ALP – A Latin Parser

Archibald Michiels
amichiels@uliege.be

Autumn 2020

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.

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 2016).

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 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

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.

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_Noun):-
( (adjective(First,Agreement_Adj),noun(Second,Agreement_Noun)) ;
  (adjective(Second,Agreement_Adj),noun(First,Agreement_Noun)) ),
Agreement_Adj=Agreement_Noun.
```

A few words of explanation are in order:

pair(First, Second,Agreement_Noun):-

% 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_Noun.*
% 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_Adj),noun(Second,Agreement_Noun)) ;
(adjective(Second,Agreement_Adj),noun(First,Agreement_Noun))),

% 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)

Agreement_Adj = Agreement_Noun.

% and the two agreement triplets are UNIFIABLE (here unification boils down to identity – see
% below for a fuller treatment).
% Identity which also explains why *Agreement_Adj* could have taken the place of *Agreement_Noun*
% at the top of the rule, yielding *pair(First, Second,Agreement_Adj):-*.

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)]).

It's time to give an example of feature unification from the real world, i.e. in our case parsing.

We need a *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 third 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]]]

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), 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*[2] – its very monotony (it does one single thing: 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.

2 Although the mean parsing time for the test sentences is approximately three seconds per sentence, see the Appendix for a very time-consuming parse... 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...

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). 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 be 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 correspond 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 : 3.1875000000000004

  vg
    selected_reading:quaero_ask
    polarity:pos
    cat:vg
    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
    illocutionary_force:question
    number:sing
    person:2
    mood:subjunctive
    tense:present
    polarity:pos
    argbound:no
    add:no
    flagint:wh_question
    c_str
      vg
        selected_reading:servo_keep_safe
        polarity:pos
        cat:vg
        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
        index:i(p(3,4))
        cat:np
        sem:[thing]
        number:sing
        person:3
        gender:masc
        lex:piscis
        case:acc
      clause_level_adjunct
        cat:advp
        value:place
        lex:ubi
        c_str
          lex:ubi
          sem:location
          cat:advp
  clause_level_adjunct
    cat:np
    value:time
    number:sing
    person:3
    gender:neuter
    lex:tempus
    case:abl
    c_str
      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 piscem quaeris ubi tempore serves .
aestivo piscem serves quaeris tempore ubi .
aestivo piscem serves quaeris ubi tempore .
aestivo piscem serves tempore quaeris ubi .
aestivo piscem serves tempore ubi quaeris .
aestivo piscem serves ubi quaeris tempore .
aestivo piscem serves ubi tempore quaeris .
aestivo piscem tempore quaeris serves ubi .
aestivo piscem tempore quaeris ubi serves .
aestivo piscem tempore serves quaeris ubi .
aestivo piscem tempore serves ubi quaeris .
aestivo piscem tempore ubi quaeris serves .
aestivo piscem tempore ubi serves quaeris .
aestivo piscem ubi quaeris serves tempore .
aestivo piscem ubi quaeris tempore serves .
aestivo piscem ubi serves quaeris tempore .
aestivo piscem ubi serves tempore quaeris .
aestivo piscem ubi tempore quaeris serves .
aestivo piscem ubi tempore serves quaeris .
aestivo quaeris piscem serves tempore ubi .
aestivo quaeris piscem serves ubi tempore .
aestivo quaeris piscem tempore serves ubi .
aestivo quaeris piscem tempore ubi serves .
aestivo quaeris piscem ubi serves tempore .
aestivo quaeris piscem ubi tempore serves .
aestivo quaeris serves piscem tempore ubi .
aestivo quaeris serves piscem ubi tempore .
aestivo quaeris serves tempore piscem ubi .
aestivo quaeris serves tempore ubi piscem .
aestivo quaeris serves ubi piscem tempore .
aestivo quaeris serves ubi tempore piscem .
aestivo quaeris tempore piscem serves ubi .
aestivo quaeris tempore piscem ubi serves .
aestivo quaeris tempore serves piscem ubi .
aestivo quaeris tempore serves ubi piscem .
aestivo quaeris tempore ubi piscem serves .
aestivo quaeris tempore ubi serves piscem .
aestivo quaeris ubi piscem serves tempore .
aestivo quaeris ubi piscem tempore serves .
aestivo quaeris ubi serves piscem tempore .
aestivo quaeris ubi serves tempore piscem .
aestivo quaeris ubi tempore piscem serves .
aestivo quaeris ubi tempore serves piscem .
aestivo serves piscem quaeris tempore ubi .
aestivo serves piscem quaeris ubi tempore .
aestivo serves piscem tempore quaeris ubi .
aestivo serves piscem tempore ubi quaeris .
aestivo serves piscem ubi quaeris tempore .
aestivo serves piscem ubi tempore quaeris .
aestivo serves quaeris piscem tempore ubi .
aestivo serves quaeris piscem ubi tempore .
aestivo serves quaeris tempore piscem ubi .
aestivo serves quaeris tempore ubi piscem .
aestivo serves quaeris ubi piscem tempore .
aestivo serves quaeris ubi tempore piscem .
aestivo serves tempore piscem quaeris ubi .
aestivo serves tempore piscem ubi quaeris .
aestivo serves tempore quaeris piscem ubi .
aestivo serves tempore quaeris ubi piscem .
aestivo serves tempore ubi piscem quaeris .
aestivo serves tempore ubi quaeris piscem .
aestivo serves ubi piscem quaeris tempore .
aestivo serves ubi piscem tempore quaeris .
aestivo serves ubi quaeris piscem tempore .
aestivo serves ubi quaeris tempore piscem .
aestivo serves ubi tempore piscem quaeris .
aestivo serves ubi tempore quaeris piscem .
aestivo tempore piscem quaeris serves ubi .
aestivo tempore piscem quaeris ubi serves .
aestivo tempore piscem serves quaeris ubi .
aestivo tempore piscem serves ubi quaeris .
aestivo tempore piscem ubi quaeris serves .
aestivo tempore piscem ubi serves quaeris .
aestivo tempore quaeris piscem serves ubi .
aestivo tempore quaeris piscem ubi serves .
aestivo tempore quaeris serves piscem ubi .
aestivo tempore quaeris serves ubi piscem .
aestivo tempore quaeris ubi piscem serves .
aestivo tempore quaeris ubi serves piscem .
aestivo tempore serves piscem quaeris ubi .
aestivo tempore serves piscem ubi quaeris .
aestivo tempore serves quaeris piscem ubi .
aestivo tempore serves quaeris ubi piscem .
aestivo tempore serves ubi piscem quaeris .
aestivo tempore serves ubi quaeris piscem .
aestivo tempore ubi piscem quaeris serves .
aestivo tempore ubi piscem serves quaeris .
aestivo tempore ubi quaeris piscem serves .
aestivo tempore ubi quaeris serves piscem .
aestivo tempore ubi serves piscem quaeris .
aestivo tempore ubi serves quaeris piscem .
aestivo ubi piscem quaeris serves tempore .
aestivo ubi piscem quaeris tempore serves .
aestivo ubi piscem serves quaeris tempore .
aestivo ubi piscem serves tempore quaeris .
aestivo ubi piscem tempore quaeris serves .
aestivo ubi piscem tempore serves quaeris .
aestivo ubi quaeris piscem serves tempore .
aestivo ubi quaeris piscem tempore serves .
aestivo ubi quaeris serves piscem tempore .

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

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

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

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

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

ubi piscem quaeris aestivo serves tempore .
ubi piscem quaeris aestivo tempore serves .
ubi piscem quaeris serves aestivo tempore .
ubi piscem quaeris serves tempore aestivo .
ubi piscem quaeris tempore aestivo serves .
ubi piscem quaeris tempore serves aestivo .
ubi piscem serves aestivo quaeris tempore .
ubi piscem serves aestivo tempore quaeris .
ubi piscem serves quaeris aestivo tempore .
ubi piscem serves quaeris tempore aestivo .
ubi piscem serves tempore aestivo quaeris .
ubi piscem serves tempore quaeris aestivo .
ubi piscem tempore aestivo quaeris serves .
ubi piscem tempore aestivo serves quaeris .
ubi piscem tempore quaeris aestivo serves .
ubi piscem tempore quaeris serves aestivo .
ubi piscem tempore serves aestivo quaeris .
ubi piscem tempore serves quaeris aestivo .
ubi quaeris aestivo piscem serves tempore .
ubi quaeris aestivo piscem tempore serves .
ubi quaeris aestivo serves piscem tempore .
ubi quaeris aestivo serves tempore piscem .
ubi quaeris aestivo tempore piscem serves .
ubi quaeris aestivo tempore serves piscem .
ubi quaeris piscem aestivo serves tempore .
ubi quaeris piscem aestivo tempore serves .
ubi quaeris piscem serves aestivo tempore .
ubi quaeris piscem serves tempore aestivo .
ubi quaeris piscem tempore aestivo serves .
ubi quaeris piscem tempore serves aestivo .
ubi quaeris serves aestivo piscem tempore .
ubi quaeris serves aestivo tempore piscem .
ubi quaeris serves piscem aestivo tempore .
ubi quaeris serves piscem tempore aestivo .
ubi quaeris serves tempore aestivo piscem .
ubi quaeris serves tempore piscem aestivo .
ubi quaeris tempore aestivo piscem serves .
ubi quaeris tempore aestivo serves piscem .
ubi quaeris tempore piscem aestivo serves .
ubi quaeris tempore piscem serves aestivo .
ubi quaeris tempore serves aestivo piscem .
ubi quaeris tempore serves piscem aestivo .
ubi serves aestivo piscem quaeris tempore .
ubi serves aestivo piscem tempore quaeris .
ubi serves aestivo quaeris piscem tempore .
ubi serves aestivo quaeris tempore piscem .
ubi serves aestivo tempore piscem quaeris .
ubi serves aestivo tempore quaeris piscem .
ubi serves piscem aestivo quaeris tempore .
ubi serves piscem aestivo tempore quaeris .
ubi serves piscem quaeris aestivo tempore .
ubi serves piscem quaeris tempore aestivo .
ubi serves piscem tempore aestivo quaeris .
ubi serves piscem tempore quaeris aestivo .
ubi serves quaeris aestivo piscem tempore .
ubi serves quaeris aestivo tempore piscem .
ubi serves quaeris piscem aestivo tempore .
ubi serves quaeris piscem tempore aestivo .
ubi serves quaeris tempore aestivo piscem .
ubi serves quaeris tempore piscem aestivo .
ubi serves tempore aestivo piscem quaeris .
ubi serves tempore aestivo quaeris piscem .
ubi serves tempore piscem aestivo quaeris .
ubi serves tempore piscem quaeris aestivo .
ubi serves tempore quaeris aestivo piscem .
ubi serves tempore quaeris piscem aestivo .
ubi tempore aestivo piscem quaeris serves .
ubi tempore aestivo piscem serves quaeris .
ubi tempore aestivo quaeris piscem serves .
ubi tempore aestivo quaeris serves piscem .
ubi tempore aestivo serves piscem quaeris .
ubi tempore aestivo serves quaeris piscem .
ubi tempore piscem aestivo quaeris serves .
ubi tempore piscem aestivo serves quaeris .
ubi tempore piscem quaeris aestivo serves .
ubi tempore piscem quaeris serves aestivo .
ubi tempore piscem serves aestivo quaeris .
ubi tempore piscem serves quaeris aestivo .
ubi tempore quaeris aestivo piscem serves .
ubi tempore quaeris aestivo serves piscem .
ubi tempore quaeris piscem aestivo serves .
ubi tempore quaeris piscem serves aestivo .
ubi tempore quaeris serves aestivo piscem .
ubi tempore quaeris serves piscem aestivo .
ubi tempore serves aestivo piscem quaeris .
ubi tempore serves aestivo quaeris piscem .
ubi tempore serves piscem aestivo quaeris .
ubi tempore serves piscem quaeris aestivo .
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 Appendix.

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, flagint:or([yes_no_question,wh_question])]]])]).
```

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 embody either a yes/no question, or a wh-question. 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 for 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 *record/erase* pair. No confusion is possible.

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 need 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)]
cputime : 0.39

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

cputime : 3.36

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.

Test Files

1. Basic Test File

% 1.
Insanis.
% one-word sentence with context-retrievable subject (here the second-person personal pronoun)

% 2.
Insaniuerunt praetores imperatoresque.
% treatment of conjunction - dealing with enclitic *-que*

% 3.
Habent sua fata libelli.
% Credits to Terentianus Maurus - word order

% 4.
Nil pensi neque sancti habebant milites.
% Multi-word unit (MWU) with frozen internal make-up

% 5.
Nauta rationes puellae pulchrae in dubium uocat.
% Multi-word unit (MWU) with open slot for object argument -
% straining factor computed on genitive phrase
% the straining factor is based on the distance between head and dependent genitive phrase
% so that *puellae pulchrae* gets attached to *rationes* rather than *nauta* in the preferred parse

% 6.
Marcus dixit regi magno salutem longam.
% MWU with open structure (*salutem dicere* with open slot for indirect object)

% 7.
Ancilla dat puero pulchro nuces pecuniamque magnam.
% Conjunction

% 8.
Mali servi legerunt reginae epistulas.
% Straining factor
% 'The bad slaves read the queen's letters' rather than
% 'The queens read the letters of the bad slave'

% 9 and 10.
Mala ancilla misit reginae epistulas.
Mala ancilla misit regi reginae epistulas.
% Argument saturation (*mittere* with two args, one direct, one indirect, object)
% preferred to single argument
% 'The bad servant sent letters to the queen' rather than
% 'The bad servant sent the queen's letters
% but in the second sentence the dative regi fills in the erg slot for the indirect object
% and the genitive reginae must be attached to epistulas
% 'The bad servant sent the queen's letters to the king'

% 11 and 12.
Marco donat ciuitas immortalitatem.
Marcum donat ciuitas immortalitate.
% alternation in argument structure

% 13.
Dedimus profecto *grande documentum patientiae.*
% unknown words (here *profecto*) skipped - credits to Tacitus
% the skipping of unknown words is NOT a plus point
% but makes life easier...

% 14.
Dicunt militibus malis aqua et igni praetorem interdixisse.
% MWU and infinitive clause as argument

% 15.
Vincere scis.
% Credits to Livy - infinitive as arg (as opposed to 'scis te vicisse'
% with full clause)

% 16.
Dixit rex reginam librum pulchrum misisse Marco.
% infinitive clause - semantic guidance on subject
% the subject must be +HUM and the meaning is therefore
% 'The king said that the queen sent a beautiful book to Marcus'
% rather than The king said that a/the beautiful book sent the queen to Marcus
% although the latter does make sense if metaphorical discourse is at a premium

% 17.
Ancilla dicit reginam patulae sub tegmine fagi recubauisse.
% Credits to Vergilius - straining in the genitive phase
% but *patulae* has no other attachment point than *fagi*

% 18.
Putabas nautam puellae pulchrae pecuniam magnam dedisse.
% preference for arg saturation – *puellae pulchrae* with preferred reading as indirect object,
% and not genitive phrase to be attached to either *nautam* or *pecuniam*

% 19.
Regina putat regem epistulas longas ad ancillam misisse.
% arg alternation with *mitto*, semantic constraint

% 20.
Rex sciebat se epistulam reginae legisse.
% *Se* as subject in subordinate indexed to subject in main clause

% 21.
Praetor non amabat milites nec faciebat pili cohortem.
% credits to Catullus - MWU - polarity clause constraint
% since the MWU is inherently negative, i.e. must be inserted in a negative context

% 22.
Praetor cui scio Marcum libellum impudicum dedisse amat servas.
% relative clause with relevant gap indexation
% *Cui* must be indexed to *praetor*, the antecedent,
% which must be assigned as subject to *amat*

% 23.
Rex quem putas reginam amare amat ancillam reginae.
% double role of accusative in infinitive clause
% object of *putas* and subject of *amare*

% 24.
Putabam ancillam pueros grammaticam docere.
% double acc with *doceo* - semantic control
% this is discussed in an appendix to this document

% 25, 26 and 27.
Reges quos vult perdere dementat. *% a*
Quos vult perdere dementat. *% b*
Quos vult perdere. *% c*

% relative with antecedent % a
% relative with antecedent included in relative % b
% 'relatif de liaison' (*et eos*) % c

% 28 and 29.
Caesar mittit legiones legato ad urbem capiendam.
Caesar mittit legiones legato urbis capiendae causa.
% Gerundivum / Gerundium
% recognition of the predicate-object link (*capere urbem* in both constructions)

% 30, 31, 32 and 33.
Mihi est opus patientia Marci.
Mihi sunt opera patientiae.
Caesari erat urbs capienda.
Est militum bonorum capere urbes.
% Various SUM-constructions with their own semantic make-up

% 34 and 35.
Urbe ab imperatore capta misit Caesar epistulam ad legatum. *% a*
Rege sub tegmine fagi recubante scribit regina epistulas ad servum Marci. *% b*
% Ablative absolutes of restricted types:
% object arg + pp, % a
% subject arg + present participle % b

% 36.
Elige cui dicas: tu mihi sola places.
% credits to Ovid - finite clause as arg of *dico*.

% 37, 38 and 39.
Si rex amasset servas, scripsisset libellos impudicos.
Rex, si amasset servas, scripsisset libellos impudicos.
Rex scripsisset libellos impudicos, si amauisset servas.
% subordinate clauses at various insertion points
% with working out of the semantic import of the main clause-subordinate clause nexus

% 40.
Me tabula sacer votiva paries indicat uvida suspendisse potenti vestimenta maris deo.
% credits to Horatius - notice that the whole thing does not sport a single toolword;
% for it to be parsed, though, we need to drop the *contiguity requirement* on the non-finite
% clause – the *Me* as first word of the line introduces discontinuity : *me ... uvida suspendisse* etc.
% The position we adopt is to insist on contiguity when parsing prose, which is a shame, but a
% blessing if efficiency is valued...

% Some more (what they are supposed to cover should be obvious):

% 41.
Eo Romam.
% 42.
Eo auxilium rogatum.
% 43.
Itum est in templum.
% 44.
Sequuntur caedes.
% 45.
Tacebant omnes senatores.
% 46.
Timeo ne veniant.
% 47.
Vixerunt.
% 48.
Vixit vitam longam beatamque.
% 49.
Romani cum Germanis pugnavere.
% 50.
Rem age.
% 51.
Rex dona ab hostibus accepit.
% 52.
Karthago delenda est.
% 53.
Imperator legiones ad proelium duxit.
% 54.
Timeo Danaos etiam dona ferentes.
% 55.
De re refertur.
% 56.
Caesari erat eundum Romam ad senatores hortandos.
% 57.
Laudamus te.
% 58.
Gaia vult libellos impudicos quos serva Marci scripsit legere.
% 59.
Memento documenti patientiae nostrae.
% 60.
% credits to Blaise Pascal
Non obliviscar sermones tuos.
% 61.
Consulatum magna cum cura petit.
% 62.
Thessaliam ex negotio petebam.
% 63.
Italiam peto videndae urbis in montibus positae.
% 64.
Rex uitam beatam in natura quaerebat.
% 65.

Me te secutum fuisse crediderunt.
% 66.
Amicis meis timeo.
% 67.
Timeo ne ad urbem capiendam veniant.
% 68.
Cicerone magistro usi sunt multi magistri.
% 69.
Parsimonia est ars uitandi sumptus supervacuos.
% 70.
Matrem eius vocavit et non venit.
% 71.
Bonum vinum faciamus.
% 72.
Libellos Marci habet rex impudicos.
% 73.
Libellos impudicos habet regina documenta ingenii humani.
% 74.
Catilina nihil pensi neque sancti habere dicitur.
% 75.
Crede hoc mihi.
% 76.
Ciceronem oratorem optimum credunt.
% 77.
Ciceronem oratorem optimum esse credunt.
% 78.
Cicero epistulas optimas scripsisse existimabatur.
% 79.
Iussit omnes tacere.
% 80.
Iudico te optimum praetorem esse.
% 81.
Negat se libellos impudicos scribere.
% 82.
Nuntiatum est amicos nostros vinum amare.
% 83.
Omnes hostes rogaturos esse auxilium ratus est.
% 84.
Caesar se Germanos vicisse sciebat.
% 85.
Victoria uti nescis.
% 86.
Cicero litteras longas scribere traditur.
% 87.
Ne hostis vincat vereor.
% 88.
Vereor Italiam petere.
% 89.
Legere possunt.
% 90.
Epistulas tuas legere uolo.
% 91.

Nolite Lugdunum ire.
% 92.
Rex Italiam petere mavult.
% 93.
Fit ut omnes me libellos impudicos legere sciant.
% 94.
Fiat lux.
Facta est lux.
% 95.
Pecunia magna documentum avaritiae est.
% 96.
Malum est insanire.
% 97.
Obliuisci in nostra potestate est.
% 98.
Eundum Romam erat Caesari.
% 99. Caesar, everybody in Belgium knows where...
Omnium fortissimi sunt Belgae.
% 100.
Marcus fortior est Catilina.
% 101 Cicero, De Amicitia.
% included because it made me stumble in my reading the De Amicitia
Suis autem incommodis graviter angi non amicum sed se ipsum amantis est.
% 102 Caesar, De Bello Gallico.
% Multi-word unit *res novae.*
Cupiditate regni adductus nouis rebus studebat.
% 103.
% Non-restrictive relative clause attached to place adjunct, long string, long parse.
Lego librum tuum in horto, ubi amica mea longas litteras ad ancillam tuam scribit.
% 104.
% Coordination
Caesar cognoverat virtutem legati sui et sciebat eum Germanos esse victurum.
% 105.
% Binding of SE
Romani putant reginam credere se a Venere amatum iri.
% 106 Martial, 2, 78.
% Word order
Aestivo serves ubi piscem tempore quaeris?
% 107 Terentius.
Humani nihil a me alienum puto.
% 108 Adapted from Tacitus, Agricola, II.
Memoriam ipsam perdidissemus, si tam facile esset oblivisci quam tacere.

2. From *VSVS*: Standard Examples Used in the Teaching of Latin in French Schools

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?

A Few Example Parses

Omnes hostes rogaturos esse auxilium ratus est.

[0/omnes,1/hostes,2/rogaturos,3/esse,4/auxilium,5/ratus,6/est,endpos(7)]
cputime: 12.7

```
4->
   vg
      selected_reading:reor_think_that
      polarity:pos
      cat:vg
      lex:reri
      person:3
      mood:indicative
      tense:perfect
      voice:act
      number:sing
      gender:masc
   subject
      source:context_retrievable
      number:sing
      person:3
      constraints_to_be_met:[sem:[hum]]
      case:nom
   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:rogo_ask_for
            polarity:pos
            cat:vg
            lex:rogare
            mood:infinitive
            tense:future
            voice:active
            number:pl
            gender:masc
         subject
            index:i(p(1,2))
            cat:np
            sem:[hum]
            number:pl
            person:3
            gender:masc
            lex:hostis
            case:acc
            c_str
               head
                  pos:noun
                  lex:hostis
                  case:acc
                  gender:masc
                  number:pl
                  sem:[hum]
               adjp
                  cat:adjp
                  case:or([nom,acc])
                  number:pl
                  gender:or([masc,fem])
                  lex:omnis
         object
            index:i(p(4,5))
            cat:np
            sem:[thing,abstract]
            number:sing
            person:3
            gender:neuter
            lex:auxilium
            case:acc
```

Caesar se Germanos vicisse sciebat.

[0/caesar,1/se,2/germanos,3/uicisse,4/sciebat,endpos(5)]
cputime: 0.90

4->

```
vg
  selected_reading:scio_know_that
  polarity:pos
  cat:vg
  pos:v
  lex:scire
  voice:act
  tense:imperfect
  mood:indicative
  number:sing
  person:3
subject
  index:i(p(0,1))
  cat:np
  sem:[hum]
  lex:caesar
  number:sing
  person:3
  gender:masc
  case:nom
object
  cat:pred
  mood:infinitive
  tense:past
  number:sing
  gender:neuter
  case:or([nom,acc])
  person:3
  polarity:pos
  argbound:no
  flagint:no
  c_str
    vg
      selected_reading:uinco_win
      polarity:pos
      cat:vg
      gender:or([masc,fem])
      pos:v
      lex:uincere
      voice:act
      tense:past
      mood:infinitive
    subject
      index:i(p(0,1))
      cat:np
      sem:[hum]
      lex:pp3refl
      number:or([sing,pl])
      person:3
      gender:or([masc,fem])
      case:acc
    object
      index:i(p(2,3))
      cat:np
      sem:[hum]
      number:pl
      person:3
      gender:masc
      lex:germanus
      case:acc
```

Angebant ingentis spiritus virum Sicilia Sardiniaque amissae.

[0/angebant,1/ingentis,2/spiritus,3/uirum,4/sicilia,5/que,6/sardinia,7/amissae,endpos(8)]
cputime: 1.7

3->

```
vg
  selected_reading:ango_torment
  polarity:pos
  cat:vg
  pos:v
  lex:angere
  voice:act
  tense:imperfect
  mood:indicative
  number:pl
  person:3
subject
  index:i([p(4,5),p(6,7)])
  sem:[loc,thing]
  lex:sicilia
  number:pl
  person:3
  gender:fem
  case:nom
  coord:yes
  c_str
    np
      head
      sicilia
      coord:[lex:que,pos:coord]
      head
      sardinia
    adjp
      cat:adjp
      case:nom
      number:pl
      gender:fem
      lex:amittere
      morph:ppt
object
  index:i([p(3,4)])
  sem:[hum]
  number:sing
  person:3
  cat:np
  lex:uir
  gender:masc
  case:acc
  c_str
    head
    uir
    noun_cplt
      index:i(p(2,3))
      cat:np
      sem:[abstract]
      number:sing
      person:3
      gender:masc
      lex:spiritus
      case:gen
      c_str
        head
          pos:noun
          lex:spiritus
          case:gen
          gender:masc
          number:sing
          sem:[abstract]
        adjp
          cat:adjp
          case:gen
          number:sing
          gender:masc
          lex:ingens
```

Litterae quas scripsisti mihi iuncundissimae fuerunt.

[0/litterae,1/quas,2/scripsisti,3/mihi,4/iucundissimae,5/fuerunt,endpos(6)]
cputime: 4.05

5->

```
vg
    selected_reading:sum_be
    polarity:pos
    cat:vg
    pos:v
    lex:esse
    voice:act
    tense:perfect
    mood:indicative
    number:pl
    person:3
subject
    cat:np
    index:i(p(0,1))
    number:pl
    gender:fem
    sem:[thing,abstract]
    person:3
    case:nom
    lex:litterae
    c_str
        head
        litterae
        rel_clause
            index:i(p(0,1))
            number:pl
            gender:fem
            case:acc
            mood:indicative
            tense:perfect
            c_str
                vg
                    selected_reading:scribo_write
                    polarity:pos
                    cat:vg
                    pos:v
                    lex:scribere
                    voice:act
                    tense:perfect
                    mood:indicative
                    number:sing
                    person:2
                subject
                    source:context_retrievable
                    number:sing
                    gender:or([masc,fem])
                    person:2
                    constraints_to_be_met:[sem:[hum]]
                    case:nom
                object
                    e:i(p(0,1))
                i_object
                    index:i(p(3,4))
                    cat:np
                    sem:[hum]
                    lex:pp1sg
                    number:sing
                    person:1
                    gender:or([masc,fem])
                    case:dat
predicative
    cat:adjp
    case:nom
    number:pl
    gender:fem
    lex:iucundus
```

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 ALP tackles coordination … during a quick ... 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 est Augustinus etiam in ecclesia tua concupiscere fructus mortis et agere negotium procurandi eos.

[0/ausus,1/est,2/augustinus,3/etiam,4/in,5/ecclesia,6/tua,7/concupiscere,8/fructus,9/mortis,10/et,11/agere,12/negotium,13/procurandi,14/eos,endpos(15)]

cputime : 97.70312500000001 seconds **i.e. nearly two minutes !**

```
vg
  selected_reading:audeo_dare
  polarity:pos
  cat:vg
  lex:audere
  person:3
  mood:indicative
  tense:perfect
  voice:act
  number:sing
  gender:masc
subject
  number:sing
  gender:masc
  index:i(p(2,3))
  cat:np
  sem:[hum]
  lex:augustinus
  person:3
  case:nom
object
  cat:pred
  mood:infinitive
  tense:present
  number:sing
  gender:neuter
  case:or([nom,acc])
  person:3
  c_str
    head
      vg
        selected_reading:concupisco_desire
        polarity:pos
        cat:vg
        pos:v
        lex:concupiscere
        voice:act
        tense:present
        mood:infinitive
      object
        index:i([p(8,9)])
        sem:[thing,abstract]
        number:pl
        person:3
        cat:np
        lex:fructus
        gender:masc
        case:acc
        c_str
          head
            index:i(p(8,9))
            cat:np
            sem:[thing,abstract]
            number:pl
            person:3
            gender:masc
            lex:fructus
            case:acc
          noun_cplt
            index:i(p(9,10))
            cat:np
            sem:[abstract,thing,hum]
            number:sing
            person:3
            gender:fem
            lex:mors
            case:gen
      clause_level_adjunct
        cat:advp
        lex:etiam
        sem:discourse
      clause_level_adjunct
        index:i(p(5,6))
        case:abl
        prep:in
```

```
        sem:[hum,thing,loc]
        lex:ecclesia
        cat:pp
        c_str
          prep:in
          head
            index:i(p(5,6))
            cat:np
            sem:[hum,thing,loc]
            number:sing
            person:3
            gender:fem
            lex:ecclesia
            case:abl
            c_str
              head
                pos:noun
                lex:ecclesia
                case:abl
                gender:fem
                number:sing
                sem:[hum,thing,loc]
              adjp
                cat:adjp
                case:abl
                number:sing
                gender:fem
                lex:tuus
coord:[lex:et,pos:coord]
head
  vg
    selected_reading:negotium_ago_take_care_of
    polarity:pos
    cat:vg
    pos:v
    lex:agere
    voice:act
    tense:present
    mood:infinitive
  object
    index:i(p(12,13))
    cat:np
    sem:[abstract]
    number:sing
    person:3
    gender:neuter
    lex:negotium
    case:acc
  object_cplt
    cat:pred
    mood:gerund
    local_case:gen
    number:sing
    person:3
    gender:neuter
    c_str
      vg
        selected_reading:procuro_procure
        pos:v
        lex:procurare
        mood:gerund
        person:3
        case:gen
      object
        index:i(p(14,15))
        cat:np
        lex:prpersaccmascpl
        number:pl
        person:3
        gender:masc
        case:acc
```

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. Note also that without the two adjuncts (*etiam* and *in ecclesia*), a few seconds are all ALP needs to output a correct parse.

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 : 0.21875 sec.

Quies inter labores renovavit corpora animosque.
7 words
cputime : 0.984375

Quies inter labores renovavit corpora animosque ad omnia patienda.
10 words
cputime : 9.359375

Quies inter labores renovavit corpora animosque ad omnia de integro patienda.
12 words
cputime : 13.562500000000004

Quies inter labores exhaustos aut exhauriendos renovavit corpora animosque ad omnia de integro patienda.
15 words
cputime : 30.062500000000004

Per totum tempus hiemis quies inter labores exhaustos aut exhauriendos renovavit corpora animosque ad omnia de integro patienda.
19 words
cputime : 1455.9687500000002

The parses produced are correct, except for the attachment of *hiemis* to *quies* instead of *tempus* in the parse of the very last string submitted to analysis.

References

Dictionary

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

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

Koster 2005 = C.H.A. Koster, *Constructing a Parser for Latin*, in A. Gelbukh (Ed.): CICLing 2005, LNCS 3406, pp. 48–59, 2005 Springer-Verlag Berlin Heidelberg 2005

Michiels 2016 = Archibald Michiels, *VERBA, a Multi-word-unit-oriented Feature-unification-based Parser*, unpublished paper, University of Liège, 2016

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