:subjunctive
      number:sing
      person:2
      subject
         number:sing
         gender:or([masc,fem])
         * tu *
         index:i(p(1,2))
         cat:np
         sem:[hum]
         lex:pp2sg
         person:2
         case:nom
      object
         * id *
         index:i(p(0,1))
         cat:np
         lex:prpersnomaccneutersing
         number:sing
         person:3
         gender:neuter
         case:or([nom,acc])
      i_object
         * mihi *
         index:i(p(7,8))
         cat:np
         sem:[hum]
         lex:pp1sg
         number:sing
         person:1
         gender:or([masc,fem])
         case:dat
```

[0/otho,1/reuocatis,2/quos,3/galba,4/miserat,5/legatis,
6/ad,7/utrumque,8/germanicum,9/exercitum,10/copias,11/misit,endpos(12)]

```
illocutionary_force:statement
vg
  selected_reading:mitto_send
  polarity:pos
  cat:vg
  * misit *
  pos:v
  lex:mittere
  voice:act
  tense:perfect
  mood:indicative
  number:sing
  person:3
  subject
    number:sing
    gender:masc
    * otho *
    index:i(p(0,1))
    cat:np
    sem:[hum]
    lex:otho
    person:3
    case:nom
  object
    * copias *
    index:i(p(10,11))
    cat:np
    sem:[hum]
    number:pl
    person:3
    gender:fem
    lex:copiae
    case:acc
    c_str
    copias
  prep_cplt
    * ad * * utrumque * * germanicum * * exercitum *
    index:i(p(9,10))
    case:acc
    prep:ad
    sem:[thing,hum,loc]
    lex:exercitus
    cat:pp
    c_str
      prep:ad
      head
        * utrumque * * germanicum * * exercitum *
        index:i(p(9,10))
        cat:np
        sem:[thing,hum,loc]
        number:sing
        person:3
        gender:masc
        lex:exercitus
        case:acc
        c_str
          head
            * exercitum *
            pos:noun
            lex:exercitus
            case:acc
            gender:masc
            number:sing
            sem:[thing,hum,loc]
          adjp
            cat:adjp
            * utrumque * * germanicum *
            case:acc
            number:sing
            gender:masc
            lex:germanicus
            c_str
              uterque
```

germanicus
ablative_absolute
* reuocatis * * quos * * galba * * miserat * * legatis *
c_str
lex:reuoco_call_back
pos:p_p
case:abl
gender:masc
number:pl
lex:reuocare
mood:participle
person:3
object
* quos * * galba * * miserat * * legatis *
cat:np
index:i(p(5,6))
number:pl
gender:masc
sem:[hum]
person:3
case:abl
lex:legatus
c_str
head
legatis
rel_clause
* quos * * galba * * miserat *
index:i(p(5,6))
number:pl
gender:masc
case:acc
reltype:restrictive
mood:indicative
tense:pluperfect
c_str
illocutionary_force:statement
vg
selected_reading:mitto_send
polarity:pos
cat:vg
* miserat *
pos:v
lex:mittere
voice:act
tense:pluperfect
mood:indicative
number:sing
person:3
subject
number:sing
gender:masc
* galba *
index:i(p(3,4))
cat:np
sem:[hum]
lex:galba
person:3
case:nom
object
e:i(p(5,6))

[0/reuocatis,1/quos,2/galba,3/miserat,4/legatos,endpos(5)]

illocutionary_force:statement
vg
selected_reading:reuoco_call_back
polarity:pos
cat:vg
* reuocatis *
pos:v
lex:reuocare
voice:act
tense:present
mood:indicative
number:pl
person:2
subject
source:context_retrievable
number:pl
gender:or([masc,fem])
person:2
cat:np
index:i(0,0)
constraints_to_be_met:[sem:[hum]]
case:nom
object
* quos * * galba * * miserat * * legatos *
cat:np
index:i(p(4,5))
number:pl
gender:masc
sem:[hum]
person:3
case:acc
lex:legatus
c_str
head
legatos
rel_clause
* quos * * galba * * miserat *
index:i(p(4,5))
number:pl
gender:masc
case:acc
reltype:restrictive
mood:indicative
tense:pluperfect
c_str
illocutionary_force:statement
vg
selected_reading:mitto_send
polarity:pos
cat:vg
* miserat *
pos:v
lex:mittere
voice:act
tense:pluperfect
mood:indicative
number:sing
person:3
subject
number:sing
gender:masc
* galba *
index:i(p(2,3))
cat:np
sem:[hum]
lex:galba
person:3
case:nom
object
e:i(p(4,5))

Morphological Variants: Rogo

verb([v(rogare,1,rog,rogau,rogat)],tr_cod,std).

% the v functor encompasses infinitive, conjugation and the three roots. We then have the verb class, and
% the indication that the verb behaves 'standardly' with respect to the production of morphological variants

lex(roga, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:roga, tense:present, kind:_, mood:imperative, number:sing, person:2]).
lex(rogabam, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabam, tense:imperfect, kind:_, mood:indicative, number:sing, person:1]).
lex(rogabamini, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogabamini, tense:imperfect, kind:_, mood:indicative, number:pl, person:2]).
lex(rogabamur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogabamur, tense:imperfect, kind:_, mood:indicative, number:pl, person:1]).
lex(rogabamus, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabamus, tense:imperfect, kind:_, mood:indicative, number:pl, person:1]).
lex(rogabant, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabant, tense:imperfect, kind:_, mood:indicative, number:pl, person:3]).
lex(rogabantur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogabantur, tense:imperfect, kind:_, mood:indicative, number:pl, person:3]).
lex(rogabar, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogabar, tense:imperfect, kind:_, mood:indicative, number:sing, person:1]).
lex(rogabaris, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogabaris, tense:imperfect, kind:_, mood:indicative, number:sing, person:2]).
lex(rogabas, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabas, tense:imperfect, kind:_, mood:indicative, number:sing, person:2]).
lex(rogabat, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabat, tense:imperfect, kind:_, mood:indicative, number:sing, person:3]).
lex(rogabatis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabatis, tense:imperfect, kind:_, mood:indicative, number:pl, person:2]).
lex(rogabatur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:middle, txt:rogabatur, tense:imperfect, kind:_, mood:indicative, number:sing, person:3]).
lex(rogabatur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogabatur, tense:imperfect, kind:_, mood:indicative, number:sing, person:3]).
lex(rogaberis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogaberis, tense:future, kind:_, mood:indicative, number:sing, person:2]).
lex(rogabimini, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogabimini, tense:future, kind:_, mood:indicative, number:pl, person:2]).
lex(rogabimur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogabimur, tense:future, kind:_, mood:indicative, number:pl, person:1]).
lex(rogabimus, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabimus, tense:future, kind:_, mood:indicative, number:pl, person:1]).
lex(rogabis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabis, tense:future, kind:_, mood:indicative, number:sing, person:2]).
lex(rogabit, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabit, tense:future, kind:_, mood:indicative, number:sing, person:3]).
lex(rogabitis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabitis, tense:future, kind:_, mood:indicative, number:pl, person:2]).
lex(rogabitur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:middle, txt:rogabitur, tense:future, kind:_, mood:indicative, number:sing, person:3]).
lex(rogabitur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogabitur, tense:future, kind:_, mood:indicative, number:sing, person:3]).
lex(rogabo, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabo, tense:future, kind:_, mood:indicative, number:sing, person:1]).
lex(rogabor, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogabor, tense:future, kind:_, mood:indicative, number:sing, person:1]).
lex(rogabunt, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogabunt, tense:future, kind:_, mood:indicative, number:pl, person:3]).
lex(rogabuntur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogabuntur, tense:future, kind:_, mood:indicative, number:pl, person:3]).
lex(rogamini, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogamini, tense:present, kind:_, mood:indicative, number:pl, person:2]).
lex(rogamur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogamur, tense:present, kind:_, mood:indicative, number:pl, person:1]).
lex(rogamus, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogamus, tense:present, kind:_, mood:indicative, number:pl, person:1]).
lex(roganda, gdiv, [pos:gdiv, txt:roganda, case:abl, gender:fem, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(roganda, gdiv, [pos:gdiv, txt:roganda, case:acc, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(roganda, gdiv, [pos:gdiv, txt:roganda, case:nom, gender:fem, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(roganda, gdiv, [pos:gdiv, txt:roganda, case:nom, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandae, gdiv, [pos:gdiv, txt:rogandae, case:dat, gender:fem, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandae, gdiv, [pos:gdiv, txt:rogandae, case:gen, gender:fem, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandae, gdiv, [pos:gdiv, txt:rogandae, case:nom, gender:fem, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandam, gdiv, [pos:gdiv, txt:rogandam, case:acc, gender:fem, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandarum, gdiv, [pos:gdiv, txt:rogandarum, case:gen, gender:fem, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandas, gdiv, [pos:gdiv, txt:rogandas, case:acc, gender:fem, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandi, gdiv, [pos:gdiv, txt:rogandi, case:gen, gender:masc, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandi, gdiv, [pos:gdiv, txt:rogandi, case:gen, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandi, gdiv, [pos:gdiv, txt:rogandi, case:nom, gender:masc, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandi, v, [pos:v, class:tr_cod, type:gerund, lex:rogare, txt:rogandi, kind:_, mood:gerund, person:3, case:gen]).
lex(rogandis, gdiv, [pos:gdiv, txt:rogandis, case:abl, gender:fem, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandis, gdiv, [pos:gdiv, txt:rogandis, case:abl, gender:masc, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandis, gdiv, [pos:gdiv, txt:rogandis, case:abl, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandis, gdiv, [pos:gdiv, txt:rogandis, case:dat, gender:fem, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandis, gdiv, [pos:gdiv, txt:rogandis, case:dat, gender:masc, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandis, gdiv, [pos:gdiv, txt:rogandis, case:dat, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogando, gdiv, [pos:gdiv, txt:rogando, case:abl, gender:masc, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogando, gdiv, [pos:gdiv, txt:rogando, case:abl, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogando, gdiv, [pos:gdiv, txt:rogando, case:dat, gender:masc, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogando, gdiv, [pos:gdiv, txt:rogando, case:dat, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogando, v, [pos:v, class:tr_cod, type:gerund, lex:rogare, txt:rogando, kind:_, mood:gerund, person:3, case:abl]).
lex(rogandorum, gdiv, [pos:gdiv, txt:rogandorum, case:gen, gender:masc, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandorum, gdiv, [pos:gdiv, txt:rogandorum, case:gen, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandos, gdiv, [pos:gdiv, txt:rogandos, case:acc, gender:masc, number:pl, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandum, gdiv, [pos:gdiv, txt:rogandum, case:acc, gender:masc, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandum, gdiv, [pos:gdiv, txt:rogandum, case:acc, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandum, gdiv, [pos:gdiv, txt:rogandum, case:nom, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogandum, v, [pos:v, class:tr_cod, type:gerund, lex:rogare, txt:rogandum, kind:_, mood:gerund, person:3, case:acc]).
lex(rogandus, gdiv, [pos:gdiv, txt:rogandus, case:nom, gender:masc, number:sing, lex:rogare, class:tr_cod, type:gdiv, kind:_, mood:gerund, person:3]).
lex(rogans, p_pr, [pos:p_pr, txt:rogans, case:acc, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3]).
lex(rogans, p_pr, [pos:p_pr, txt:rogans, case:nom, gender:_, number:sing, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3]).
lex(rogant, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogant, tense:present, kind:_, mood:indicative, number:pl, person:3]).
lex(rogante, p_pr, [pos:p_pr, txt:rogante, case:abl, gender:_, number:sing, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3, typeabl:e]).
lex(rogantem, p_pr, [pos:p_pr, txt:rogantem, case:acc, gender:or([masc, fem]), number:sing, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3]).
lex(rogantes, p_pr, [pos:p_pr, txt:rogantes, case:acc, gender:or([masc, fem]), number:pl, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3]).
lex(rogantes, p_pr, [pos:p_pr, txt:rogantes, case:nom, gender:or([masc, fem]), number:pl, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle,

person:3]).
lex(roganti, p_pr, [pos:p_pr, txt:roganti, case:abl, gender:_, number:sing, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3, typeabl:i]).
lex(roganti, p_pr, [pos:p_pr, txt:roganti, case:dat, gender:_, number:sing, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3]).
lex(rogantia, p_pr, [pos:p_pr, txt:rogantia, case:acc, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3]).
lex(rogantia, p_pr, [pos:p_pr, txt:rogantia, case:nom, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3]).
lex(rogantibus, p_pr, [pos:p_pr, txt:rogantibus, case:abl, gender:_, number:pl, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3]).
lex(rogantibus, p_pr, [pos:p_pr, txt:rogantibus, case:dat, gender:_, number:pl, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3]).
lex(rogantis, p_pr, [pos:p_pr, txt:rogantis, case:gen, gender:_, number:sing, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3]).
lex(rogantium, p_pr, [pos:p_pr, txt:rogantium, case:gen, gender:_, number:pl, lex:rogare, class:tr_cod, type:p_pr, kind:_, mood:participle, person:3]).
lex(roganto, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:roganto, tense:future, kind:_, mood:imperative, number:pl, person:3]).
lex(rogantur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogantur, tense:present, kind:_, mood:indicative, number:pl, person:3]).
lex(rogare, v, [pos:v, class:tr_cod, type:nonfinite, lex:rogare, voice:act, txt:rogare, tense:present, kind:_, mood:infinitive, number:_, person:_]).
lex(rogarem, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogarem, tense:imperfect, kind:_, mood:subjunctive, number:sing, person:1]).
lex(rogaremini, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogaremini, tense:imperfect, kind:_, mood:subjunctive, number:pl, person:2]).
lex(rogaremur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogaremur, tense:imperfect, kind:_, mood:subjunctive, number:pl, person:1]).
lex(rogaremus, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogaremus, tense:imperfect, kind:_, mood:subjunctive, number:pl, person:1]).
lex(rogarent, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogarent, tense:imperfect, kind:_, mood:subjunctive, number:pl, person:3]).
lex(rogarentur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogarentur, tense:imperfect, kind:_, mood:subjunctive, number:pl, person:3]).
lex(rogarer, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogarer, tense:imperfect, kind:_, mood:subjunctive, number:sing, person:1]).
lex(rogareris, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogareris, tense:imperfect, kind:_, mood:subjunctive, number:sing, person:2]).
lex(rogares, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogares, tense:imperfect, kind:_, mood:subjunctive, number:sing, person:2]).
lex(rogaret, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogaret, tense:imperfect, kind:_, mood:subjunctive, number:sing, person:3]).
lex(rogaretis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogaretis, tense:imperfect, kind:_, mood:subjunctive, number:pl, person:2]).
lex(rogaretur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogaretur, tense:imperfect, kind:_, mood:subjunctive, number:sing, person:3]).
lex(rogari, v, [pos:v, class:tr_cod, type:nonfinite, lex:rogare, voice:middle, txt:rogari, tense:present, kind:_, mood:infinitive, number:_, person:_]).
lex(rogari, v, [pos:v, class:tr_cod, type:nonfinite, lex:rogare, voice:pass, txt:rogari, tense:present, kind:_, mood:infinitive, number:_, person:_]).
lex(rogaris, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogaris, tense:present, kind:_, mood:indicative, number:sing, person:2]).
lex(rogas, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogas, tense:present, kind:_, mood:indicative, number:sing, person:2]).
lex(rogat, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogat, tense:present, kind:_, mood:indicative, number:sing, person:3]).
lex(rogata, p_p, [pos:p_p, txt:rogata, case:abl, gender:fem, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogata, p_p, [pos:p_p, txt:rogata, case:acc, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogata, p_p, [pos:p_p, txt:rogata, case:nom, gender:fem, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogata, p_p, [pos:p_p, txt:rogata, case:nom, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatae, p_p, [pos:p_p, txt:rogatae, case:dat, gender:fem, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatae, p_p, [pos:p_p, txt:rogatae, case:gen, gender:fem, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatae, p_p, [pos:p_p, txt:rogatae, case:nom, gender:fem, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatam, p_p, [pos:p_p, txt:rogatam, case:acc, gender:fem, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatarum, p_p, [pos:p_p, txt:rogatarum, case:gen, gender:fem, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatas, p_p, [pos:p_p, txt:rogatas, case:acc, gender:fem, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogate, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogate, tense:present, kind:_, mood:imperative, number:pl, person:2]).
lex(rogati, p_p, [pos:p_p, txt:rogati, case:gen, gender:masc, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogati, p_p, [pos:p_p, txt:rogati, case:gen, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogati, p_p, [pos:p_p, txt:rogati, case:nom, gender:masc, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatis, p_p, [pos:p_p, txt:rogatis, case:abl, gender:fem, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatis, p_p, [pos:p_p, txt:rogatis, case:abl, gender:masc, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatis, p_p, [pos:p_p, txt:rogatis, case:abl, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatis, p_p, [pos:p_p, txt:rogatis, case:dat, gender:fem, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatis, p_p, [pos:p_p, txt:rogatis, case:dat, gender:masc, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatis, p_p, [pos:p_p, txt:rogatis, case:dat, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogatis, tense:present, kind:_, mood:indicative, number:pl, person:2]).
lex(rogato, p_p, [pos:p_p, txt:rogato, case:abl, gender:masc, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogato, p_p, [pos:p_p, txt:rogato, case:abl, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogato, p_p, [pos:p_p, txt:rogato, case:dat, gender:masc, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogato, p_p, [pos:p_p, txt:rogato, case:dat, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogato, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogato, tense:future, kind:_, mood:imperative, number:sing, person:2]).
lex(rogato, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogato, tense:future, kind:_, mood:imperative, number:sing, person:3]).
lex(rogatorum, p_p, [pos:p_p, txt:rogatorum, case:gen, gender:masc, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatorum, p_p, [pos:p_p, txt:rogatorum, case:gen, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatos, p_p, [pos:p_p, txt:rogatos, case:acc, gender:masc, number:pl, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatote, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogatote, tense:future, kind:_, mood:imperative, number:pl, person:2]).
lex(rogatum, p_p, [pos:p_p, txt:rogatum, case:acc, gender:masc, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatum, p_p, [pos:p_p, txt:rogatum, case:acc, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatum, p_p, [pos:p_p, txt:rogatum, case:nom, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogatum, v, [pos:v, class:tr_cod, type:supine, lex:rogare, txt:rogatum, kind:_, mood:supine, person:3, case:_]).
lex(rogatur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:middle, txt:rogatur, tense:present, kind:_, mood:indicative, number:sing, person:3]).
lex(rogatur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogatur, tense:present, kind:_, mood:indicative, number:sing, person:3]).
lex(rogatura, p_f, [pos:p_f, txt:rogatura, case:abl, gender:fem, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogatura, p_f, [pos:p_f, txt:rogatura, case:acc, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogatura, p_f, [pos:p_f, txt:rogatura, case:nom, gender:fem, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogatura, p_f, [pos:p_f, txt:rogatura, case:nom, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturae, p_f, [pos:p_f, txt:rogaturae, case:dat, gender:fem, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturae, p_f, [pos:p_f, txt:rogaturae, case:gen, gender:fem, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturae, p_f, [pos:p_f, txt:rogaturae, case:nom, gender:fem, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturam, p_f, [pos:p_f, txt:rogaturam, case:acc, gender:fem, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturarum, p_f, [pos:p_f, txt:rogaturarum, case:gen, gender:fem, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturas, p_f, [pos:p_f, txt:rogaturas, case:acc, gender:fem, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturi, p_f, [pos:p_f, txt:rogaturi, case:gen, gender:masc, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturi, p_f, [pos:p_f, txt:rogaturi, case:gen, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturi, p_f, [pos:p_f, txt:rogaturi, case:nom, gender:masc, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturis, p_f, [pos:p_f, txt:rogaturis, case:abl, gender:fem, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturis, p_f, [pos:p_f, txt:rogaturis, case:abl, gender:masc, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturis, p_f, [pos:p_f, txt:rogaturis, case:abl, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturis, p_f, [pos:p_f, txt:rogaturis, case:dat, gender:fem, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturis, p_f, [pos:p_f, txt:rogaturis, case:dat, gender:masc, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturis, p_f, [pos:p_f, txt:rogaturis, case:dat, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturo, p_f, [pos:p_f, txt:rogaturo, case:abl, gender:masc, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturo, p_f, [pos:p_f, txt:rogaturo, case:abl, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturo, p_f, [pos:p_f, txt:rogaturo, case:dat, gender:masc, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturo, p_f, [pos:p_f, txt:rogaturo, case:dat, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturorum, p_f, [pos:p_f, txt:rogaturorum, case:gen, gender:masc, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).

lex(rogaturorum, p_f, [pos:p_f, txt:rogaturorum, case:gen, gender:neuter, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturos, p_f, [pos:p_f, txt:rogaturos, case:acc, gender:masc, number:pl, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturum, p_f, [pos:p_f, txt:rogaturum, case:acc, gender:masc, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturum, p_f, [pos:p_f, txt:rogaturum, case:acc, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturum, p_f, [pos:p_f, txt:rogaturum, case:nom, gender:neuter, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogaturus, p_f, [pos:p_f, txt:rogaturus, case:nom, gender:masc, number:sing, lex:rogare, class:tr_cod, type:p_f, kind:_, mood:participle, person:3]).
lex(rogatus, p_p, [pos:p_p, txt:rogatus, case:nom, gender:masc, number:sing, lex:rogare, class:tr_cod, type:p_p, kind:_, mood:participle, person:3]).
lex(rogaueram, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogaueram, tense:pluperfect, kind:_, mood:indicative, number:sing, person:1]).
lex(rogaueramus, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogaueramus, tense:pluperfect, kind:_, mood:indicative, number:pl, person:1]).
lex(rogauerant, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauerant, tense:pluperfect, kind:_, mood:indicative, number:pl, person:3]).
lex(rogaueras, v, [pos:v, class:tr_cod, type:finite, lex:rogare, kind:_, mood:indicative, voice:act, txt:rogaueras, tense:pluperfect, number:sing, person:2]).
lex(rogauerat, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauerat, tense:pluperfect, kind:_, mood:indicative, number:sing, person:3]).
lex(rogaueratis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogaueratis, tense:pluperfect, kind:_, mood:indicative, number:pl, person:2]).
lex(rogauere, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauere, tense:perfect, kind:_, mood:indicative, number:pl, person:3]).
lex(rogauerim, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauerim, tense:perfect, kind:_, mood:subjunctive, number:sing, person:1]).
lex(rogauerimus, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauerimus, tense:future_perfect, kind:_, mood:indicative, number:pl, person:1]).
lex(rogauerimus, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauerimus, tense:perfect, kind:_, mood:subjunctive, number:pl, person:1]).
lex(rogauerint, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauerint, tense:future_perfect, kind:_, mood:indicative, number:pl, person:3]).
lex(rogauerint, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauerint, tense:perfect, kind:_, mood:subjunctive, number:pl, person:3]).
lex(rogaueris, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogaueris, tense:future_perfect, kind:_, mood:indicative, number:sing, person:2]).
lex(rogaueris, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogaueris, tense:perfect, kind:_, mood:subjunctive, number:sing, person:2]).
lex(rogauerit, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauerit, tense:future_perfect, kind:_, mood:indicative, number:sing, person:3]).
lex(rogauerit, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauerit, tense:perfect, kind:_, mood:subjunctive, number:sing, person:3]).
lex(rogaueritis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogaueritis, tense:future_perfect, kind:_, mood:indicative, number:pl, person:2]).
lex(rogaueritis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogaueritis, tense:perfect, kind:_, mood:subjunctive, number:pl, person:2]).
lex(rogauero, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauero, tense:future_perfect, kind:_, mood:indicative, number:sing, person:1]).
lex(rogauerunt, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauerunt, tense:perfect, kind:_, mood:indicative, number:pl, person:3]).
lex(rogaui, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogaui, tense:perfect, kind:_, mood:indicative, number:sing, person:1]).
lex(rogauimus, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauimus, tense:perfect, kind:_, mood:indicative, number:pl, person:1]).
lex(rogauisse, v, [pos:v, class:tr_cod, type:nonfinite, lex:rogare, voice:act, txt:rogare, tense:past, kind:_, mood:infinitive, number:_, person:_]).
lex(rogauissem, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauissem, tense:pluperfect, kind:_, mood:subjunctive, number:sing, person:1]).
lex(rogauissemus, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauissemus, tense:pluperfect, kind:_, mood:subjunctive, number:pl, person:1]).
lex(rogauissent, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauissent, tense:pluperfect, kind:_, mood:subjunctive, number:pl, person:3]).
lex(rogauisses, v, [pos:v, class:tr_cod, type:finite, lex:rogare, kind:_, mood:subjunctive, voice:act, txt:rogauisses, tense:pluperfect, number:sing, person:2]).
lex(rogauisset, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauisset, tense:pluperfect, kind:_, mood:subjunctive, number:sing, person:3]).
lex(rogauissetis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauissetis, tense:pluperfect, kind:_, mood:subjunctive, number:pl, person:2]).
lex(rogauisti, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauisti, tense:perfect, kind:_, mood:indicative, number:sing, person:2]).
lex(rogauistis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauistis, tense:perfect, kind:_, mood:indicative, number:pl, person:2]).
lex(rogauit, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogauit, tense:perfect, kind:_, mood:indicative, number:sing, person:3]).
lex(rogem, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogem, tense:present, kind:_, mood:subjunctive, number:sing, person:1]).
lex(rogemini, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogemini, tense:present, kind:_, mood:subjunctive, number:pl, person:2]).
lex(rogemur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogemur, tense:present, kind:_, mood:subjunctive, number:pl, person:1]).
lex(rogemus, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogemus, tense:present, kind:_, mood:subjunctive, number:pl, person:1]).
lex(rogent, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogent, tense:present, kind:_, mood:subjunctive, number:pl, person:3]).
lex(rogentur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogentur, tense:present, kind:_, mood:subjunctive, number:pl, person:3]).
lex(roger, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:roger, tense:present, kind:_, mood:subjunctive, number:sing, person:1]).
lex(rogeris, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogeris, tense:present, kind:_, mood:subjunctive, number:sing, person:2]).
lex(roges, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:roges, tense:present, kind:_, mood:subjunctive, number:sing, person:2]).
lex(roget, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:roget, tense:present, kind:_, mood:subjunctive, number:sing, person:3]).
lex(rogetis, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogetis, tense:present, kind:_, mood:subjunctive, number:pl, person:2]).
lex(rogetur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:middle, txt:rogetur, tense:present, kind:_, mood:subjunctive, number:sing, person:3]).
lex(rogetur, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogetur, tense:present, kind:_, mood:subjunctive, number:sing, person:3]).
lex(rogo, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:act, txt:rogo, tense:present, kind:_, mood:indicative, number:sing, person:1]).
lex(rogor, v, [pos:v, class:tr_cod, type:finite, lex:rogare, voice:pass, txt:rogor, tense:present, kind:_, mood:indicative, number:sing, person:1]).

Appendix I ALP reads Augustine … during a coffee break

AUGUSTINE GOES TO CHURCH TO BUY THE SERVICES OF A PROSTITUTE

Conf. 3, III,5, 6-9 Budé 1933 de Labriolle
"*Ausus sum etiam in celebritate sollemnitatum tuarum intra parietes ecclesiae tuae concupiscere et agere negotium procurandi fructus mortis*"

De Labriolle: "N'ai-je pas osé, en pleine célébration de vos solennités, dans l'enceinte de votre église, convoiter des fruits de mort et négocier le moyen de me les procurer?" (qu'en termes élégants...)

Arnaud d'Andilly: "Mon impudence passa même jusqu'à ce point, qu'en l'une de vos fêtes les plus solennelles, et dans votre propre église, j'osai concevoir un désir damnable et ménager un accord funeste qui ne pouvoit produire que des fruits de mort."
(no marks for concision...)

William Watts (Loeb Classical Library): "I was so bold one day, as thy solemnities were a celebrating, even within the walls of thy Church, to desire and to execute a business, enough to purchase me the very fruits of death."
(not very clear...)

De Labriolle (rightly, it seems to me) parses 'fructus mortis' as the object of both 'concupiscere' and 'procurandi' - that's well beyond the power of ALP !!!

We use a somewhat simpler version:

Ausus sum in ecclesia tua concupiscere fructus mortis et agere negotium procurandi eos.

[0/ausus,1/sum,2/in,3/ecclesia,4/tua,5/concupiscere,6/fructus,7/mortis,8/et,9/agere,10/negotium,11/procurandi,12/eos,endpos(13)]

cputime : 801 sec i.e. nearly a quarter of an hour !!

```
illocutionary_force:statement
vg
  selected_reading:audeo_dare
  polarity:pos
  cat:vg
  * ausus * * sum *
  lex:audere
  person:1
  mood:indicative
  tense:perfect
  voice:act
  number:sing
  gender:masc
subject
  source:context_retrievable
  number:sing
  gender:or([masc,fem])
  person:1
  cat:np
  index:i(0,0)
  constraints_to_be_met:[case:or([nom,acc]),sem:[hum]]
  case:nom
object
  cat:pred
  mood:infinitive
  * in * * ecclesia * * tua * * concupiscere * * fructus * * mortis * * et * * agere * * negotium * * procurandi * * eos *
  case:or([nom,acc])
  constituent_structure
    head
      vg
        selected_reading:concupisco_desire
        polarity:pos
```

cat:vg
* concupiscere *
pos:v
lex:concupiscere
voice:act
tense:present
mood:infinitive
object
* fructus * * mortis *
index:i([p(6,7)])
sem:[thing,abstract]
number:pl
person:3
cat:np
lex:fructus
gender:masc
case:acc
constituent_structure
head
* fructus *
index:i(p(6,7))
cat:np
sem:[thing,abstract]
number:pl
person:3
gender:masc
lex:fructus
case:acc
constituent_structure
fructus
noun_cplt
* mortis *
index:i(p(7,8))
cat:np
sem:[abstract,thing,hum]
number:sing
person:3
gender:fem
lex:mors
case:gen
constituent_structure
mortis
prep_phrase_adjunct_3
* in * * ecclesia * * tua *
index:i(p(3,4))
case:abl
prep:in
sem:[hum,thing,loc]
lex:ecclesia
cat:pp
constituent_structure
prep:in
head
* ecclesia * * tua *
index:i(p(3,4))
cat:np
sem:[hum,thing,loc]
number:sing
person:3
gender:fem
lex:ecclesia
case:abl
constituent_structure
head
* ecclesia *
pos:noun
lex:ecclesia
case:abl
gender:fem
number:sing
sem:[hum,thing,loc]
adjp
cat:adjp
* tua *
case:abl
number:sing
gender:fem
lex:tuus
constituent_structure
tuus
coord:et
head
vg
selected_reading:negotium_ago_take_care_of_IDIOM
polarity:pos
cat:vg
* agere *
pos:v
lex:agere
voice:act

```
      tense:present
      mood:infinitive
   object
      * negotium *
      index:i(p(10,11))
      cat:np
      sem:[abstract]
      number:sing
      person:3
      gender:neuter
      lex:negotium
      case:acc
      constituent_structure
      negotium
   object_cplt
      cat:pred
      mood:gerund
      local_case:gen
      number:sing
      person:3
      gender:neuter
      * procurandi * * eos *
      constituent_structure
         vg
            selected_reading:procuro_procure
            pos:v
            lex:procurare
            mood:gerund
            person:3
            case:gen
         object
            * eos *
            index:i(p(12,13))
            cat:np
            lex:prpersaccmascpl
            number:pl
            person:3
            gender:masc
            case:acc
            constituent_structure
            prpersaccmascpl
```

(version : alp192)

Such a result is to be seen in an optimistic light – ALP plods on, but comes up with a correct parse. Much better than just giving up, or, worse, turning out rubbish.

It should be kept in mind that the most time-consuming phrases (in terms of parsing, of course) are the ones that are not bound to any argument but function as clause-level adjuncts. As a general rule, a fifteen-word limit imposed on strings to be parsed seems reasonable. Consider the following data based on Livius, *Ab Urbe Condita*, Liber XXI, 8:

Per totum tempus hiemis quies inter labores iam exhaustos aut mox exhauriendos renovavit corpora animosque ad omnia de integro patienda.

Quies renovavit corpora animosque.
5 words
cputime : 0.08

Quies inter labores renovavit corpora animosque.
7 words
cputime : 0.7

Quies inter labores renovavit corpora animosque ad omnia patienda.
10 words
cputime : 15.5

Quies inter labores renovavit corpora animosque ad omnia de integro patienda.
12 words
cputime : 18.7

Quies inter labores exhaustos aut exhauriendos renovavit corpora animosque ad omnia de integro patienda.

15 words
cputime : 229 i.e. nearly four minutes

The parses produced I believe to be correct. Here is the one associated with the 15-word string:

```
 5--->

   illocutionary_force:statement
   vg
      selected_reading:renouo_renew
      polarity:pos
      cat:vg
      * renouauit *
      pos:v
      lex:renouare
      voice:act
      tense:perfect
      mood:indicative
      number:sing
      person:3
   subject
      number:sing
      gender:fem
      * quies *
      index:i(p(0,1))
      cat:np
      sem:[abstract]
      person:3
      lex:quies
      case:nom
      constituent_structure
      quies
   object
      * corpora * * que * * animos *
      index:i([p(7,8),p(9,10)])
      distance:[1]
      sem:[thing]
      lex:corpus
      number:pl
      person:3
      gender:masc
      case:acc
      coord:yes
      constituent_structure
         head
            * corpora *
            index:i(p(7,8))
            cat:np
            sem:[thing]
            number:pl
            person:3
            gender:neuter
            lex:corpus
            case:acc
            constituent_structure
            corpora
         coord:que
         head
            * animos *
            index:i(p(9,10))
            cat:np
            sem:[abstract,thing]
            number:pl
            person:3
            gender:masc
            lex:animus
            case:acc
            constituent_structure
            animos
```

adjunct
* ad * * omnia * * de * * integro * * patienda *
value:purpose
constituent_structure
prep:ad
head
cat:pred
subtype:gerundive
case:acc
mood:gerund
local_case:acc
number:sing
person:3
gender:neuter
* omnia * * de * * integro * * patienda *
constituent_structure
vg
selected_reading:patior_bear
pos:gdiv
case:acc
gender:neuter
number:pl
lex:pati
mood:gerund
person:3
object
* omnia *
index:i(p(11,12))
cat:np
sem:[thing,abstract]
lex:omnis
number:pl
person:3
gender:neuter
case:acc
constituent_structure
omnis
adjunct
* de * * integro *
value:manner_means
constituent_structure
* de * * integro *
lex:de_integro
value:manner_means
prep_phrase_adjunct_1
* inter * * labores * * exhaustos * * aut * * exhauriendos *
index:i(p(2,3))
case:acc
prep:inter
sem:[abstract]
lex:labor
cat:pp
constituent_structure
prep:inter
head
* labores * * exhaustos * * aut * * exhauriendos *
index:i(p(2,3))
cat:np
sem:[abstract]
number:pl
person:3
gender:masc
lex:labor
case:acc
constituent_structure
head
* labores *
pos:noun
lex:labor
case:acc

```
      gender:masc
      number:pl
      sem:[abstract]
   adjp
      cat:adjp
      * exhaustos * * aut * * exhauriendos *
      case:acc
      number:pl
      gender:masc
      lex:exhaustos
      constituent_structure
         head
            cat:adjp
            * exhaustos *
            case:acc
            number:pl
            gender:masc
            lex:exhaustos
            morph:ppt
         coord:aut
         head
            cat:adjp
            * exhauriendos *
            case:acc
            number:pl
            gender:masc
            lex:exhauriendos
            constituent_structure
            exhauriendos
```

(version : alp192)

Appendix II How to Use ALP

ALP is made up of three SWI-Prolog programs :

alpxxx.pl : main program (xxx to be replaced by version number, e.g. alp189.pl)
template.pl : lexical entries for *argument bearers*, mainly verbs
makelex.pl : program building 'vocfile', i.e. the morphological variants needed by ALP

Once SWI-Prolog has been properly installed, ALP is ready for use.

Both *alpxxx* and *makelex* are executables, with a GO step. To execute the programs, simply type in : *go.* (don't forget the final dot). In the case of *makelex*, press the *Enter key* once the sytem has replied with '*true*'.

To get out of a Prolog session, type : halt. (again, don't forget the dot!)

On execution, ALP will load both the *vocfile* produced by *makelex.pl* and the *template.pl* file, provided the files are reachable by ALP.

The easiest way to use ALP, therefore, is to have a single directory containing both the executable files (*alpxxx.pl* and *makelex.pl*), the *template.pl* file, the *vocfile* and the test files that one wishes to submit to the parsing process.

If no testfile is available, *stdin* can be used as input stream. The sentences one wishes to parse are then to be entered directly from the keyboard, and should end with either a *dot*, a *question mark*, or a *slash* (see below). To get out of the input loop type : *stop.* (mind the dot !). And to get out of Prolog, remember : *halt.* (nothing to say about the dot?)

The test file names should preferably be made up of a single atom, such as *test*, *testfile*, *mytest*, *alptest*, etc.

The test files must end on a line that is made up of the single word *stop* followed by a dot, i.e. *stop.*

The result file name (i.e. the name of the file containing the parses) should also be a single atom, to which the system will add a *.lst* extension. The filename selected by the user for output can therefore be the same as for input (*test* → *test.lst*).

Each sentence in a test file should end with a final punctuation sign, either dot (.),question mark (?), or a slash (/).The question mark is not used by the parser as an indication that it should parse the string as a question. The slash is used for partial structures, as in :

% 20.
Ut amicitia societasque nostra in aeternum rata sint /
% 21.
postulamus a uobis muros coloniae munimenta servitii detrahatis.

Commas are best left out, unless they are used to isolate an *addressee* (vocative), as in :

Argiletanas mauis habitare tabernas cum tibi, parue liber, scrinia nostra vacent.
Incipe, parve puer, risu cognoscere matrem.
Hanc etiam vix, Tityre, duco.

Or to guide *distributed* structures as in:

Contemnunt novitatem meam, ego illorum ignaviam.
Mihi fortuna, illis probra obiectantur.
Nulli parietes nostram salutem, nullae leges, nulla iura custodient.
Insidiatores ab Othone in Germaniam, a Vitellio in urbem missi.
Versabitur in rostris furor, in curia timor, in foro coniuratio, in Campo exercitus, in agris vastitas.

Comment lines open with the percentage sign (*%*) and close with a dot (.). No dot can be used within a comment line. See the test file for examples.

The following settings are made available in the ALP settings menu:

Depth Level for Finite Pass
1.Shallow % default
2.Intermediate
3.Deep

Type of Parses Displayed
1.Whole Sentences Only % default
2.Phrases and Clauses

Number of Parses Displayed
0.one parse only
1.one parse only, or two if the second parse has the same ranking as the first % default
2.two parses, whatever the ranking of the second
3.all available parses

The Prolog programs are *ascii* files, and can therefore be explored with a large number of tools. The editor within SWI-Prolog is a wise choice (type in *edit.* once the program has loaded).

The *vocfile* produced by *makelex* is also an ascii file. It can be sorted and explored outside of Prolog. If the sorted file is given a *.pl* extension, it will also be explorable by Prolog :
sort vocfile > alplex.pl

To be able to add vocabulary to ALP one should make a careful study of the lexical items that ALP already features. We should emphasize that there is no point in adding a verb to *makelex* without giving it an argument structure in *template*. Only then can it be used in a sentence to be parsed (see next section).

Adding to ALP

To add new grammar rules one has to know both Prolog and ALP rather intimately.
But adding vocabulary to ALP is quite feasible without having to enter the arcanes or bowels of the beast.

Note first that in the vocabulary to be entered in *makelex* no use should be made of letter *v* (*u* is to be used instead) and capital letters are banned. The entry for *Ariovistus* reads :

noun(2,masc,ariouistus,ariouist,class:proper,ab:no, sem:[male],[nb:sg]).

When entering text to be parsed *v* and capital letters can be used freely :

Ariovistum se consule cupidissime populi Romani amicitiam appetisse. (Caesar, BG, I, XL, 1)
(Note that ALP will recognise *appetisse* as variant of *appetiuisse*).

Suppose we want ALP to be able to parse
Memorabantur exempla maiorum qui iuventutis irreverentiam gravibus decretis notavissent. (adapted from Tac ANN III 31 4)

Perusing the *makelex.pl* file enables us to add the missing lexical entries.

On the model of

noun(1,fem,impudentia,impudenti,class:common, ab:mm, sem:[quality],[]).
we add:
noun(1,fem,irreuerentia,irreuerenti,class:common, ab:mm, sem:[quality],[]).
and on the model of
noun(2, neuter, edictum, edict,class:common, ab:mm, sem:[thing,abstract],[]).
we add:
noun(2, neuter, decretum, decret,class:common, ab:mm, sem:[thing,abstract],[]).

Similarly, on the basis of
verb([v(renouare,1,renou,renouau,renouat)],tr_cod,std). % standard transitive verb first conjugation
we create the new lexical entries:
verb([v(memorare,1,memor,memorau,memorat)],tr_cod,std).
verb([v(notare,1,not,notau,notat)],tr_cod,std).

The latter, being verb entries, need to be associated with the right template.

Looking at the template file (*template.pl*) we find the template for transitive verbs with human subject and adjunct of a specifiable case :

```
lexarg(Xre,
    arglist:[ws(v_v,tr_cod,clause:[],mwuw:0,
        args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
            object:[type:np,oblig:no,constraints:[case:acc]],
            adjunct:[type:np,oblig:no,constraints:[case:CASE]]])]).
```

We use this template to build the template we associate with the *notare* we are interested in:

```
lexarg(notare,
    arglist:[ws(noto_blame,tr_cod,clause:[],mwuw:0,
        args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
            object:[type:np,oblig:yes,constraints:[case:acc]],
            adjunct:[type:np,oblig:yes,constraints:[case:abl]]])]).
```

Note that we have specified the obligatory character of the ablative adjunct, which will help us distinguish between this reading of *noto* and others we might wish to add in the future.

For *memoro* we use the template for transitive verbs with human subject:

lexarg(memorare,
arglist:[
ws(memoro_recall,tr_cod,clause:[],mwuw:0,
args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
object:[type:np,oblig:yes,constraints:[case:acc]]])]).

Once the entries have been added to *makelex.pl* and the templates to *template.pl*, we run *makelex* to build the new vocfile.

1 ?- go.

MAKELEX, building vocabulary file for ALP

Expanding the lexical macros...

Done
true .

(here we press the Enter key and then stop Prolog with : halt.)

Makelex has produced a new *vocfile*, which will henceforth be used by ALP.

We can now run the current version of ALP (alp187.pl at the time of writing)

Once ALP is launched, we are ready to enter the sentence when prompted by ALP to do so. We can use *stdin* as input stream to enter the sentence using the keyboard and select a filename for the results (below we specify *additions*, which will result in the production of *additions.lst*, which will house the resulting parse or parses.

We shouldn't forget the end dot...

1 ?- go.

ALP, a data-driven feature-unification parser for a subset of classical Latin

Current settings are: DEFAULT. Modify ? (y/n)

Input file? [stdin. or file_name.] --> stdin.
Output file? [file_name.] --> |: additions.

Key in your sentence or stop. to quit
|: Memorabantur exempla maiorum qui iuventutis irreverentiam gravibus decretis notavissent.

[0/memorabantur,1/exempla,2/maiorum,3/qui,4/irreuerentiam,5/grauibus,6/decretis,7/notauissent,endpos(8)]
cputime : 1.3125000000000009

1.5--->
[parse_list]

illocutionary_force:statement
vg
selected_reading:memoro_recall
polarity:pos
cat:vg
* memorabantur *
pos:v
lex:memorare
voice:pass
tense:imperfect
mood:indicative
number:pl
person:3
subject
number:pl
gender:neuter

* exempla * * maiorum * * qui * * irreuerentiam * * grauibus * * decretis * * notauissent *
index:i([p(1,2)])
sem:[thing,abstract]
person:3
cat:np
lex:exemplum
case:nom
c_str
  head
    * exempla *
    index:i(p(1,2))
    cat:np
    sem:[thing,abstract]
    number:pl
    person:3
    gender:neuter
    lex:exemplum
    case:nom
    c_str
    exempla
  noun_cplt
    * maiorum * * qui * * irreuerentiam * * grauibus * * decretis * * notauissent *
    cat:np
    index:i(p(2,3))
    number:pl
    gender:masc
    sem:[hum]
    case:gen
    lex:maiores
    c_str
      head
      maiorum
      rel_clause
        * qui * * irreuerentiam * * grauibus * * decretis * * notauissent *
        index:i(p(2,3))
        number:pl
        gender:masc
        case:nom
        person:3
        reltype:restrictive
        mood:subjunctive
        tense:pluperfect
        c_str
          illocutionary_force:statement
          vg
            selected_reading:noto_blame
            polarity:pos
            cat:vg
            * notauissent *
            pos:v
            lex:notare
            voice:act
            tense:pluperfect
            mood:subjunctive
            number:pl
            person:3
          subject
            index:i(p(2,3))
          object
            * irreuerentiam *
            index:i(p(4,5))
            cat:np
            sem:[quality]
            number:sing
            person:3
            gender:fem
            lex:irreuerentia
            case:acc
            c_str
            irreuerentiam

```
adjunct
   * grauibus * * decretis *
   index:i(p(6,7))
   cat:np
   sem:[thing,abstract]
   number:pl
   person:3
   gender:neuter
   lex:decretum
   case:abl
   c_str
      head
         * decretis *
         pos:noun
         lex:decretum
         case:abl
         gender:neuter
         number:pl
         sem:[thing,abstract]
      adjp
         cat:adjp
         * grauibus *
         case:abl
         number:pl
         gender:neuter
         lex:grauis
         c_str
         grauis
```

[0/stop,endpos(1)] % on entering *stop.*

TOTAL TIME : 1.3125000000000009

To get out of Prolog : *halt.*

Note the use of the index

subject
index:i(p(2,3))

making reference to the word list :
[0/memorabantur,1/exempla,2/maiorum,3/qui,4/irreuerentiam,5/grauibus,6/decretis,7/notauissent,endpos(8)]

to pick *maiorum* as the antecedent of *qui* and subject of *notauissent*.

Appendix III Parsing and Word Sense Assignment

On top of revealing syntactic structure a parser should attempt to assign word senses to the lexical items that make up its leaves. It should not stop at the lemmatisation stage, in so far as word sense recognition ties up with structural assignment.

Wordsenses do not exist out there : their status as lexicographical constructs is not in doubt. But recognizing their true nature as constructs should not detract from acknowledging their usefulness in the accomplishment of a number of NLP-oriented tasks, translation being an outstanding example.

As a matter of fact, any description of sense is a construct. There is a lot to be said in favour of keeping the metalinguistic description to a minimum, so that belonging to a WordNet *synset* may be the most adequate way of specifying a wordsense. Each word sense of a lexical item should give rise to assignment to a different synset. The synset glosses should serve for orientation only, and are not to be confused with definitions. Definitions are part of the exploitable lexicographical description, as will be shown below.

A good starting point is an examination of the wordsenses assigned by the body of lexicographical work for a given language, in a monolingual perspective first, but without rejecting a bilingual or multilingual framework (translation-oriented dictionaries).

The question of granulatity is likely to be settled by the range and depth of the analysis tools that we are able to devise.

For the items that the parser recognizes as *argument bearers*, we shall need to specify word sense descriptions for the arguments as well as for the arg bearer iself. We shall have to proceed step by step. In a first stage, we suggest that an argument can be described by a *lexical world*, i.e. a set of lemmas gleaned mainly from the examples offered by the lexicographical ressources. See below for two wordsenses of *colo* as described in *Forcellini* and *Lewis and Short*.

A *lexical world* should exclude *toolwords* which we can store in a *stoplist*, in order to be able to concentrate on full lexical items. In specific cases, we may wish to register wordforms instead of lemmas, and we should be ready to house descriptions of any degree of specificity in the case of *multi-word units* (phraseological component).

Once our tools are powerful enough, we can use them in a bootstrapping procedure by applying them to the dictionary-registered citations, attempting to bring the granularity level of our analysis down to word senses.

In the parsing process, to work out whether an item in the string to be parsed potentially filling an arg position is to be parsed as filling that position indeed, we have to measure the lexical distance between the item in text and the lemmas belonging to the lexical world assigned to that arg position in the description the parser makes use of.

In the standard case, where the arg bearer has a number of word senses, each word sense will entail a specific description for the args, and the measure of lexical distance should lead to the selection of the appropriate word senses, both at the level of the arg and at that of the arg bearer.

Lexical distance can be equated with the length of the path that needs to be covered for two items to meet along synset and synset-extending relations (hyponymy, etc). In our case, the two items should

be pairs resulting from the pairing of the item found in the text to be parsed and each member of the lexical world associated with the arg under a given wordsense. The lexical distance to be used for decision taking with respect to wordsense selection would be based on a mean value computed on all the pairs.

We would need to specify limits to the exploration of lexical distance. If the two items do not meet on a path of length *l* they will be deemed to be unrelated (although they may of course be related at a higher level).

Two word senses of COLO in Forcellini

Colo,colare vs colo,colere : the distinction is in most cases to be decided on the basis of lemmatisation ; if necessary (parsing the very form *colo*), on the basis of the object arg.
Let's concentrate on c*olo, colere.*

We should exploit, not only the examples, but also the definitions and semantic characterization with respect to *habitare*, *observare*, *venerari*.

The *lexical worlds* can be hand-crafted or, more roughly, computer-derived. As a matter of fact, if we insert the missing headword in the examples where it does not occur, and decorate it with the appropriate wordsense number, or simply add the wordsense number where the headword occurs, and then build up a concordance excluding stopwords, we would have a most useful tool for specifying the lexical worlds that we have in mind.

Definition : **studium**, **operam**, **laborem** pono in re aliqua **perficienda**, **assiduus** sum circa rem aliquem, **exerceo**, **excolo**, **curo**

(It. *lavorare, coltivare, esercitare;* Fr. *travailler, cultiver, soigner;* Hisp. *labrar, cultivar;* Germ. *etwas treiben, betreiben, um etwas herum sein Geschäft treiben;* Angl. *to exercise, labour upon, cultivate*).

We concentrate on the object arg for the two wordsenses that ALP has so far been concerned with :

```
lexarg(colere,
       arglist:[ws(colo_inhabit,tr_cod,clause:[],mwuw:0,
                args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                      object:[type:np,oblig:yes,constraints:[case:acc, sem:[loc]]]]),

                ws(colo_practise,tr_cod,clause:[],mwuw:0,
                args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                      object:[type:np,oblig:yes,constraints:[case:acc, sem:[abstract]]]])]).
```

¶ 2. Quoniam qui agros colunt, ibidem **permanere** et **habitare** necessario debent, (qua in re populi agricolae a pastoribus distinguuntur); vel etiam quoniam loco, in quo habitamus, curam impendimus: factum est, ut colere ponatur saepissime pro habitare, quamvis aliquantulum ab ipso differat: *V.* Homonym. in fin. Occurrit autem -

a) Cum Accusativo loci. *Plaut. Aulul. prol.* 3. Hanc domum jam multos annos est cum possideo et colo. *Id. Bacch* 2. 2. 21. Colere regiones Acberunticas. *Cic.* 2. *Fam.* 12. 2. Urbem, urbem, mi Rufe, cole et in ista luce vive. *Forcellinus* recte hîc accepit *colere* pro frequentem et assiduum esse in loco aliquo; quod tamen idem est atque habitare. *Id.* 2. *Nat. D.* 66. 164. Homines, qui has nobiscum terras ab oriente ad occidentem colunt. Adde *Tac.* 2. *Ann.* 60. *Lucret.* 5. 953. Colere nemora, montes

silvasque. *Catull.* 63. 70. Idae loca. *Virg.* 4. *AEn.* 343. urbem Trojanam. *Id.* 5. *ibid.* 735. Elysium. *Ovid.* 4. *Fast.* 487. Unaque, pastorem vidisset an arva colentem, Vox erat. *Id.* 11. *Met.* 146. Colere silvas et rura. *Curt.* 4. 8. 6. Urbem advenarum frequentia cultura. *Id.* 7. 7. 4. Colere regionem. *Id.* 9. 2. 3.; et *Tac.* 1. *Ann.* 59. et *Germ.* 28. **ripam.** - Et de Poetis *Propert.* 3. 5. 41. Me juvat in primâ coluisse Helicona juventâ. - De brutis vero *Virg.* 3. *G.* 430. Anguis stagna colat. *Ovid.* 2. *Met.* 380. Quae colat elegit (*cygnus*) contraria flumina flammis. -

Lexical world for this word sense (roughly : inhabit*): {domus, regio, urbs, terra, nemus, mons, silva, locus, Elysium, arvum, rus,* ***ripa****, Helicon, stagnum, flumen}*

Second word sense : roughly *honour*

¶ 2. Saepissime, morali ratione, de rebus ponitur, et est **curam** habere, **fovere**, **plurimi facere**, **honorare**. *Cic.* 1. *Off.* 2. 4. In officio colendo sita est vitae honestas omnis, in negligendo turpitudo. *Ovid.* 2. *Art. am.* 121. ingenuas pectus coluisse per artes Cura sit. - Hac significatione jungitur pluribus Substantivis, quorum praecipua, servato litterarum ordine, exhibentur. *Plaut. Men.* 4. 2. 10. aequum et bonum. *Lucret.* 5. 1144. aevum. *h. e.* aetatem agere. *Cic.* 3. *Herenn.* 3. 4. affinitates. *Plaut. Cist.* 1. 1. 23. et *Cic.* 1. *Off.* 2. 5. amicitiam. *Sall. Jug.* 8. amicitiam populi Romani. *Cic.* 2. *Fin.* 26. 83. et *Amic.* 22. 35. amicitias. *Curt.* 8. 2. 32. amicitiam cum fide. *Virg.* 11. *AEn.* 584. et *Tibull.* 2. 4. 52. amorem. *Cic.* 3. *Fam.* 13.; *Ovid.* 2. *Fast.* 508. et *Sueton. Tib.* 60. artes. *Plaut. Trin.* 2. 2. 16. artes malas. *Propert.* 3. 13. 48. aurum victâ pietate. *Liv.* 7. 30. beneficium acceptum. *Cic.* 3. *Herenn.* 3. 4. clientelas, cognationes. *Id.* 1. *Off.* 41. 149. conciliationem et consociationem communem totius generis hominum. *Id.* 2. *Fin.* 31. 102. diem natalem alicujus. *Id.* 2. *Orat.* 35. 148. diligentiam. *Id. Brut.* 31. 117. disciplinam aliquam. *Plaut. Capt.* 2. 1. 26. dolos astu. *Cic.* 1. *Invent.* 2. 3. et *Curt.* 7. 8. fidem. *Ovid.* 1. *Met.* 90. fidem rectumque. *Curt.* 10. 3. 9. fortitudinem. *Cic.* 1. *Off.* 1. 3. genus orationis aequabile et temperatum. *Id.* 3. *Herenn.* 3. 4. hospitia. *Id. ibid.* et 1. *Legg.* 12. 33. jus. *Liv.* 27. 17. jus et fas. *Cic.* 1. *Off.* 2. 5. justitiam et liberalitatem. Adde *eumd.* 2. *Tusc.* 13. 31. et 3. *Fin.* 22. 71. *Id. Parad.* 5. 1. . 34. leges sequi et colere. *Sil. It.* 8. 464. Martem. *h. e.* bellum. *Cic.* 2. *Fin.* 31. 101. memoriam defuncti. *Id. post red. ad Quirit.* 10. 24. memoriam beneficii colere benevolentia sempiterna. *Ovid.* 2. *Fast.* 518. militiam. *Plin. Paneg.* 1. morem. *h. e.* sequi. *Plaut. Cas.* 5. 4. 1. mores Massilienses. *Sall. Cat.* 9. mores bonos. *Martial.* 9. 12. Musas severiores. *Cic.* 1. *Legg.* 5. 16. munus aliquod colere et efficere. *Plaut. Stich.* 1. 1. 35. officium suum. *Liv.* 45. 8. et *Ovid.* 11. *Met.* 297. pacem. *Plaut. Asin.* 3. 1. 6.; *Ter. Hecyr.* 3. 4. 33.; et *Cic.* 3. *Off.* 21. 82. pîetatem. *Cic.* 5. *Verr.* 3. 6. pudorem et pudicitiam. *Plaut. Poen.* 5. 2. 137. quaestum suum. *Cic. Flacc.* 4. 9. religionem et fidem testimoniorum. Cf. *eumd. Fontej.* 10. 21. *Plaut. Poen.* 4. 2. 7. servitutem colere apud aliquem. *h. e.* servire. *Cic.* 3. *Herenn.* 3. 4. societates. *Id.* 5. *Fin.* 19. 50. et 3. *Fam.* 13. studia. *Id. Brut.* 91. 315. studium philosophiae. *Id.* 12. *Att.* 28. nec victum, nec vitam. *Id. fragm.* apud *Non.* 4. 280. *edit. vet.* victum lautum et elegantem. *Cic. Arch.* 7. 16. virtutem percipere et colere. *Plaut. Rud.* 1. 5. 25. vitam. *h. e.* agere, seu vivere. Adde *eumd. Trin.* 3. 2. 74. Sic *Ter. Heaut.* 1. 1. 84. vitam inopem colere. Adde *Cic.* 4. *Herenn.* 14. 21. et 12. *Att.* 28.; et *Virg.* 3. *G.* 532.

Lexical world 1: {officium, aequum, bonum, affinitas, amicitia, amor, ars, aurum, beneficium, clientela, cognatio, fides, fortitudo, hospitium, ius, fas, lex, Mars, bellum, memoria, militia, mos, Musa, munus, pax, pietas, pudor, pudicitia, quaestus, religio, servitus, societas, studium}

Perhaps we should isolate a lexical world to be associated with a multi-word unit :
Lexical world 2: {aevum, aetas, vita, militia, victus} : basis for a semi-open mwu : *vitam colere/agere ?* The decision depends on whether *militiam colere* is ambiguous with respect to *militia* as military *service* vs military *spirit*, with *colo* as *spend/pass* vs *cultivate/cherish* (see Lewis and Short)

Colo in Lewis and Short

(the building of the relevant lexical worlds is left to the reader)

1. **cŏlo, colŭi, cultum, 3, *v. a.* [from the stem ΚΟΛ, whence βουκόλος, βουκολέω; cf.: colonus, in-cola, agri-cola] (orig. pertaining to agriculture), *to cultivate, till, tend, take care of a field, garden,* etc. (freq. in all per. and species of composition).**

I. Prop. 1. **A.** 1. (α) With *acc.:* fundum, Varr. R. R. 1, 1, 2: agrum, id. ib. 1, 2, 14; Cato, R. R. 61; Col. 1 pr.: agri non omnes frugiferi sunt qui coluntur, Cic. Tusc. 2, 5, 13; id. Agr. 2, 25, 67: arva et vineta et oleas et arbustum, Quint. 1, 12, 7: praedia, Cic. Rosc. Am. 17, 49: rus, Col. 1, 1: rura, Cat. 64, 38; Tib. 1, 5, 21; Verg. G. 2, 413: hortos, Ov. M. 14, 624 al.: jugera, Col. 1 pr.: patrios fines, id. ib.: solum, id. 2, 2, 8: terram, id. 2, 2, 4: arbustum, Quint. 1, 12, 7: vitem, Cic. Fin. 4, 14, 38: arbores, Hor. C. 2, 14, 22: arva, id. ib. 3, 5, 24; Ov. Am. 1, 13, 15: fructus, Verg. G. 2, 36: fruges, Ov. M. 15, 134: poma, id. ib. 14, 687; cf. under *P. a.*

2 (β) ***Absol.*,** Varr. R. R. 1, 2, 8; Verg. G. 1, 121; Dig. 19, 2, 54, § 1.

1. **B.** In gen., without reference to economics, *to abide, dwell, stay in a place, to inhabit* (syn.: incolo, habito; most freq. since the Aug. per.).

1. (α) With *acc.:* hanc domum, Plaut. Aul. prol. 4: nemora atque cavos montes silvasque colebant, Lucr. 5, 955: regiones Acherunticas, Plaut. Bacch. 2, 2, 21: colitur ea pars (urbis) et habitatur frequentissime, Cic. Verr. 2, 4, 53, § 119: urbem, urbem, mi Rufe, cole, id. Fam. 2, 12, 2: has terras, id. N. D. 2, 66, 164; Tac. A. 2, 60: loca Idae, Cat. 63, 70: Idalium, id. 36, 12 sq.; 61, 17: urbem Trojanam, Verg. A. 4, 343: Sicaniam, Ov. M. 5, 495: Maeoniam Sipylumque, id. ib. 6, 149: Elin Messeniaque arva, id. ib. 2, 679: regnum nemorale Dianae, id. ib. 14, 331: hoc nemus, id. ib. 15, 545: Elysium, Verg. A. 5, 735: loca magna, Ov. M. 14, 681; Liv. 1, 7, 10: Britanniam, Tac. Agr. 11: Rheni **ripam**, id. G. 28: victam **ripam**, id. A. 1, 59: terras, id. ib. 2, 60; cf. id. H. 5, 2: insulam, id. A. 12, 61; id. G. 29: regionem, Curt. 7, 7, 4.
Poet., of poets: me juvat in primā coluisse Helicona juventā, *i. e. to have written poetry in early youth,* Prop. 3 (4), 5, 19.
Also of animals: anguis stagna, Verg. G. 3, 430; Ov. M. 2, 380.

2. (β) ***Absol.*:** hic, Plaut. Ps. 1, 2, 68: subdiu colere te usque perpetuom diem, id. Most. 3, 2, 78; Liv. 42, 67, 9; Curt. 9, 9, 2: colunt discreti ac diversi, Tac. G. 16: proximi Cattis Usipii ac Tencteri colunt, id. ib. 32: circa utramque ripam Rhodani, Liv. 21, 26, 6: quā Cilices maritimi colunt, id. 38, 18, 12: prope Oceanum, id. 24, 49, 6: usque ad Albim, Tac. A. 2, 41: ultra Borysthenem fluvium, Gell. 9, 4, 6: super Bosporum, Curt. 6, 2, 13: extra urbem, App. M. 1, p. 111.

II. Trop. (freq. and class.).

1. **A.** *To bestow care upon a thing, to care for.*

1. **1.** Of the gods: colere aliquem locum, *to frequent, cherish, care for, protect, be the guardian of,* said of places where they were worshipped, had temples, etc.: deos deasque veneror, qui hanc urbem colunt, Plaut. Poen. 5, 1, 19; Cat. 36, 12: Pallas, quas condidit arces, Ipsa colat, Verg. E. 2, 62: ille (Juppiter) colit terras, id. ib. 3, 61; id. A. 1, 16 Forbig. ad loc.: undis jura dabat, nymphisque colentibus undas, Ov. M. 1, 576: urbem colentes di, Liv. 31, 30, 9; 5, 21, 3: vos, Ceres mater ac Proserpina, precor, ceteri superi infernique di, qui hanc urbem colitis, id. 24, 39, 8: divi divaeque, qui maria terrasque colitis, id. 29, 27, 1.
2. **2.** Rarely with persons as object (syn.: curo, studeo, observo, obsequor): Juppiter, qui genus colis alisque hominum, Plaut. Poen. 5, 4, 24; cf.: (Castor et Pollux) dum terras hominumque colunt genus, *i. e. improve, polish,* Hor. Ep. 2, 1, 7.
3. **3.** Of the body or its parts, *to cultivate, attend to, dress, clothe, adorn,* etc.: formamque augere colendo, *by attire, dress,* Ov. M. 10, 534: corpora, id. A. A. 3, 107: tu quoque dum coleris, id. ib. 3, 225.
With *abl.:* lacertos auro, Curt. 8, 9, 21: lacertum armillā aureā, Petr. 32: capillos, Tib. 1, 6, 39; 1, 8, 9.
4. **4.** With abstr. objects, *to cultivate, cherish, seek, practise, devote one's self to,* etc.; of mental and moral cultivation: aequom et bonum, Plaut. Men. 4, 2, 10: amicitiam, id. Cist. 1, 1, 27: fidem rectumque, Ov. M. 1, 90: fortitudinem, Curt. 10, 3, 9: jus et fas, Liv. 27, 17 *fin.*: memoriam alicujus,

Cic. Fin. 2, 31, 101: bonos mores, Sall. C. 9, 1: suum quaestum colit, Plaut. Poen. 5, 2, 137: pietatem, id. As. 3, 1, 5; Ter. Hec. 3, 4, 33: virtutem, Cic. Arch. 7, 16; id. Off. 1, 41, 149: amicitiam, justitiam, liberalitatem, id. ib. 1, 2, 5: virginitatis amorem, Verg. A. 11, 584: pacem, Ov. M. 11, 297; cf. Martem, Sil. 8, 464: studium philosophiae, Cic. Brut. 91, 315: disciplinam, id. ib. 31, 117: aequabile et temperatum orationis genus, id. Off. 1, 1, 3: patrias artes militiamque, Ov. F. 2, 508; cf.: artes liberales, Suet. Tib. 60: ingenium singulari rerum militarium prudentiā, Vell. 2, 29, 5 Kritz.

5. **5.** Of a period of time or a condition, *to live in, experience, live through, pass, spend,* etc.: servitutem apud aliquem, *to be a slave,* Plaut. Poen. 4, 2, 7: nunc plane nec ego victum, nec vitam illam colere possum, etc., Cic. Att. 12, 28, 2; and **poet.** in gen.: vitam or aevum = degere, *to take care of life,* for *to live:* vitam, Plaut. Trin. 3, 2, 74; id. Cas. 2, 1, 12; id. Rud. 1, 5, 25: vitam inopem, Ter. Heaut. 1, 1, 84: aevum vi, Lucr. 5, 1144 and 1149.

2. **B.** Colere aliquem, *to regard one with care,* i. e. *to honor, revere, reverence, worship,* etc. (syn.: observo, veneror, diligo).

1. **1.** Most freq. of the reverence and worship of the gods, and the respect paid to objects pertaining thereto, *to honor, respect, revere, reverence, worship:* quid est enim cur deos ab hominibus colendos dicas? Cic. N. D. 1, 41, 115: hos deos et venerari et colere debemus, id. ib. 2, 28, 71; cf. id. ib. 1, 42, 119; id. Agr. 2, 35, 94; Liv. 39, 15, 2; Cat. 61, 48: Phoebe silvarumque potens Diana … o colendi Semper et culti, Hor. C. S. 2 and 3; cf. Ov. M. 8, 350: deos aris, pulvinaribus, Plin. Pan. 11, 3: Mercurium, Caes. B. G. 6, 17: Apollinem nimiā religione, Curt. 4, 3, 21: Cererem secubitu, Ov. A. 3, 10, 16: (deam) magis officiis quam probitate, id. P. 3, 1, 76: per flamines et sacerdotes, Tac. A. 1, 10; Suet. Vit. 1: quo cognomine is deus quādam in parte urbis colebatur, id. Aug. 70: deum precibus, Sen. Herc. Oet. 580: testimoniorum religionem et fidem, Cic. Fl. 4, 9; cf. id. Font. 10, 21; and: colebantur religiones pie magis quam magnifice, Liv. 3, 57, 7; and: apud quos juxta divinas religiones humana fides colitur, id. 9, 9, 4: sacra, Ov. M. 4, 32; 15, 679: aras, id. ib. 3, 733; 6, 208; cf. Liv. 1, 7, 10; Suet. Vit. 2 et saep.: numina alicujus, Verg. G. 1, 30: templum, id. A. 4, 458; Ov. M. 11, 578: caerimonias sepulcrorum tantā curà, Cic. Tusc. 1, 12, 27: sacrarium summā caerimoniā, Nep. Th. 8, 4: simulacrum, Suet. Galb. 4.

2. **2.** Of the honor bestowed upon men: ut Africanum ut deum coleret Laelius, Cic. Rep. 1, 12, 18: quia me colitis et magnificatis, Plaut. Cist. 1, 1, 23; Ter. Ad. 3, 2, 54: a quibus diligenter observari videmur et coli, Cic. Mur. 34, 70; cf. id. Fam. 6, 10, 7; 13, 22, 1; id. Off. 1, 41, 149; Sall. J. 10, 8: poëtarum nomen, Cic. Arch. 11, 27: civitatem, id. Fl. 22, 52; cf.: in amicis et diligendis et colendis, id. Lael. 22, 85 and 82: semper ego plebem Romanam militiae domique … colo atque colui, Liv. 7, 32, 16: colere et ornare, Cic. Fam. 5, 8, 2: me diligentissime, id. ib. 13, 25 *init.*: si te colo, Sexte, non amabo, Mart. 2, 55: aliquem donis, Liv. 31, 43, 7: litteris, Nep. Att. 20, 4: nec illos arte colam, nec[7] opulenter, Sall. J. 85, 34 Kritz.

2. cōlo, *āvi, ātum, āre, v. a. [colum], to filter, strain, to clarify, purify (post-Aug.): ceram, Col. 9, 16, 1: mel, id. 12, 11, 1: vinum sportā palmeā, Pall. Febr. 27: sucum linteo, Plin. 25, 13, 103, § 164: thymum cribro, Col. 7, 8, 7: aliquid per linteum, Scrib. Comp. 271: ad colum, Veg. 2, 28, 19: per colum, Apic. 4, 2: aurum, App. Flor. p. 343, 20: terra colans, Plin. 31, 3, 23, § 38: faex colata, id. 31, 8, 44, § 95.*
Poet.: *amnes inductis retibus, i. e. to spread out a fish-net, Manil. 5, 193.*

Depending on the level of granularity that is deemed adequate for the parser, the number of word senses associated with a lexeme will vary, and so will the number of lexical worlds that will help to discriminate between the various word senses. The entries for *colo* in Lewis and Short offer a nice playground for lexical world buiding : clearly two lexemes (*colo,colare* vs *colo,colere*). *Colo,colare* does not need lexical world building, since only one word sense is offered. For *colo,colere* the granularity level is open to discussion. How many word senses, which items to include in the lexical world associated with each word sense ? Should we distinguish between lexical worlds derived from definitions and lexical worlds derived from citations (when the definitions are provided in the language described, as is the case in Forcellini) ?

Should a lexical world be made out of word-forms or their corresponding lexemes ?
Which items should be regarded as stopwords ? Function words, very high frequency items, lexicographic place holders (*res* etc.), …
How do we measure the distance between the head of an argument filler in the sentence to be parsed and what we find as possible fillers in the dictionaries (the situation we have with *ripam* as filler of

7 According to modern editions the citation ought to run : *neque illos arte colam,* ***me*** *opulenter*

the object arg in our sentence *Liceat nobis vobisque utramque **ripam** colere* and at the same time occurring in the dictionary entry in the same syntactic position– as is the case in both Forcellini and Lewis and Short – is of course due to the fact that the text we are parsing is found in a work that is a primary source for dictionary citations). When we do not find the item occurring as such in the dictionary sources (that is, if we found neither *ripam* nor *ripa* (the corresponding lexeme) in our sources), we have to find a way of selecting the right lexical world for our argument, that is measure the distance between ripa and the fillers of the arg position in the dictionary sources ; if we are unable to determine the position filled by the members of the lexical world, we can still use lexical distance to determine the word sense to be selected, although the degree of accuracy is likely to be less.

Lexical distance can be measured with a sufficient degree of accuracy for English because of the quality and number of lexicographical resources that can be called into play (see Michiels 2016b). For Latin we will have to rely on coocurrence of items in *definitions* (if given in Latin) and *citations*, i.e. in lexical worlds. The process may look strange, but I think it is worth trying because lexicographical resources are much denser than other textual resources.

To make things clearer, what I'm claiming is that if we establish lexical worlds such as the one I listed above : *{domus, regio, urbs, terra, nemus, mons, silva, locus, Elysium, arvum, rus, **ripa**, Helicon, stagnum, flumen}* then the claim is that, for instance, *urbs* and *terra* are related because they cooccur in this particular lexical world. Their relatedness would be computed on the number of times they cooccur in the whole set of lexical worlds derived from the lexicographical ressources we exploit. If we build lemmatised lexical worlds, we gain in density but introduce a source of error due to wrong lemmatisation.

I leave such issues as food for thought.

Word sense assignment in ALP

Liceat nobis vobisque utramque ripam colere.

```
illocutionary_force:statement
vg
    selected_reading:licet_permission_is_given_to
    polarity:pos
    cat:vg
    * liceat *
    pos:v
    lex:licet
    voice:act
    tense:present
    mood:subjunctive
subject
    cat:pred
    mood:infinitive
    tense:present
    * nobis * * que * * uobis * * utramque * * ripam * * colere *
    case:or([nom,acc])
    polarity:pos
    c_str
        vg
            selected_reading:colo_inhabit
            polarity:pos
            cat:vg
            * colere *
            pos:v
            lex:colere
            voice:act
            tense:present
            mood:infinitive
        object
            * utramque * * ripam *
            index:i(p(5,6))
            cat:np
            sem:[thing,loc]
            number:sing
            person:3
            gender:fem
            lex:ripa
            case:acc
            c_str
                head
                    * ripam *
                    pos:noun
                    lex:ripa
                    case:acc
                    gender:fem
                    number:sing
                    sem:[thing,loc]
                adjp
                    cat:adjp
                    * utramque *
                    case:acc
                    number:sing
                    gender:fem
                    lex:uterque
                    c_str
                    uterque
        dative_of_interest
            * nobis * * que * * uobis *
            index:i([p(1,2)])
            sem:[hum]
            lex:pp1pl
            number:pl
            person:1
            gender:or([masc,fem])
            case:dat
            coord:yes
            c_str
                head
                pp1pl
                coord:que
                head
                pp2pl
```

Missi sunt cultum virtutem.

```
  illocutionary_force:statement
vg
    selected_reading:mitto_send
    polarity:pos
    cat:vg
    * missi * * sunt *
    lex:mittere
    person:3
    mood:indicative
    tense:perfect
    voice:pass
    number:pl
    gender:masc
subject
    source:context_retrievable
    number:pl
    person:3
    cat:np
    index:i(0,0)
    constraints_to_be_met:[case:nom]
    case:nom
cplt
    cat:pred
    mood:supine
    tense:present
    * cultum * * uirtutem *
    number:sing
    gender:neuter
    case:or([nom,acc])
    person:3
    c_str
        vg
            selected_reading:colo_practise
            polarity:pos
            cat:vg
            * cultum *
            pos:v
            lex:colere
            mood:supine
            person:3
        object
            * uirtutem *
            index:i(p(3,4))
            cat:np
            sem:[abstract,hum]
            number:sing
            person:3
            gender:fem
            lex:uirtus
            case:acc
            c_str
            uirtutem
```

In the case of *colo* the specifications so far (i.e in the present version of ALP)are in terms of **semantic features**, whose hierarchy is explored by the feature unification algorithm, so that hyponymy is taken into account (we are asking for a dog, we are given a poodle, OK ; we are asking for a poodle, we are given a dog, noway).

More on selectional restrictions

In a sentence such as *Ille sapientissimus vir Ithacam ut videret immortalitatem scribitur repudiasse,* we should realize that inserting commas around *Ithacam ut videret* amounts to guiding the parsing; as a matter of fact, it can be seen as a first, important step in parsing the sentence. But we should try to do without such commas, as they do not belong to the text but were introduced at various points in time by editors, sometimes in a way that reflects usage in one of our modern languages, such as the comma we find following *vereor* and preceding *ne* in the Budé edition of the *De Amicitia*, 14, which is more appropriate to German than to French, the language of Combès himself:

Quocirca maerere hoc eius eventu uereor, ne inuidi magis quam amici sit. (edited by Robert Combès, Paris, 1971)

Here too selectional restrictions are essential. Not only to discriminate between *repudiare* as *reject* and *repudiare* as *repudiate*, but to help the parser assign *ithacam* as object of *videret* and *immortalitatem* as object of *repudiasse*.
The two readings of *repudiare* place different selectional restrictions on the object:

```
% REPUDIARE
lexarg(repudiare,
       arglist:[ws(repudio1_reject,tr_cod,clause:[],mwuw:0,
                args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                      object:[type:np,oblig:yes,constraints:[case:acc,sem:[abstract] ]]])]).

lexarg(repudiare,
       arglist:[ws(repudio2_repudiate,tr_cod,clause:[],mwuw:0,
                args:[subject:[type:np,oblig:yes,constraints:[sem:[hum]]],
                      object:[type:np,oblig:yes,constraints:[case:acc,sem:[hum]]]])]).
```

Our test sentence, with no commas added:

% 77 Cicero De Legibus II 1 3.
Ille sapientissimus vir Ithacam ut videret immortalitatem scribitur repudiasse.

is assigned the following parse:

```
subordinator
    lex:ut
    pos:sub
    mood:subjunctive
    value:or([purpose,consequence])
subordinate_clause
    ut
        illocutionary_force:statement
        vg
            selected_reading:uideo_see
            polarity:pos
            cat:vg
            * uideret *
            pos:v
            lex:uidere
            voice:act
            tense:imperfect
            mood:subjunctive
            number:sing
            person:3
        subject
            source:context_retrievable
            number:sing
            person:3
            cat:np
            index:i(0,0)
            case:nom
        object
            * ithacam *
            index:i(p(3,4))
            cat:np
            sem:[loc]
            number:sing
            person:3
            gender:fem
            lex:ithaca
            case:acc
            c_str
            ithacam
main_clause
    vg
        polarity:pos
        cat:vg
        * scribitur *
        pos:v
        lex:scribere
        voice:pass
        tense:present
        mood:indicative
        number:sing
        person:3
    pred
        subject
            * ille * * sapientissimus * * uir *
            index:i(p(2,3))
            cat:np
            sem:[hum]
            number:sing
            person:3
            gender:masc
            lex:uir
            case:nom
            c_str
                head
                    * uir *
                    pos:noun
                    lex:uir
                    case:nom
                    gender:masc
                    number:sing
                    sem:[hum]
                adjp
                    cat:adjp
                    * ille * * sapientissimus *
                    case:nom
                    number:sing
                    gender:masc
                    lex:sapiens
                    c_str
                        ille
                        sapiens
        pred
            vg
                selected_reading:repudio1_reject
                polarity:pos
                cat:vg
                * repudiauisse *
                pos:v
                lex:repudiare
                voice:act
                tense:past
                mood:infinitive
            object
                * immortalitatem *
                index:i(p(6,7))
                cat:np
                sem:[abstract]
                number:sing
                person:3
                gender:fem
                lex:immortalitas
                case:acc
                c_str
                immortalitatem
```

Appendix IV Syncrétisme grammatical ou ambiguïté de construction ?

(Why is this written in French, I wonder ? ;-)

Considérons Tacite Hist. I 79 1 :
conuersis ad ciuile bellum animis externa sine cura habebantur

La ponctuation que nous allons introduire n'est pas innocente :

Conuersis ad ciuile bellum animis, externa sine cura habebantur.

Avec le marquage des limites de l'énoncé et sa division à l'aide de la virgule, l'interprétation de *conuersis ad ciuile bellum animis* comme ablatif absolu est fortement suggérée. C'est le texte proposé entre autres par Bornecque, et sa traduction (inspirée de celle de Burnouf) est conforme à cette interprétation qui scinde la phrase en deux :

Les esprits tournés à la guerre civile, les questions extérieures étaient négligées.

Le lien entre les deux parties de la phrase est laissé à l'inférence, procédé d'interprétation que l'on sait universel autant qu'automatique. Comme le dit très bien Lavency dans VSVS, l'ablatif absolu *décrit la situation concomitante au procès signifié dans la proposition principale.* C'est l'inférence qui donne à cette concomitance la possibilité de se teinter de la valeur sémantique et discursive qui complète son insertion dans l'énoncé.

Si l'on va voir chez Burnouf, on constate que sa traduction a bel et bien été remaniée par Bornecque. Burnouf traduisait en effet :

Les esprits tournés à la guerre civile ne songeaient plus aux dangers du dehors.

Ici l'énoncé est un seul bloc et le lien entre les deux parties n'est pas laissé à l'inférence mais matérialisé par le rapport sujet-verbe.

Goelzer va plus loin encore dans l'intégration des deux parties. Il interprète *externa* comme adjectif épithète, le *bella* qu'il modifie se déduisant sans peine du *bellum* de la première partie[8] :

Les esprits tournés à la guerre civile n'avaient plus souci des guerres étrangères ;

Notons bien le point virgule en clôture d'énoncé. Ce qui suit est en effet la relation d'une attaque sarmate en Mésie, un acte de guerre :

conuersis ad ciuile bellum animis externa sine cura habebantur eo audentius rhoxolani sarmatica gens priore hieme caesis duabus cohortibus magna spe moesiam inruperant ad nouem milia equitum ex ferocia et successu praedae magis quam pugnae intenta

Si le *conuersis ad ciuile bellum animis* n'est pas un ablatif absolu, comment faut-il l'analyser ? On

8 ALP n'a pas la sophistication nécessaire pour prendre en considération l'analyse de *externa* comme adjectif modifiant un *bella* sous-entendu. Il fait d'*externis* l'ablatif d'un adjectif substantivé neutre pluriel, *externa*, que l'on rencontre chez Tacite au même titre que l'adjectif *externus* non substantivé. Il me semble préférable de considérer *externa* comme substantivé avec le sens de 'ce qui concerne l'extérieur', les 'affaires extérieures'. En effet la guerre n'est pas négligée, la réponse à l'attaque des Sarmates est décisive et musclée, mais c'est le climat d'indifférence envers ce qui n'est pas Rome ou l'Italie qui a offert l'occasion de la guerre aux Sarmates. Je propose : *Les esprits préoccupés par la guerre civile n'avaient cure des affaires extérieures.*

peut en faire un complément d'agent de *sine cura habebantur*, *animis* ne nécessitant pas de préposition pour jouer ce rôle.

Tacite ne se proposait pas d'accompagner son texte d'arbres syntaxiques qui nous indiqueraient la lecture qu'il avait en tête au moment de l'écriture. On connaît son goût pour l'ablatif absolu. Il peut très bien avoir commencé en ablatif absolu et continué en complément d'agent, se rendant coupable (ou enrichissant son texte) de syncrétisme grammatical.

Il est intéressant de constater que ALP retient les deux analyses, les jugeant donc conformes à la grammaire sur laquelle se base son algorithme de parsage. Conformément au système de pondération qui favorise les arguments au détriment des satellites (tels les circonstanciels et les ablatifs absolus), l'analyse de *conuersis animis* comme complément d'agent est préférée à l'analyse qui en fait un ablatif absolu.

```
[0/conuersis,1/ad,2/ciuile,3/bellum,4/animis,5/externa,6/sine,7/cura,8/habebantur,endpos(9)]

7.47--->
   illocutionary_force:statement
    vg
        selected_reading:habeo_have
        polarity:pos
        cat:vg
        * habebantur *
        pos:v
        lex:habere
        voice:pass
        tense:imperfect
        mood:indicative
        number:pl
        person:3
    subject
        number:pl
        gender:neuter
        * externa *
        index:i(p(5,6))
        cat:np
        sem:[abstract,thing]
        person:3
        lex:externus
        case:nom
        c_str
        externa
    agent
        * conuersis * * ad * * ciuile * * bellum * * animis *
        index:i(p(4,5))
        cat:np
        sem:[hum,thing,abstract]
        number:pl
        person:3
        gender:fem
        lex:anima
        case:abl
        c_str
            head
                * animis *
                index:i(p(4,5))
                cat:np
                sem:[hum,thing,abstract]
                number:pl
                person:3
                gender:fem
                lex:anima
                case:abl
                c_str
                animis
            participle_clause
                * conuersis * * ad * * ciuile * * bellum *
                cat:ppclause
                number:pl
                gender:fem
                lex:conuerto_turn
                case:abl
                c_str
                    past_participle
                    conuerto_turn
                        prep_cplt
                            * ad * * ciuile * * bellum *
                            index:i(p(3,4))
                            case:acc
                            prep:ad
                            sem:[thing]
                            lex:bellum
                            cat:pp
                            c_str
                                prep:ad
                                head
                                    * ciuile * * bellum *
                                    index:i(p(3,4))
                                    cat:np
                                    sem:[thing]
                                    number:sing
                                    person:3
                                    gender:neuter
                                    lex:bellum
                                    case:acc
                                    c_str
                                        head
                                            * bellum *
                                            pos:noun
                                            lex:bellum
                                            case:acc
                                            gender:neuter
                                            number:sing
                                            sem:[thing]
                                        adjp
```

```
                                        cat:adjp
                                        * ciuile *
                                        case:acc
                                        number:sing
                                        gender:neuter
                                        lex:ciuilis
                                        c_str
                                        ciuilis
prep_phrase_adjunct_2
    * sine * * cura *
    index:i(p(7,8))
    case:abl
    prep:sine
    sem:[abstract]
    lex:cura
    cat:pp
    c_str
        prep:sine
        head
            * cura *
            index:i(p(7,8))
            cat:np
            sem:[abstract]
            number:sing
            person:3
            gender:fem
            lex:cura
            case:abl
            c_str
            cura
```

0.5--->
illocutionary_force:statement
vg
selected_reading:habeo_have
polarity:pos
cat:vg
* habebantur *
pos:v
lex:habere
voice:pass
tense:imperfect
mood:indicative
number:pl
person:3
subject
number:pl
gender:neuter
* externa *
index:i(p(5,6))
cat:np
sem:[abstract,thing]
person:3
lex:externus
case:nom
c_str
externa
prep_phrase_adjunct_2
* sine * * cura *
index:i(p(7,8))
case:abl
prep:sine
sem:[abstract]
lex:cura
cat:pp
c_str
prep:sine
head
* cura *
index:i(p(7,8))
cat:np
sem:[abstract]
number:sing
person:3
gender:fem
lex:cura
case:abl
c_str
cura
ablative_absolute
*** conuersis * * ad * * ciuile * * bellum * * animis ***
c_str
lex:conuerto_turn
pos:p_p
case:abl
gender:fem
number:pl
lex:conuertere
mood:participle
person:3
prep_cplt
* ad * * ciuile * * bellum *
index:i(p(3,4))
case:acc
prep:ad
sem:[thing]
lex:bellum
cat:pp
c_str
prep:ad
head
* ciuile * * bellum *
index:i(p(3,4))
cat:np
sem:[thing]
number:sing
person:3
gender:neuter
lex:bellum
case:acc
c_str
head
* bellum *
pos:noun
lex:bellum
case:acc
gender:neuter
number:sing
sem:[thing]
adjp
cat:adjp
* ciuile *

```
                        case:acc
                        number:sing
                        gender:neuter
                        lex:ciuilis
                        c_str
                        ciuilis
object
    * animis *
    index:i(p(4,5))
    cat:np
    sem:[hum,thing,abstract]
    number:pl
    person:3
    gender:fem
    lex:anima
    case:abl
    c_str
    animis
```

Considérons un second exemple, où l'interprétation de la chaîne en tant qu'ablatif absolu distend les liens de manière plus nette encore, bien que l'inférence s'avère assez puissante pour resserrer les liens que l'analyse grammaticale a relâchés. L'exemple est ici aussi tiré des Histoires de Tacite,

Tacite Hist. III 41
missis ad uitellium litteris auxilium postulat

Ici aussi Bornecque scinde l'énoncé en deux, et lit la première partie comme ablatif absolu :

Missis ad Vitellium litteris, auxilium postulat.

Et ici aussi il traduit (ou plutôt reprend textuellement la traduction de Burnouf) par un énoncé en deux parties, jointes par la conjonction de coordination *et* :

Il écrit à Vitellius et lui demande du secours.

L'inférence fait en sorte que le lecteur ne s'imagine pas que l'écriture de la lettre et la demande de secours sont deux actions différentes, la coordination étant à interpréter comme marquant la succession chronologique (une inférence tout à fait puissante en d'autres contextes).

Cela donne bien sûr une traduction assez relâchée, semblable à celle de Goelzer, qui néanmoins propose un texte sans virgule :

Missis ad Vitellium litteris auxilium postulat.
Il envoie un message à Vitellius et lui réclame du renfort.

Si on se libère de la lecture de *missis litteris* comme ablatif absolu, on aura moins de scrupule à propose une traduction plus resserrée :

Il écrit à Vitellius pour lui réclamer du renfort.
Par lettre il réclame à Vitellius des renforts.

C'est ce type de traduction qu'on rencontre en anglais :

He wrote to Vitellius asking for aid.
(Complete Works of Tacitus. Tr. Alfred John Church and William Jackson Brodribb 1873)

Ici, l'analyse alternative fait de *missis litteris* un ablatif de moyen. Le *missis* peut apparaître alors comme assez superflu, peu conforme à la recherche de la brièveté qui caractérise notre auteur. Encore un cas de syncrétisme ? ALP donne les deux analyses, auxquelles il confère le même poids, puisqu'elles font toutes deux de *missis litteris* un satellite et non un argument (circonstanciel de

moyen ou ablatif absolu).

```
[0/missis,1/ad,2/uitellium,3/litteris,4/auxilium,5/postulat,endpos(6)]

2--->
    illocutionary_force:statement
    vg
        selected_reading:postulo_request
        polarity:pos
        cat:vg
        * postulat *
        pos:v
        lex:postulare
        voice:act
        tense:present
        mood:indicative
        number:sing
        person:3
    subject
        source:context_retrievable
        number:sing
        person:3
        cat:np
        index:i(0,0)
        constraints_to_be_met:[sem:[hum]]
        case:nom
    object
        * auxilium *
        index:i(p(4,5))
        cat:np
        sem:[thing,abstract]
        number:sing
        person:3
        gender:neuter
        lex:auxilium
        case:acc
        c_str
        auxilium
    adjunct
        * missis * * ad * * uitellium * * litteris *
        value:manner_means
        c_str
            head
                * litteris *
                index:i(p(3,4))
                cat:np
                sem:[thing,abstract]
                number:pl
                person:3
                gender:fem
                lex:litterae
                case:abl
                c_str
                litteris
            participle_clause
                * missis * * ad * * uitellium *
                cat:ppclause
                number:pl
                gender:fem
                lex:mitto_send
                case:abl
                c_str
                    past_participle
                    mitto_send
                        prep_cplt
                            * ad * * uitellium *
                            index:i(p(2,3))
                            case:acc
                            prep:ad
                            sem:[male]
                            lex:uitellius
                            cat:pp
                            c_str
                                prep:ad
                                head
                                    * uitellium *
                                    index:i(p(2,3))
                                    cat:np
                                    sem:[male]
                                    lex:uitellius
                                    number:sing
                                    person:3
                                    gender:masc
                                    case:acc
                                    c_str
                                    uitellius
```

```
2--->
    illocutionary_force:statement
    vg
        selected_reading:postulo_request
        polarity:pos
        cat:vg
        * postulat *
        pos:v
        lex:postulare
        voice:act
        tense:present
        mood:indicative
        number:sing
        person:3
    subject
        source:context_retrievable
        number:sing
        person:3
        cat:np
        index:i(0,0)
        constraints_to_be_met:[sem:[hum]]
        case:nom
    object
        * auxilium *
        index:i(p(4,5))
        cat:np
        sem:[thing,abstract]
        number:sing
        person:3
        gender:neuter
        lex:auxilium
        case:acc
        c_str
        auxilium
    ablative_absolute
        * missis * * ad * * uitellium * * litteris *
        c_str
            lex:mitto_send
                pos:p_p
                case:abl
                gender:fem
                number:pl
                lex:mittere
                mood:participiple
                person:3
                prep_cplt
                    * ad * * uitellium *
                    index:i(p(2,3))
                    case:acc
                    prep:ad
                    sem:[male]
                    lex:uitellius
                    cat:pp
                    c_str
                        prep:ad
                        head
                            * uitellium *
                            index:i(p(2,3))
                            cat:np
                            sem:[male]
                            lex:uitellius
                            number:sing
                            person:3
                            gender:masc
                            case:acc
                            c_str
                            uitellius
            object
                * litteris *
                index:i(p(3,4))
                cat:np
                sem:[thing,abstract]
                number:pl
                person:3
                gender:fem
                lex:litterae
                case:abl
                c_str
                litteris
```

Il est à noter que l'on trouve dans les *Histoires* des exemples clairs de *litteris* comme ablatif de moyen :

Flaccus omisit inceptum aliisque litteris Gallum monuit ne terreret abeuntis 4,19
profectus eadem nocte Flauianus obuiis Vespasiani litteris discrimini exemptus est 3,10

On trouve de même *missis ... litteris* comme ablatif absolu :
missis sane ad eum Primi Antonii litteris 4,13 (le *ad eum* est déterminant dans l'analyse comme ablatif absolu).

La voie est ainsi ouverte à une analyse de nos énoncés qui penche d'un côté ou de l'autre, ou se maintient au beau milieu. Nous ne pensons pas que cette dernière analyse, ouverte au syncrétisme syntaxique, soit à rejeter d'emblée. Un parseur tel que ALP a le mérite de nous faire voir où une grammaire donnée nous conduit.

Il convient de dire clairement que ce syncrétisme est un produit de l'analyse, et non un donné linguistique. Il n'existe que des textes et des interprétations, les deux intimement liés, puisque l'interprétation se fait sur la base des textes, et les textes s'établissent conformément à des interprétations, qui à leur tour se matérialisent par le biais de traductions, lesquelles sont à leur tour soumises à une procédure d'interprétation : on ne peut s'échapper de la prison de la langue.

Lavency 2002, aux prises avec le phénomène que nous examinons ici, parle de 'doubles analyses' et 'd'incertitudes d'analyse'. C'est le point de vue analytique. L'option syncrétique est celle de l'interprétation.

Notons en passant qu'un des exemples donnés par Lavency pour illustrer cette incertitude est assez curieux :

Id Helvetii ratibus ac lintribus iunctis transibant. (Caesar, B.G. I, 12).

Lavency écrit :

L'expression *flumen ratibus ac lintribus iunctis transire* (cfr Caes.,G.II,12,1) est lue comme Ablatif de modalité (*hoc modo*) dans la mesure où *iunctis*, jugé omissible, est interprété comme participe épithète. Lu comme élément nécessaire, on considère l'ensemble comme Ablatif absolu.

Il semblerait toutefois que *iunctis* ne peut être associé qu'à *lintribus* (il s'agit d'un pont de bateaux), et non pas à l'ensemble *ratibus ac lintribus*[9]. L'omissibilité de *iunctis* est dès lors de nature purement grammaticale et non sémantique. Certes, pour qu'il y ait ablatif absolu, le *iunctis* est nécessaire. Mais cet ablatif absolu serait coordonné à un ablatif complément de moyen, ce qui est extrêmement peu probable sous la plume de César. Je crois que ce qui a joué ici c'est l'association quasi automatique chez l'analyste du couple **GN+pp à l'ablatif** avec la possibilité de l'attribution de la construction syntaxique **ablatif absolu**.

Or, dès lors qu'un participe passé entre en jeu, il est susceptible d'avoir sous sa dépendance tout le système actanciel de son prédicat, quel que soit le rôle joué localement par le participe. Dans le second exemple d'incertitude analytique impliquant un ablatif absolu proposé par Lavency, à savoir

9 Les traductions sont conformes : Les Helvétiens la passaient sur des radeaux et sur des barques jointes ensemble. (Artaud)
Les Helvètes étaient en train de la franchir à l'aide de radeaux et de barques assemblées. (Constans)
Les Helvètes la traversaient sur des radeaux et sur un **pont de bateaux.** (Maissiat)

Cuius aduentu spe inlata militibus ac redintegrato animo cum pro se quisque in conspectu imperatoris etiam extremis suis rebus operam nauare cuperet paulum hostium impetus tardatus est. (Caesar, B.G. II, 25, 3),

à propos duquel Lavency écrit :

Lors de l'épisode critique de la bataille contre les Nerviens, César s'avance en première ligne : *cuius aduentu spe inlata militibus… paulum hostium impetus tardatus est* (Caes., *G.* II,25,3). L'arrivée de César peut être tenue comme le moment du rétablissement du moral (paradigme *hoc tempore*) ; un admirateur de César peut choisir une autre lecture (paradigme *ea re* + passif) et y voir la source du sursaut romain. Avec ses deux constituants jugés solidaires, *spe inlata* est sans doute spontanément interprété comme Ablatif absolu et comme désignant le cadre dans lequel se relâche la pression ennemie.

je crois que le même automatisme d'analyse (GN ablatif + pp -> Abl. Absolu) s'est déclenché. Une analyse où les liens grammaticaux sont plus étroits (un complément d'agent de *tardatus est*, à deux volets, *spe inlata* et *redintegrato animo*, les participes *inlata* et *redintegrato* ayant leur propre agent, *cuius aduentu*) me semble plus plausible. Mais dans l'optique syncrétique que je tente de développer ici, la lecture de *spe inlata* et *redintegrato animo* (*inlata* et *redintegrato* se partageant l'agent *cuius aduentu*) comme ablatif absolu coordonné et explicité par la subordonnée en *cum* (qui en développe la conséquence), est *concurrente,* au sens étymologique du mot.

Appendix V Ordre des mots

(Again, why French? Again, why not?)

Principe général : la force des liens qui unissent les éléments tend à les rapprocher dans l'énoncé.

Des éléments liés à des degrés divers peuvent se trouver plus ou moins éloignés les uns des autres. Toute une panoplie de facteurs entre en jeu : expressivité, respect de la métrique, recherche de l'originalité ou au contraire *imitatio*, etc. Et il est à constater que plus le lien est étroit entre deux éléments, plus on peut les maintenir éloignés l'un de l'autre pour les raisons que nous venons d'évoquer. C'est la contrepartie (et en fait confirmation) du principe général énoncé ci-dessus.

Cet éloignement est toutefois contrôlé (maintenu dans certaines limites) par l'exigence de *computabilité*, la nécessité pour les liens existants de rester *récupérables*.

Cette computabilité dépend elle aussi de très nombreux facteurs. Si le lien est très étroit (par exemple participation à une même *unité phraséologique*), il résistera plus aisément à la distension créée par la distance, comme nous l'indiquions ci-dessus. L'étroitesse des liens dépend également de la participation à des *structures actantielles* mais aussi à des ensembles *sémantiques* ou *pragmatiques*. Les ressources lexicographiques s'avèrent essentielles pour l'établissement de tels ensembles.

Un autre facteur très important est la qualité du *marquage* des relations. Des éléments qui ne sont guère susceptibles d'offrir un éventail d'analyses morphologiques (par exemple deux génitifs pluriels) seront plus facilement repérables comme éléments d'une même structure. Un marquage sera d'autant plus puissant qu'il distingue un élément de son environnement, qu'il le fait 'ressortir'.

De même que la grammaticalité se mesure par le degré de conformité d'un énoncé à une grammaire (elle-même dérivée, essentiellement par analyse distributive, d'un corpus d'énoncés présumés 'standard', c'est-à-dire non déviants), la computabilité s'apprécie par un indice de plausibilité attribué à une analyse élaborée sur base de la grammaticalité de l'énoncé et des distensions dont les liens postulés par l'analyse grammaticale ont pu faire l'objet.

On conçoit ainsi qu'on envisage de rendre compte de l'ordre des mots par un calcul (aspect computationnel de la computabilité) de l'équilibre entre puissance des liens, distance, et qualité du marquage. Les analyses suggérées par un module grammatical couplé à un lexique seront pondérées en fonction des résultats de ce calcul. C'est la démarche que ALP tente de mettre en œuvre.

References

Dictionaries

LASLA Frequency Dictionary = Dictionnaire fréquentiel et Index inverse de la langue latine, L. Delatte ,Et. Evrard, S. Govaerts, J. Denooz, L.A.S.L.A.,1981

Forcellini = Forcellini et al. Lexicon totius latinitatis

Lewis and Short = Lewis, C.T. and Short,C. A Latin Dictionary, Oxford, 1879

Grammars

Michel = Jacques Michel, Grammaire de base du latin, 2nd edition, De Sikkel, 1962

VSVS = Marius Lavency, Usus: grammaire latine: description du latin classique en vue de la lecture des auteurs, Duculot, 1985

Ernout, A. and Thomas, F., Syntaxe latine, 2nd edition, Klincksieck, 1964

Other References

Covington 2003 : Michael A. Covington, *A Free-Word-Order Dependency Parser in Prolog*, Artificial Intelligence Center,The University of Georgia, 2003

Gal et al. 1991 : Gal, A., Lapalme, G., Saint-Dizier, P. & Somers, H., Prolog for Natural Language Processing, Wiley, Chichester, 1991

Koster 2005 : C.H.A. Koster, *Constructing a Parser for Latin*, in A. Gelbukh (ed.): CICLing 2005, LNCS 3406, pp. 48–59, Springer-Verlag, Berlin Heidelberg, 2005

Lavency 2002 : Marius Lavency, *Pour une taxinomie des syntagmes à l'Ablatif en latin classique*, Folia Electronica Classica, 4, Louvain-la-Neuve, 2002

Michiels 2016a : Archibald Michiels, *VERBA, a Multi-word-unit-oriented Feature-unification-based Parser*, unpublished paper, University of Liège, 2016

https://github.com/archibaldmichiels/alp/tree/main/verba/doc/verba.pdf

Michiels 2016b : Archibald Michiels, *LEXDIS, a Tool for Measuring Lexical Distance*, unpublished paper, University of Liège, 2016

https://github.com/archibaldmichiels/alp/blob/main/lexdis/doc/lexdis.pdf

Wielemaker 2003 : Jan Wielemaker, *An Overview of the SWI-Prolog Programming Environment*, in Mesnard, F. and Serebenik, A., eds, Proceedings of the 13th International Workshop on Logic Programming Environments, Katholieke Universiteit Leuven, Heverlee, Belgium, 2003