HORATIO

A middle-sized NLP application in Prolog

Archibald Michiels

University of Liège, Liège

1994

Table of Contents

1. Foreword

There now exist quite a number of books on **Prolog** and even quite a few on **Prolog and NLP** (Natural Language Processing). Among the latter the following stand out: **Pereira and Shieber 1987**, **Gazdar and Mellish 1989**, **Gal et al. 1991, Covington 1993** and the chapter on natural language in **Walker et al. 1987** (by Michael McCord, whose approach is the one followed here; see also **McCord 1982**, as well as **McCord 1989a, 1989b** and **1990** for new developments).

However these books remain at an introductory level. More specifically, they discuss only short introductory programs. None of them describes and gives the full code of a medium-sized application.

This is precisely what this book does. **horatio** (and its generation counterpart **horgen**) is a middle-sized application in the field of NLP, more precisely natural language analysis[1] and generation. The full source code, with numerous comments, is to be found on the disk distributed with the book[2].

No pseudo-code is given. No piece of code is presented in a simplified form. Cited code always corresponds exactly to the runnable code to be found on the companion disk. This may lead to the reader feeling somewhat swamped under barely digestible code. However, if he persists in his study of **horatio**, allowing some areas to remain unexplained for a while, he will have mastered not only the general design decisions, but also the interactions between the various components of the grammar and the coding idioms and mechanisms in full detail. The hard step from elementary to advanced NLP applications in Prolog is due to the need for managing the interactions between various 'solutions', which are all beautifully simple when looked at in isolation but display the irritating tendency not to mesh with their neighbours as readily as one would wish.

horatio remains a 'toy' system in that it is oriented towards the teaching of Prolog for NLP rather than any real life application. It is geared to the parsing and generation of 'linguistic' rather than 'real' sentences (in the sense of **Tomita 1991**, i.e. sentences made up for the purpose of testing linguistic hypotheses rather than utterances occurring in actual text).

The emphasis throughout this book is on **grammar writing** rather than on the writing of **grammar interpreters** or **compilers** as extensions to standard Prolog. The grammars here are interpreted within the strict top down regime of Prolog; although they belong to the family of **definite clause grammars** (dcg's), the **dcg** notation is not made use of.

This position needs to be explained and argued for. Recently a lot of research has been geared towards extensions of Prolog for NLP, to be either interpreted or compiled into standard Prolog (cf. *inter alia* **Definite Clause Grammars** (Pereira and Warren 1980), **Extraposition Grammars** (Pereira 1981), **Gapping Grammars** (Dahl and Abramson 1984), **Dislog** (Saint-Dizier et al. 1990), **Restriction Grammar** (Hirschman and Dowding 1990), **CLE** (The **Core Language Engine**, Alshawi et al. 1992). These extensions aim at enhancing the following desirable properties for NLP systems:

clean separation between the grammar and the parsing algorithm;

expressive power of the grammar, leading to a compact grammar that directly embodies linguistic generalizations (for instance in the treatment of linguistic discontinuities such as the relation between a trace and its filler);

automatic construction of the parse tree.

The **dcg notation** is clearly a first step in this direction: the linguist need no longer be concerned with the **difference list** technique that takes care of progression in the input word list. The distinction between grammar predicates and standard Prolog predicates is emphasized notationally.

However, these extensions to Prolog are not 'free'. The price to pay is on the one hand some loss of control over the parsing process and on the other much harder debugging. The first point can be illustrated with reference to the **dcg** notation. Since the progression in the input list is added when the **dcg** grammar is interpreted or compiled into standard Prolog, it is no longer possible to specify no progression in the input list in the head of the clause. A body needs to be written. Compare:

```
gap(...,Input,Input).
```

with

```
gap(...) --> [].
```

This may seem a matter of no great import. However, when **horatio** is rewritten in **dcg** notation, a considerable loss of efficiency is incurred. Parsing time increases by about 50% on the standard test suite.

One of the lessons to be learnt from the **Eurotra** project is the importance of efficiency for testing and debugging purposes. If the system needs half an hour to provide a sentence with a parse, or to generate a surface form from some deeper representation, the grammar writer will be very wary of experimenting with alternative approaches. He will keep his test suite as short as possible. He will not run the full test suite after so-called 'minimal', 'housekeeping' changes to the grammar. He will invent tricks to attempt to improve efficiency, even if they are detrimental to the readability of the code. As a result, the grammar will be brittle, not properly debugged and even harder to debug than if efficiency had been a primary concern right from the start.

Debugging is known to be hard in Prolog, on account of nondeterminism and non-permanent variable instantiation. It gets much harder when what is debugged is not the grammar the linguist has written but its extended form, i.e. its translation into standard Prolog. Published documentation on the extensions to Prolog mentioned above does not discuss debugging facilities. We have preferred to restrict ourselves to standard Prolog: what is debugged is then the grammar as written by the linguist.

Efforts have been made to keep the code of **horatio** reasonably clean and maintainable: no use has been made of program-modifying predicates (`assert`, `retract`, `recorda`,...) and the grammar itself is devoid of cuts. The whole system relies on standard Prolog unification, which is a clean, monotonic process. Control of execution in the system is restricted to testing whether variables are instantiated at the point a call is made to a predicate.

Our parsing algorithm is **top-down**, **left-to-right** and **depth-first**. This is of course the parsing algorithm that Prolog itself uses, its 'native' parsing algorithm as it were. It is not necessarily the most or the least efficient: this depends on the nature of the grammar, and on the inputs it is meant to account for. Well-known **bottom-up** parsers (with **top down oracles** implemented by a **link** predicate) have been designed in Prolog, for instance the **left-corner** parser of Pereira and Shieber 1987 and **BUP** (Matsumoto et al. 1983).

The programs in this book are written in **Arity Prolog**[3]. Use is made of Arity Prolog extensions to standard Edinburgh Prolog, mainly in the area of input and output (cf. such predicates as **concat**, **create**, **open**, **read_line**, etc.). The syntax and semantics of these predicates will be briefly explained in the body of the text and in comments to the source code, as well as in a short appendix (see Appendix A, p. 116). The reader who wants a full specification is referred to the Arity Prolog documentation or to **Marcus 1986**.

It should be made clear that **horatio** is not "tied" to Arity Prolog. It is easily convertible to standard Edinburgh Prolog notation, and as a matter of fact **horatio** also exists in a **Yap** Prolog version running on Sun. Arity Prolog has been selected because it is both reasonably fast and available on PC platforms (DOS, WINDOWS and OS/2).

2. Introduction

horatio is a **parser** for a subset of English based on a **definite clause grammar** belonging to the **slot grammar** framework (cf. the work of Michael McCord and associates; cf. e.g. McCord 1987[4], a presentation of the framework in half-tutorial fashion). It is written in **ARITY Prolog** (Version 5.1 for DOS) and runs on a **386/486** PC under **DOS** or **OS/2** (for OS/2, version 6 of Arity Prolog has been used).

All parses shown in this book were produced on an **IBM Model 70**, with 4 Megabytes of core memory and a 120 Mega hard disk. The operating system is DOS 6.0.

On an Intel 486 DX2 PC clone (66 MHz) the standard **horatio** test suite (**horsuite**, the 156 sentences to be found in Appendix G, p. 164, from "*they failed*" to "*I decided what to tell her I believed her to like*") takes less than 9 minutes real time (**user** time, not **cpu**) to parse. The average parsing time for a sentence in the test suite is therefore about three seconds. Parsing here includes writing the raw (horsuite**.ter**) and pretty-printed (horsuite**.lst**) parses to disk files (**horsuite.lst:** 159.702 bytes). The generator **horgen** is considerably faster than the parser. Compiling and linking the grammar to make the executable file **horatio.exe** (executing **moratio.bat**, the DOS counterpart of **moratio.cmd** to be found on page 118) takes less than one minute under DOS 6.0. on the specified machine.

A first question that we need to tackle concerns the nature of parsing. Obviously the nature and depth of the parses produced is a crucial issue. Parsing goes from **tagging** (the association of form with grammatical tags reflecting Part of Speech (**POS**)) to deep analysis, looking for the semantic invariant behind different phrasings.

The level chosen here is the one that is deemed to be adequate for the translation from/into English into/from a related language, such as French. In terms of depth the type of parse produced is not very different from those in the **IS** (**Interface Structure**) in the EEC **Eurotra** project, with which the author was associated[5]. The backbone remains syntactic.

In order to give an idea of the type of parses produced by **horatio**, we shall look at the parse returned by the system for the following sentence: *The workshop is believed to have taken place in the library I wanted her to go to*.

It will be seen that the parse is uncontroversial. Any application that needs to rely on a **linguistic** analysis of the sentences it is confronted with (i.e. an application such as machine translation, for which **template matching** or **keyword search**, however refined, are not good enough) will **at least** have to be able to retrieve the information provided by the **horatio** parse. I tend to agree with McCord, who writes: "It also appears reasonable to use syntactic analysis (embodying some semantic choices, such as word sense disambiguation) in machine translation systems." (in **McCord 1987**, p. 325)

We shall look at the parse in its pretty-printed format. Indentation from the left margin reflects depth of embedding: the further we are from the left margin, the deeper we are in the postulated structure. Items at the same distance from the margin are supposed to display the same level of embedding.

Here is the parse (other parses can be found in appendix F, p. 155):

```
28
 clause
  pred_arg_mod_structure
  prop(vce: passive,asp: none,mod: none,tns: present)
   predicate(believe_1,agr(en_passive))
    object
     clause
      pred_arg_mod_structure
      prop(vce: active,asp: [perfect],mod: none,tns: present)
       predicate(take_place_1,agr(en_active))
        subject
         nounphrase
         index(_0508)
         agr(third,sing)
           det(the)
           noun(workshop_1,agr(sing))
        pp_arg
         prepphrase
         index(_09EC)
         prep(in)
          np_arg_of_prep
           nounphrase
           index(_09F4)
           agr(third,sing)
             det(the)
             noun(library_1,agr(sing))
            relative_clause
             clause
              pred_arg_mod_structure
              prop(vce: active,asp: none,mod: none,tns: past)
               predicate(want_1,agr(finite,past,sing,first))
                subject
                 nounphrase
                 index(_0C28)
                 agr(first,sing)
                 ppro(first,sing,_0CAC)
                object
                 clause
                  pred_arg_mod_structure
                  prop(vce: active,asp: none,mod: none,tns: present)
                   predicate(go_1,agr(infinitive))
                    subject
                     nounphrase
                     index(_0E10)
                     agr(third,sing)
                     ppro(third,sing,fem)
                    pp_arg
                     prepphrase
                     index(_1010)
                     prep(to)
                      np_arg_of_prep
        nounphrase
        index(_09F4)
        agr(third,_1074)
```

The first line of the returned parse is the **preference** (28). In the case of multiple parses, the one with the highest preference index is to be preferred. The mechanisms used in the computation of the preference index are discussed on page 92 and following.

The parse is best conceived of as a set of **clause parses** each headed by a clause **header** of the following form:

```
 clause
   pred_arg_mod_structure
```

This means that the parser has found a clause and that it is going to display its structure in terms of its **predicate**, the **arguments** pertaining to that predicate and the clause **modifiers**, if any (the latter are not tied to the lexically-determined argument structure opened up by the predicate).

We then have a line devoted to the properties of the clause: **voice** (*active/passive*), **aspect** (*none/perfect/progressive*), **modality** (*none/modal aux*), and **tense** (*present/past*). *Have taken place* yields the following **prop** line:

```
prop(vce: active,asp: [perfect],mod: none,tns: present)
```

The predicate has its own property line, made up of the **lexeme** (with **reading number**) and of an **agreement** structure. **Multi-word units** are recognized as such, even if their component parts are not adjacent to each other in the input string (see the relevant section on page 21). The predicate line for *wanted* is the following:

```
predicate(want_1,agr(finite,past,sing,first))
```

The values **sing** and **first** (person) are obviously not computed on the basis of *wanted*, but on the basis of the surface subject *I*.

We then get the list of **arguments**, in **canonical** order. Unspecified arguments (such as the subject of *believe*) are left out. The relationships between the four clauses as displayed by the parse are the following:

clause 1
    predicate: *believe*
        args: subject: unspecified
            object: clausal (clause 2)

clause 2
    predicate: *take_place*
        args: subject: *workshop*
            pp_arg: *in library* (index X)
                np modifier: rel clause (clause 3)

clause 3
    predicate: *want*
        args: subject: *I*
            object: clausal (clause 4)

clause 4
    predicate: *go*

```
args:  subject: she
          pp_arg: to library (index X)
```

Prepositional phrases and noun phrases bear an **index** that is used for **coindexing**. In the sample parse, the missing np governed by the preposition *to* is coindexed with the np *the library*: **(index(_09F4))**. Such coindexing is crucial for the treatment of **gapping** and **long distance dependencies** (see page 99).

Noun phrases also display an agreement structure. For *her* we find the following two lines:

```
agr(third,sing)
ppro(third,sing,fem)
```

They indicate that we have a personal pronoun whose **gender** is *feminine*, **number** *singular* and **person** *third*. The agreement structures are part of the information that the **horatio** parses keep about surface structure to make it possible for the generator **horgen** to retrieve the surface forms from the raw Prolog terms corresponding to the parses.

However, the adequacy of this type of parsing for translation purposes is not proven - the reader is given a program that parses and generates, not one that translates; besides, and on a more positive note, the structures arrived at are presumably usable for other purposes than translation from and into a related language.

We claim that the real touchstone in **horatio** is the **ability to disambiguate between the various readings** of the lexical items belonging to the string to be parsed. Such reading assignment can be seen as one of the central tasks of any parsing system geared towards high quality translation. But of course this is not a rigorous test, because there is no way to decide on the number of readings an item has - the granularity depends on the purposes that are set to the lexicon in the system, as it does on the size of the dictionary and the targeted audience in lexicographical practice.

In the last instance the best way for the reader to decide whether he is interested in what **horatio** can do is to look at the sample parses and the test suites provided, and then at the mechanisms and strategies put to work in the parser, to assess their degree of generality and reusability.

We make no claim as to the originality of the solutions provided by **horatio** to parsing problems (although we would be entitled to do so for the treatment of multi-word units and hard coordination). As has already been stressed, it is the interaction between the components that proves the most difficult to manage in applications that go beyond the 'toy' stage.

It should be noted that the generator **horgen** provides a minimal check on the parses produced by **horatio**. It guarantees that the parses produced keep enough information for generating back the strings the parser worked on. This minimal ability is crucial in machine translation, although it is of less interest for other endeavours, such as the development of a natural language front end to a data base.

3. The Lexicon

A main principle of **horatio** is that information which belongs **to** the lexicon should belong **in** the lexicon. A prime example is **frame** information, i.e. information on the syntactic (and/or semantic) environment a given item can or must fit into. The lexical entries themselves contain the relevant frames; they do not refer to information stored elsewhere. Consider the entries for ALLOW in **horatio:**

```
m_verb(verbtr,allow_1,allow,allow,allow,allows,allowing,
     allowed,allowed,allowed,trans,abstract,
        [np(oblig,posprec(1,Wnp),object,abstract)]).

/* the facts allow the explanation */

m_verb(vthat,allow_2,allow,allow,allow,allows,allowing,
        allowed,allowed,allowed,trans,human,
        [s(oblig,posprec(1,Precs),object)]).

/* she allows that he is good */

m_verb(vio,allow_3,allow,allow,allow,allows,allowing,
        allowed,allowed,allowed,trans,human,
        [np(oblig,posprec(2,Wnp1),object,thing),
         io(oblig,posprec(1,W2),indirect_object,human,_)]).

/* the teacher allows the boys money for books */

m_verb(vinf,allow_4,allow,allow,allow,allows,allowing,
        allowed,allowed,allowed,trans,_,
        [np(oblig,posprec(1,Wnp),surf_object,_),
         np_vp(oblig,to_inf,object)]).

/* they allowed him to teach linguistics */

m_verb(vobjadv,allow_5,allow,allow,allow,allows,allowing,
        allowed,allowed,allowed,trans,human,
        [np(oblig,posprec(1,Wnp),object,human),
         pp(oblig,posprec(1,Wpp),pp_arg,_,direction,_)]).

/* he allowed the girl into the library */

m_verb(vtrprep,allow_for_1,allow,allow,allow,allows,allowing,
        allowed,allowed,allowed,trans,human,
        [pp(oblig,posprec(1,Wpp),pp_arg,_,_,for)]).

/* he allowed for the oversimplifications */

m_verb(vtrprep,allow_for_1,allow,allow,allow,allows,allowing,
        allowed,allowed,allowed,trans,human,
        [string(oblig,posprec(1,0),[for]),
         np(oblig,posprec(2,Wnp),object,_)]).

/* he allowed for the oversimplifications */
```

(the existence of two `m_verb` clauses for the same reading of ALLOW is explained below, p. 19.)

The arguments appear in a list which is the last argument of the predicate `m_verb`, which acts as **macro-clause**. The first argument is the class the predicate belongs to, the second is the **lexeme** value - including **reading number** - , positions 3 to 10 take care of inflectional morphology, position 11 is the value for the transitivity feature, position 12 is a semantic restriction on the deep subject[6]. Each element of the argument list opens with the value for the optionality feature - either **oblig**(atory) or **opt**(ional). The **posprec** structure is discussed below, p. 91 - it is used to establish linear precedence. The nature of a given argument in the lexical predicate's argument list is of course given by the functor of the structure (such as **string**, **np**, **pp**, etc. in the entries for ALLOW). A common feature is that for surface or deep **gf** (grammatical function).

The advantage of putting lexical information in the lexicon is obvious: additions, changes or enhancements in the argument structure of lexical predicates (whether individual predicates or whole classes) do not entail changes in the grammar.

An alleged disadvantage is the size of the lexicon, which very soon grows rather bulky. However, this disadvantage is not a real one because lexical entries need not be produced as such by the linguist or lexicographer; they can result from the expansion of macro-clauses, either within or outside Prolog. Besides, lexical entries can be imported from a machine-readable dictionary (MRD), as in the importation from **ldoce** (The Longman Dictionary of Contemporary English) to **horatio**, discussed in Appendix E. The task of the linguist or lexicographer is then reduced to selecting retrieval criteria and checking and expanding the resulting entries. A string manipulation language such as AWK is an ideal tool for performing the necessary format transformations.

As for consultation, at least for languages such as English, it is not necessary to load the whole lexicon into Prolog. Selective downloading can easily be achieved by a simple AWK program, such as **getvoc.awk** (see Appendix D, page 123).

3.1. Use of Macros

In **horatio macro-expansion** is done in Prolog. Consider an **`m_verb`** clause and one of its **expansion clauses**:

Macro-clause

```
m_verb(verbtr,_,allow_1,allow,allow,allow,allows,allowing,
      allowed,allowed,allowed,trans,abstract,
      [np(oblig,posprec(1,Wnp),object,abstract)]).

/* the facts allow the explanation */
```

Expansion for third person singular present tense inflectional form

```
verb([Vs|X],X,Class,[predicate(Lex,agr(finite,
          present,sing,third))],
      finite,present,sing,thirdsg,Semsubj,Args):-
m_verb(Class,Part,Lex,_,_,_,Vs,_,_,_,_,_,Semsubj,Args).
```

The **`verb`** clause is responsible for progression in the input list. The word that is to be read in must be a third person singular present tense form. This form is read off the macro-clause, where it occupies a certain position in the predicate's argument list. The **agr** functor is filled in the **`verb`** clause and appears as part of the structure that is returned in the parse tree. Other relevant information is transferred from the macro-clause to its expansions by unification (such as for instance the argument list **Args**).

By having as many **`verb`** clauses as is warranted by the inflectional paradigm associated with English verbs we manage to account - in a fairly economical fashion - for all possible forms for the verbs our lexicon includes.

It is of course possible to generate the macro-clauses themselves, at least partially. In fact, it is even possible to import them from a computerized dictionary such as **LDOCE**, once it has been converted to data base format, as it has been at the University of Liège. The interested reader is referred to Appendix E for a full discussion, including the complete code of the **awk** programs that take care of the necessary reformatting operations.

3.2. Double Analysis

Quirk et al. 1985 and **Bresnan 1981** argue cogently that some English lexical constructions can be parsed in two ways. Such a double analysis is necessary to account for the syntactic manipulations that these constructions admit of.

A case in point for English is the **verb+preposition** combination, as in LOOK AT. We can conceive of LOOK AT as a transitive verb like any other, or we can conceive of it as the verb LOOK governing a prepositional phrase headed by AT. Schematically:

1) LOOK AT + NP
2) LOOK +PP (AT)

The following sentences illustrate two of the syntactic manipulations (WH-movement and passivization) that lead one to postulate the need for a double analysis. Others can be found in **Quirk et al. 1985** and **Bresnan 1981**.

1: What are you looking at ?
   The man he was looking at ...
   The problem has been looked at from every angle
2: The text at which we have been looking for too long ...

Pulman in **Alshawi et al. 1992** (p. 74) points out that if take advantage of is treated as a complex V only one passive can be derived in the GPSG meta-rule treatment of the passive, because **advantage** will not be available as an NP node for the meta-rule to apply to. Consequently, only the first of the following two passive S's will be generated:

Kim was taken advantage of.
Advantage was taken of Kim.

This leads Pulman to reject the GPSG treatment. But the problem disappears if a double analysis is provided, evidence for which is precisely the availability of two passives.

It should be noted that the need for double analysis of some lexical constructions is not limited to English. Consider AVOIR L'AIR in French. We need to assign the following two analyses:

1) AVOIR L'AIR + ADJ
2) AVOIR + NP (AIR + ADJ)

on account of the two ways in which agreement can be made (either with AIR or with the subject of the whole phrase AVOIR L'AIR):

Elle a l'air idiote.
Elle a l'air idiot.

In **horatio** the lexicon file **lexatio2.ari** holds two macro-clauses for prepositional verbs such as LOOK AT. The first caters for the analysis in which the preposition belongs to the prepositional phrase rather than to the verb (analysis 2 in our account). The **arglist** contains a prepositional phrase specified in terms of the preposition heading it (AT in the case of LOOK AT):

```
m_verb(vtrprep,_,look_at_1,look,look,look,looks,looking,
       looked,looked,looked,trans,living,
       [pp(oblig,posprec(1,Wpp),pp_arg,_,_,at)]).

/* they were looking at her
   the girl at whom they had been looking */
```

The second macro-clause identifies AT as a particle to be appended immediately to the right of the verb LOOK (second argument of the macro-clause). The **arglist** opens with a string (AT) and further contains the np playing the object role:

```
m_verb(vtrprep,part0:at,look_at_1,look,look,look,looks,looking,
       looked,looked,looked,trans,living,
       [string(oblig,posprec(1,0),[at]),
        np(oblig,posprec(2,Wnp),object,_)]).

/* they were looking at her
   whom are they looking at ? */
```

It should be noted that in both `m_verb` clauses the lexeme value is the same, viz. **look_at_1**. We are dealing with the same lexical item.

In the analysis of sentences such as *They were looking at her*, both `m_verb` clauses will succeed, and two parses will be returned. Such redundancy is not felt to be a negative feature, as the relationship between verb and preposition is truly indeterminate in such cases.

Parsing an S such as *the problem was paid attention to* relies in **horatio** on the availability of a lexical entry for pay attention to in which **attention to** is simply a string appended to **pay**, the only argument being the np inside the to-phrase, and therefore the only candidate for the subject role in the passive clause:

```
m_verb(vtrphrprep,part0:'attention to',
pay_attention_to_1_a,pay,pay,pay,pays,paying,
     paid,paid,paid,trans,human,
     [string(oblig,posprec(1,0),[attention,to]),
     np(oblig,posprec(2,Wnp),object,_)]).
```

On the other hand parsing the other passive (*attention was paid to the problem*) will take advantage of the np node whose head is the word **attention**:

```
m_verb(vobjfreepp,_,pay_attention_1,pay,pay,pay,pays,paying,
     paid,paid,paid,trans,human,
     [np(oblig,posprec(1,Wnp),object,attention),
     pp(oblig,posprec(1,Wpp),pp_arg,_,_,to)]).
```

Note that here we have a pp argument governed by preposition **to**, which is necessary to account for such relative clauses as *to which he had paid great attention*.

3.3. Verb Classes

In **horatio** the lexical clauses for verbs (`m_verb` clauses) have as first argument the class the verb belongs to. Such a piece of information is used as a handle, useful when we wish to have a quick and easy way of ascertaining that a given verb is appropriate for the operation we want to perform. For instance, it is very handy to be able to pick out quickly **raising** or **extraposition** verbs, or **copula** verbs.

However, we do not use the sharing of verb class as a necessary or sufficient condition for the two verbs to be coordinated in a verb phrase. We shall see that we cannot readily dispense with a double parsing of the remaining word list, once as **arglist** of the first verb, and once as **arglist** of the second (see the section on hard coordination, p. 94).

The most important part played by the verb class is to provide an entry point for **consistency checks** and **template determination** in a lexicographer's workbench, or in the importation process of lexical material from a machine-readable dictionary.

3.4. Raising and Control

The treatment of raising and control in **horatio** is based on the distinction between **surface** and **deep** grammatical functions. The main principle is that a subject will go on playing the subject role until a potential new subject is found.

I should point out that I disagree with McCord as to the potential subject status of the indirect object. He claims (cf. **McCord 1987**, p. 346) that the slot frame for PROMISE should have an **iobj** slot, not an **obj** slot, and that the indirect object (**iobj**) cannot play the part of new subject in complement verb phrases, whereas **obj** (direct objects) and **pobj** (prepositional objects) can. Therefore *Bill* is able to play the part of subject of the complement vp (*to see Mary* and *to find Mary*) in the first two of the following sentences (because *Bill* is **obj** in the first and **pobj** in the second), but not in the third, where *Bill* is **iobj:**

John wants Bill to see Mary.
John depended on Bill to find Mary.
John promised Bill to see Mary.

I hold PROMISE to be a real exception, to be marked as such in the lexicon. Compare the following sentences, which show that PROMISE and TEACH behave the same way, **syntactically**:

I promised him to teach linguistics.
I taught him to teach linguistics.
What did I promise him ?
What did I teach him ?
Who did I promise to teach linguistics ?
Who did I teach to teach linguistics ?
I promised it to him.
I taught it to him.

The distinction between the two arguments cannot be captured by syntactic tests. *Him (Who)* is **iobj** (indirect object) in both cases, but cannot fill the **new subject** slot (i.e. that of subject for any **vp** to the right) in the case of PROMISE, whereas it can (and does) in the case of TEACH and other verbs governing an indirect object.

3.5. Multi-Word Units

In **horatio multi-word units** (**mwus**) are dealt with according to the degree of morphological, syntactic and lexical frozenness that they exhibit.

Certain pieces of structure develop ties (a degree of internal cohesiveness) that go beyond what the grammar predicts, or have meanings (and often translations) that are non-compositional with respect to the grammar being used, or -as is often the case- display both these characteristics at one and the same time. We call them **mwus**.

Mwus illustrate the **non-givenness** of the lexicon. More than single word units, they are theoretical constructs. Their recognition -and the structure that they are assigned- should result from their behaviour in discourse, more precisely from their **potential for manipulation**. The main principle adhered to in **horatio** is that mwus should be assigned as little structure as their behaviour warrants. It is this amount of assigned structure which determines the appropriate techniques to be used for the recognition of mwus from their manifestation in discourse. To give just one example: in order to recognize the mwu **TAKE PLACE** we look for a morphological form of the verb **TAKE** immediately followed by the **string P-L-A-C-E**; we do not look for an object **NP** whose realization is the noun **PLACE**; we do not look for the noun **PLACE** either. Consequently, the entry for **TAKE PLACE** runs as follows:

```
m_verb(vidiomintr,take_place_1,take,take,take,takes,taking,
       took,took,taken,intrans,abstract,
       [string(oblig,posprec(1,0),[place]),
        pp(oblig,posprec(2,Wpp),pp_arg,_,location,_)]).

/* the workshop took place in the university */
```

Mwus also illustrate the **arbitrariness** of the grammar-lexis distinction. In **horatio** there is no linguistically motivated border between syntax and lexis. We can choose to say that **unit clauses** (a Prolog concept) make up the dictionary of the system, but then the term **dictionary** is no longer used in a sense that is relevant to linguistic theory.

In order to assess the degree of internal cohesion of mwus we explore three classes of manipulation:

3.5.1. Insertion

Insertion of material into the lexical unit; compare:

PLAY A ROLE ---> PLAY AN important ROLE

SET FIRE TO ---> * SET dangerous FIRE TO

This type of insertion (insertion of modifiers attached to elements belonging to a piece of the mwu) should be distinguished from:

a) **interruption of the mwu by foreign material:**

he PAID, if I may say so, ATTENTION TO the problem
* the match TOOK, if I may say so, PLACE in the library

b) **insertion into the mwu of one or several of its arguments**:

he TOOK the problems INTO ACCOUNT (insertion of the object '*the problems*' into the mwu **TAKE INTO ACCOUNT**)

3.5.2. Extraction

Extraction of an element from its position within the canonical representation of the lexical unit; this basic manipulation subsumes all standard transformations effecting **movement** or **deletion**; compare:

PAY ATTENTION TO ---> *attention was paid to every single detail*

MAKE A FOOL OF **---> * *a fool was made of the new head***

3.5.3. Proformation

Replacement of a node in the **mwu** by a suitable **pro-form**: personal or indefinite pronoun for **NP**, **so** for **S**, **do so** for (certain classes of) **VP**, **there** for **PPs** functioning as place adjunct, etc. Compare:

PLAY A ROLE **--->** *play it again*

PAY ATTENTION TO **---> *** *pay some again* **/ *** *don't pay any to him*

In **horatio** we distinguish (in a hierarchy from **frozen** to **open**):

a) **completely frozen mwus**

A standard example is the adverb **BY AND LARGE**.

These mwus have no internal structure. They should be regarded as objects of type **string**, with their various elements bound by the **adjacency operator** (i.e. white space). In particular, there is no reason whatsoever for trying to assign a **part of speech** to any of the constitutive elements: for example, **BY** is not a preposition here (or whatever else for that matter: it is no more than the sequence of letters **B-Y**) and **LARGE** is not an adjective.

b) **mwus that allow only inflectional morphology variation** (in one or several of their constituents)

Examples in **horatio** are **TAKE PLACE** and **SHOOT THE BREEZE**, in which **TAKE** and **SHOOT** can be inflected. Only the complete configurations are assigned structures. There is no reason to assign any structure to **PLACE**, which is simply the sequence of letters **P-L-A-C-E**. It does not behave as an **NP**, so is not an **NP**. It does not behave as a noun, so is not a noun. We have already given the entry for TAKE PLACE. Here is the one for SHOOT THE BREEZE:

```
m_verb(vidiomintr,part0:'the breeze',
          shoot_the_breeze_1,shoot,shoot,shoot,shoots,shooting,
      shot,shot,shot,intrans,human,
      [string(oblig,posprec(1,0),[the,breeze])]).
```

c) **mwus which can be interrupted by one or several of their arguments**.

An example in **horatio** is **TAKE INTO ACCOUNT**. **TAKE** can be inflected. **TAKE** and **INTO ACCOUNT** can be separated by the object of the **mwu**:

he took the problems into account
he took into account the problems that she had seen

(the relevant feature for position of the object is its **weight**) Here is the entry for TAKE INTO ACCOUNT:

```
m_verb(vobjfixedpp,part1:'into account',
       take_into_account_1,take,take,take,takes,taking,
      took,took,taken,trans,human,
      [string(oblig,posprec(1,3),[into,account]),
       np(oblig,posprec(1,Wnp),object,_)]).
```

d) **collocations**: these are mwus whose elements are free to behave as the normal (i.e. with respect to a given grammar) structure assignment predicts. An example in **horatio** is **TAKE MEASURE**, where both

TAKE and the NP whose head is the noun **MEASURE** behave as predicted by the 'normal' structure assignment:

VP [V [TAKE] NP [... Head N [MEASURE]]]]

The link between **TAKE** and **MEASURE** is **collocational**, i.e. **TAKE** is the preferred verb to express what it expresses here. The implementation of such a lexical affinity in **horatio** is achieved through a feature on the object of **TAKE**, namely **[measure]**, feature which is assigned to the noun **MEASURE** under one of its readings. Such features can be regarded as hyperspecialised semantic features, i.e. it is hypothesized that they will not be needed alongside semantic features, and that consequently they can share the same slot. The entry for TAKE MEASURE looks like this:

```
m_verb(verbtr,_,take_measure_1,take,take,take,takes,taking,
      took,took,taken,trans,human,
      [np(oblig,posprec(1,Wnp),object,measure)]).
```

horatio also has the corresponding entry for the noun MEASURE when used in the TAKE MEASURE collocation:

```
m_noun(measure_1,measure,measures,[measure],[]).
```

In connection with the implementation of mwu's it should be noted that when we `satisfy` (i.e. match against the input word list) a fixed string, we return no parse tree, as the fixed string is included in the predicate's lexical entry (as in **look_down_on_1)** as well as in the predicate's arglist:

```
satisfy(P0,P1,[],0,Posprec,Rel,Intrel,[],
    string(Type,Posprec,String),_,_):-
append(String, P1, P0).
```

The **String** appended to the remaining list should yield the input list. In the lexical entry, **String** is a list as in:

```
[string(oblig,posprec(1,0),[down]),
```

part of the entry for look down on:

```
m_verb(vtrphrprep,part0:down,look_down_on_1,look,look,look,looks,looking,
    looked,looked,looked,trans,human,
    [string(oblig,posprec(1,0),[down]),
    pp(oblig,posprec(2,Wpp),pp_arg,_,_,on)]).

/* the teacher looked down on his students */
```

3.6. Inflectional Morphology

horatio works with a full form dictionary (of course, morphological variants can be looked up -irregular forms- or generated -regular forms- cf. Appendix E); verbs have nine morphological variants:

am	first person sing present tense
is	third person sing present tense
was	first person singular past tense
was	third person sing past tense
were	second person or plural past tense
are	second person or plural present tense
being	ING form
been	past participle
be	infinitive

Note: even BE does not have nine, but eight, distinct forms; the assignment of nine forms is grammar-internal.

4. The Grammar: Interpreting the Lexicon

horatio is a lexicon-driven parser. The main task of the grammar component is to interpret the information contained in the lexical predicates of the system.

What does interpreting the lexicon actually mean? Consider the **arglist** for the second reading of ALLOW, repeated here for convenience:

```
[s(oblig,posprec(1,Precs),object)]
```

This argument is interpreted by a clause for the predicate `satisfy`[7], which recursively calls the grammar for the parsing of a sentence (job of the `xsentence` predicate):

```
satisfy(P0,P1,Gap,Prefgen,posprec(Pos,4),
          Rel,Intrel,[Function,Tree],
        s(Type,posprec(Pos,4),Function),
        subject(SUBJ,Semsubj),
        subject(SUBJ,Semsubj)):-

       xsentence(P0,P1,Gap,Pref,Tree,finite,Person,Number,Voice),
       Prefgen is Pref + 4.
```

The same process is used for other argument types; for instance, to satisfy an **np** arg, a call on the `nounphrase` predicate is made; information can be read off the lexicon when necessary, as in this case information on the function to be filled by the **np**. The argument list is traversed non-deterministically. The checking of linear precedence is discussed below (see p.91).

4.1. General Strategy

Arguments and Modifiers

Arguments are **lexically** determined. They are either **obligatory** or **optional**. In **Horatio**, each lexical **argbearer** (argument bearer) has an **arglist** (argument list) associated with it in the lexicon (the **arglist** is always the last argument of a lexical predicate).

Modifiers are not associated with lexical items, but with syntactic classes. They are always **optional**.

When an **argbearer** participates in a syntactic construction, its obligatory arguments must be satisfied in the construction. Besides, they must appear in a sequence which satisfies the **precedence** relation: each argument must satisfy the `precede` predicate with respect to the argument which follows it in the left to right order of the word list to be parsed.

Let us take the example of an **arglist** associated with a verb. The verb **CONSIDER** can take, under one of its readings, an **arglist** consisting of an object and an object complement:

```
m_verb(vcomp,_,consider_2,consider,consider,consider,considers,
     considering,
       considered,considered,considered,trans,human,
       [np(oblig,posprec(1,Wnp1),object,_),
        np(oblig,posprec(2,Wnp2),object_attribute,_)]).
```

/* *he considered the claim she made an oversimplification* */

This arglist will be passed on to the predicate `arglist` when the verb phrase which has **consider** as main verb is parsed by the predicate `verbphrase`:

```
verbphrase(P1,P3,subject(SUBJ,Semsubj),Gap,Pref,
              [pred_arg_mod_structure,
              prop(vce:V,asp:A,mod:Modality,tns:Tense),
              VERB,SParse],
            Rel,Intrel,Type,Tense,aspect(Aspect),Modality,
            Number,Person,Voice,nsubject(NSUBJ,Nsem)):-
    verb(P1,P2,Class,VERB,Type,Tense,
          Number,Person,Semsubj,Args) ,
    arglist(P2,P3,Gap,Status,Pref,Preclist,Rel,
              Intrel,Voice,Parse,vp,
              Args,Func,subject(SUBJ,Semsubj),
              nsubject(NSUBJ,Nsem),
              Class),
     (nonvar(Aspect); var(Aspect),A = none ),
     (Aspect = [] , A = none; Aspect \= [],A = Aspect ),
     (nonvar(Modality); var(Modality),Modality = none ),
     (nonvar(Tense); var(Tense), Tense = present ),
     (nonvar(Voice), V = Voice; var(Voice),V = active ),
     append([NSUBJ],Parse,AParse),
     insort(AParse,SParse).
```

The predicate **`arglist`** will call **`reog`** to deal with possible subject changes induced by passivization and other subject-changing transformations and then will call **`satisfylist`** with the new, reorganized, **arglist**:

```
arglist(P0,P2,Gaps,ArgOrModFound,
            Pref,Posprec1,Rel,
            Intrel,Voice,Parse,NpOrVp,List,
            Func,subject(SUBJ,Semsubj),
            nsubject(NSUBJ,Nsem),
            Class):-
            reog(Voice,Class,subject(SUBJ,Semsubj),List,
                nsubject(NSUBJ,Nsem),Nlist,Func),
            satisfylist(P0,P2,Gaps,ArgOrModFound,Pref,Posprec1,
               Rel,Intrel,Voice,Parse,NpOrVp,Nlist,
               Func,subject(NSUBJ,Nsem)).
```

The predicate **`satisfylist`**[8] will non-deterministically **`pick`** an element out of the **arglist** and try to satisfy it by calling the predicate **`satisfy`**; it will then go on to try and satisfy the remainder of the **arglist**, making sure that the **`precede`** relation between the satisfied argument and the remainder is satisfied:

```
satisfylist(P0,P2,Gaps,ArgOrModFound,
            Pref,Posprec1,Rel,
            Intrel,Voice,Parse,NpOrVp,List,
            Func,subject(SUBJ,Sem)):-
            append(Gap1,Gap2,Gaps),
            pick(List,Elem,Tail),

/* pick[9] is non-deterministic selection of an element from a list:
List is the list to select from
Elem is the selected element
Tail is List from which the selected element has been deleted */

            satisfy(P0,P1,Gap1,Pref1,Posprec1,
                  Rel,Intrel,Parse1,
                  Elem,subject(SUBJ,Sem),
                  **subject(NEXTSUBJ,Nsem)),**
                ArgOrModFound = 1,
            **satisfylist(P1,P2,Gap2,ArgOrModFound,**
                 **Pref2,Posprec2,Rel,Intrel,Voice,**
                 **Parse2,NpOrVp,Tail,**
                 **Func,subject(NEXTSUBJ,Nsem)),**
            **precede(Posprec1,Posprec2),**
            accu(Pref,[Pref1,Pref2]),
            append([Parse1],Parse2,Parse).
```

The predicate **`satisfy`** is defined in different ways according to the argument that it is passed. In the case of CONSIDER, it will be passed two nps, and the following clause will be triggered:

```
satisfy(P0,P1,Gap,Pref,posprec(Pos,Prec),Rel,Intrel,
       [Function,Rest],
       np(Type,posprec(Pos,Prec),
       Function,Semvp),
       subject(SUBJ,Semsubj),
       subject([subject,Rest],Semvp)):-

  nsubject(Function),
  xnounphrase(P0,P1,Gap,index(J),
     Prefnp,Prec,Rel,Intrel,
      Function,
      [Function,Rest],
      Number,Person,Sem),
  sfok(Semvp,Sem),
  Pref is Prefnp + 4.
```

`Satisfy` calls the **`xnounphrase`** predicate to parse the argument np. It will be called twice, as the two elements in the **arglist** are both nps (but they do not fill the same function, the first being the object, and the second the complement, unless weight considerations have disturbed the canonical arg order).

How is the difference between **obligatory** and **optional** arguments accounted for? We have seen that **`satisfylist`** calls itself recursively. When it cannot succeed by parsing more of the input, it is allowed to succeed doing nothing, provided the arglist no longer contains obligatory args, i.e. all the remaining args are optional:

```
satisfylist(P0,P0,Gap,_,0,_,_,_,_,[],_,List,Func,_):-
                                   allopt(List).
```

`Allopt` checks that all the args are optional. It does so by looking at their **Type**, which is always the first member of the functor representing the arg (np, pp, s, etc.):

```
allopt([Head|Tail]):-
           arg(1,Head,opt),
           allopt(Tail).

allopt([]).
```

Modifiers are parsed by the predicate **`modifier`**. The crucial difference is that their parsing does not affect the arglist:

```
satisfylist(P0,P2,Gaps,ArgOrModFound,
      Pref,Posprec1,Rel,
      Intrel,Voice,Parse,NpOrVp,List,
      Func,subject(SUBJ,Sem)):-
      append(Gap1,Gap2,Gaps),
      modifier(P0,P1,NpOrVp,Gap1,
              Prefmod,Posprec1,
              Rel,Intrel,Parse1,
              subject(SUBJ,Sem)),
      ArgOrModFound = 1,
      satisfylist(P1,P2,Gap2,ArgOrModFound,
                Preflist,Posprec2,
                Rel,Intrel,Voice,
                Parse2,NpOrVp,List,
                Func,subject(SUBJ,Sem)),
      precede(Posprec1,Posprec2),
      accu(Pref,[Prefmod,Preflist]),
      append([Parse1],Parse2,Parse).
```

The predicate `modifier` will parse **modifier pps** for both nps and vps, and will also parse **ing-phrases** and **en-phrases** as np modifiers. In the case of pps it will call on the `xprepphrase` predicate, and in the case of ing and en-phrases on the `xverbphrase` predicate. Here is, for example, the definition of the predicate `modifier` for the parsing of modifier pps within vps:

```
modifier(P0,P1,vp,Gap,Prefgen,posprec(1,Precpp),
      Rel,Intrel,Tree,subject(SUBJ,Sem)):-
      xprepphrase(P0,P1,Gap,index(J),npindex(I),Pref,Precpp,
            Prepform,Rel,
            Intrel,vp_modifier,
           Tree,PPsem,PPsemnp),
      modppvp(Prepform),
      Prefgen is Pref + 2.
```

4.2. The S Level

At the highest level, we have the `parse` predicate, with three clauses: one for **declarative** sentences, a second for **yes-no questions** and a third for **wh-questions** (imperatives are not covered). This highest level is the only one at which the **Preference** value is included in the parse tree.

```
parse(P0,[],[Preference,Tree]):-
    xsentence(P0,[],[],Preference,Tree,finite,Person,Number,Voice).
```

Note that the S parsed by `xsentence` must be **finite** and that the whole string must have been traversed. The first arg of `xsentence`, **P0**, is the input word list and the second arg is the remaining word list to be traversed. It is here set to the empty list (**[]**). The third argument of `xsentence` is the gap specification. It is also set to the empty list: a main declarative clause cannot feature any gap.

```
parse(P0,[],[Preference,Tree]):-
    yesnoquestion(P0,[],Preference,Tree).

parse(P0,[],[Preference,Tree]):-
    whquestion(P0,[],Preference,Tree).
```

4.2.1. Declarative Clauses

The main predicate here is `xsentence`. In **horatio** a predicate name beginning with an **x**, such as here `xsentence`, is used to parse a phrase that can, but need not, result from the coordination of two phrases of the type indicated by the predicate name without the x. `xsentence` will take care of both simple and coordinated S's. In the case of coordinated S's, `xsentence` will make a call on `c_sentence` and then recursively call itself.

```
xsentence(P0,P2,[],Prefs,[and_sentence,S1,S2],Type,Person,
        Number,Voice):-
inlist(and,P0),
c_sentence(P0,[and|P1],[],Pref1,S1,Type,Person,Number,
        Voice),
xsentence(P1,P2,[],Pref2,S2,Type,Person2,Number2,Voice2),
accu(Prefs,[Pref1,Pref2]).
```

In the case of a simplex S (non-coordinated main clause), `xsentence` will simply make a call on `c_sentence`:

```
xsentence(A,B,C,D,E,F,G,H,I):- c_sentence(A,B,C,D,E,F,G,H,I).
```

`c_sentence` parses main clauses that can, but need not, be flanked by an adverbial subordinate clause on either side. Here is the code for a main clause preceded by an adverbial subordinate clause:

```
c_sentence(P0,P2,[],Preftot,[sentence,Adverbs,[S]],
                    finite,Person,Number,Voice):-
adverb_sentence(P0,P1,[],Prefsub,Adverbs,finite,Person1,Number1,Voice1),
sentence(P1,P2,[],Prefmain,S,finite,Person,Number,Voice),
accu(Preftot,[Prefsub,Prefmain]).
```

The `adverb_sentence` predicate parses an adverbial subordinate clause by finding a subordinator and parsing an S:

```
adverb_sentence(P0,P2,[],Pref,[SUB,S],finite,Person,Number,Voice):-
sub(P0,P1,SUB,_),
/* subordinating conjunction; last position in argument list is currently unused */
sentence(P1,P2,[],Pref,S,finite,Person,Number,Voice).
```

Finally, a **c_sentence** can have no adverbial subordinate clause, but consist of a main clause only:

```
c_sentence(A,B,C,D,E,F,G,H,I):- sentence(A,B,C,D,E,F,G,H,I).
```

We can now look at the `sentence` predicate itself.

```
sentence(P0,P2,Gaps,Prefs,[clause,VP],Type,Personvp,Number,Voice):-

append(Gapnp,Gapvp,Gaps),

/* it is more efficient to do the appending of the gaplists now, because there are cases where we know that
the result of the appending must be the empty list */

/* the subject np */

xnounphrase(P0,P1,Gapnp,index(I),Prefnp,Weight,Rel1,
                Intrel1,subject,SUBJ,Number,Personnp,Semsubjnp) ,
var(Rel1),
```

/* the **Rel** variables must be uninstantiated; it is the conjunction of an antecedent and a relative clause that releases, i.e. **uninstantiates** the **Rel** var; see the treatment of relative clauses below, p. 99 */

/* Subject-verb agreement */

```
agree(Personnp,Number,Personvp),
```

/* the first two arguments of **agree** come from the np, the third from the vp */

/* the vp */

/* the parse tree corresponding to the subject will be included in the parse tree returned by the **xverbphrase** predicate */

```
xverbphrase(P1,P2,subject(SUBJ,Semsubjvp),
            Gapvp,Prefvp,VP,Rel2,
            Intrel2,Type,Tense,aspect(Aspect),Modality,
            Number,Personvp,
            Voice,
            nsubject(NSUBJ,Nsemsubjvp)),
```

/* **Nsemsubjvp** records the semantic restriction on the subject if the verbphrase has been found to be passive; the **Voice** variable is left uninstantiated in the active */

/* **nsubject** potentially changes the deep subject (passives, for instance); the semantic check (sfok) must therefore be between **Nsemsubjvp** and the **Semsubjnp** list */

```
var(Rel2),
```

/* semantic check */
```
sfok(Nsemsubjvp,Semsubjnp),
```

/* computing the preference index */
```
accu(Prefs,[Prefnp,Prefvp,4]).
```

4.2.1.1. Subject / Operator Agreement Rules

Agreement between the subject np and the verb phrase is checked by the predicate `agree`, whose code is the following:

```
agree(first,sing,firstsg).
agree(first,plural,other).
agree(second,sing,other).
agree(second,plural,other).
agree(third,sing,thirdsg).
agree(third,plural,other).
```

The first two arguments come from the subject np: **Person** and **Number**. The third argument comes from the verb phrase. The value **firstsg** is necessary for the forms **was** and **am** of **BE**; the value **thirdsg** is used to capture the agreement feature of **was** and of the present tense third person of most verbs. The value **other** is a catchall value.

A call to the `agree` predicate is made at the S level, in the definition of the `sentence` predicate, repeated below with the relevant pieces in bold type:

```
sentence(P0,P2,Gaps,Prefs,[clause,VP],Type,Personvp,
               Number,Voice):-
append(Gapnp,Gapvp,Gaps),
xnounphrase(P0,P1,Gapnp,index(I),Prefnp,Weight,Rel1,
              Intrel1,subject,SUBJ,Number,Personnp,Semsubjnp) ,
var(Rel1),
agree(Personnp,Number,Personvp),
xverbphrase(P1,P2,subject(SUBJ,Semsubjvp),
              Gapvp,Prefvp,VP,Rel2,
              Intrel2,Type,Tense,aspect(Aspect),Modality,
              Number,Personvp,
              Voice,
              nsubject(NSUBJ,Nsemsubjvp)),
var(Rel2),
sfok(Nsemsubjvp,Semsubjnp),
accu(Prefs,[Prefnp,Prefvp,4]).
```

4.2.1.2. Priority among Person Features

When dealing with coordinated nps we need to call on the `priority` predicate to determine the person of the np resulting from the coordination. For instance *she and I* is first person, not third. The code for the `priority` predicate reflects the person hierarchy: first has priority over second and third, and second over third:

```
priority(first,second,first):- !.
priority(first,third,first):- !.
priority(third,first,first):- !.
priority(second,first,first):- !.
priority(third,second,second):- !.
priority(second,third,second):- !.
priority(X,X,X).
```

The first two arguments are the person values of the first and second member of the coordination; the third arg is the resulting person value, the one associated with the coordination as a whole. Note that we can use the cut and that we need so many clauses because we need to cater for the two positions that the winning value can have in the coordination. The last clause deals with the case where the two coordinated nps share the person value.

The predicate `priority` is called in the parsing of coordinated nps:

```
xnounphrase(P0,P2,[],index(I),Pref,Weight,Rel,Intrel,
               Function,
              [Function,[and_nounphrase,Rest1,Rest2]],
              plural,Person,Sem):-
    inlist(and,P0),
    nounphrase(P0,[and|P1],[],_,Pref1,Weight1,Rel,
             Intrel,Function,
             [Function,Rest1],
             Number1,Person1,Sem),
    xnounphrase(P1,P2,[],_,Pref2,Weight2,Rel,
              Intrel,Function,
              [Function,Rest2],
              Number2,Person2,Sem2),
    priority(Person1,Person2,Person),
    accu(Pref,[Pref1,Pref2]),
    accu(Weightaccu,[Weight1,Weight2]),
    Weight is (Weightaccu/2)+1.
```

4.2.1.3. Passives

4.2.1.3.1. Parsing

Passivization induces changes in the **arglist** associated with the passivized verb: the active subject is left out of the arglist, or demoted to head of a by-phrase; one of the other arguments is promoted to subject.

In **horatio** it is the predicate **`reog`** (for **reorganization of the arglist**) which takes care of accounting for these changes, i.e. maintaining the relation between the arglist as expressed in the lexicon and the arguments as occurring in the passivized clause.

It should be noted that in **horatio** the voice value is left **uninstantiated** when the clause is active, and set to **`passive`** when the clause is passive. Here is the relevant definition of the **`reog`** predicate:

```
reog(passive,Class,subject([SFunc,Rest],Semsubj),List,
          nsubject(NSUBJ,Nsem),Nlist,Func):-
     pick(List,Elem,Remainder),
     Elem =.. Elemlist,

/* tree to list conversion: each element in the arglist is a tree whose root indicates its nature: np, pp, s, etc... */

     Elemlist = [Nature,Type,Posprec,Func,Nsem|_],
     psubject(Func),

/* Func points to a function in the active S that can be promoted to subject of passive; this is checked by psubject  */

     NSUBJ=[Func,Rest],

/* note that the deep function is preserved in the parse tree */

     append(Remainder,
           [byphrase(opt,posprec(_,3),subject,Semsubj)],
          Nlist).

/* the subject position opened up for the active vp yields a by-phrase position to be appended to the remaining arglist */
```

The code for **`psubject`** enumerates the various functions that can fill in the subject role in a passivized S:

```
psubject(indirect_object).
psubject(object).
psubject(surf_object).
```

The last one will be used in the parsing of such sentences as *She is expected to teach*, where the subject (*she*) results from a promotion to surface subject of the main clause of a surf_object (*They expect* *her to teach*). This surface object results from the promotion of the subject of the embedded clause (via subject-to-object raising), which is the real (i.e. deep), clausal object (**she to teach**).

`Reog` is always called in the definition of the **`arglist`** predicate (passage from **`arglist`** to **`satisfylist`**), but it is allowed to succeed doing nothing in the case of active clauses to which neither raising nor extraposition applies:

```
reog(Voice,Class,subject(SUBJ,Semsubj),List,
          nsubject(SUBJ,Semsubj),List,Func):-
     Class \= sraising,
     Class \= extrapos,
     var(Voice),!.
```

The code for **`arglist`** includes a call to **`reog`**, which, if appropriate, will reassign the subject and modify the argument list:

```
arglist(P0,P2,Gaps,ArgOrModFound,
          Pref,Posprec1,Rel,
          Intrel,Voice,Parse,NpOrVp,List,
          Func,subject(SUBJ,Semsubj),
          nsubject(NSUBJ,Nsem),
          Class):-
          reog(Voice,Class,subject(SUBJ,Semsubj),List,
              nsubject(NSUBJ,Nsem),Nlist,Func),
          satisfylist(P0,P2,Gaps,ArgOrModFound,Pref,Posprec1,
             Rel,Intrel,Voice,Parse,NpOrVp,Nlist,
             Func,subject(NSUBJ,Nsem)).
```

4.2.1.3.2. Generation

Passive in generation is the inverse of passive in analysis. Analysis yields parse trees which exhibit **deep** grammatical functions, and the surface relations must be re-established before the strings corresponding to the surface phrases can be generated. The whole reorganization process is in the hands of the predicate **`prepgen`**, which adheres to the **transformational cycle** (see the section on the cycle on page 108), and **`prepgen`** calls on **`passive`** where appropriate. `Passive` has a fair number of clauses on account of the various geometries of the trees it is supposed to work on: they may contain indirect objects, direct objects, subjects, in various orders. `Passive` has a clause for each configuration. It is not necessary to examine them all here.

First, let us look at the clause for `passive` which applies when voice is **active**. Obviously enough, we do not want `passive` to fail in such cases, but rather to succeed trivially, i.e. leaving the tree untouched:

```
passive([H1,[pred_arg_mod_structure,prop(vce:active,B,C,D)|R1]|R2],
          [H1,[pred_arg_mod_structure,prop(vce:active,B,C,D)|R1]|R2] ):-
second_header(H1),
!.
```

`Second_header(H1)` is only a check on the environment, namely on the clause header.

As an example of a clause for `passive` that actually does something, let us look at the one for clauses with an indirect object that can be demoted to subject of the passive clause:

```
passive([H1,[pred_arg_mod_structure,prop(vce:passive,B,C,D),Pred1,
   [[Subject,S],
    [object|Robject],
    [indirect_object|Rio]|Otherargs]]],

    [H1,[pred_arg_mod_structure,prop(vce:passive,B,C,D),Pred1,
   [[subject_pass|Rio],[object|Robject],
    [pp_arg,[prepphrase,index(I),prep(by),[np_arg_of_prep,
          S]]]|Otherargs]]]):-

subject_active(Subject),
second_header(H1) .
```

The actual work gets done in the head of the clause, the body only containing checks on the environment, namely a check on the clause header (H1) and on the subject of the active clause.

The reorganization work consists in demoting the active subject to **by-phrase** in the passive clause. The by-phrase is created as a **pp_arg**, with preposition BY governing an **np_arg_of_prep** corresponding to the body of the subject in the active clause. The second transformation concerns the status of the active indirect object. Its body is not changed, but its function is turned to **subject_pass**, i.e. subject of a passive S.

Note that we need to take care of the case where the **by-phrase** has an np arg that is **uninstantiated**, so that we do not generate a surface by-phrase with an empty np inside. The following clause for `gen` does the job:

```
gen([pp_arg,[prepphrase,index(I1),prep(by),
     [np_arg_of_prep,[nounphrase,index(I2)|VAR]]]],[]):-
var(VAR).
```

The second arg of `gen`, which houses the generated string, is set here to the empty list (**[]**).

4.2.1.4. The Assignment of the Subject Role

Control relations (which determine the assignment of one of the predicate's arguments to the subject slot of a nonfinite complement clause, such as an infinitive or ing clause) are computed while syntactically parsing the S. They are not taken out of the syntactic component and assigned to semantic interpretation rules, as they are in **CLE** (see **Alshawi et al. 1992**, section 5.3.2, **VP control phenomena**, p. 101 and foll.)

4.2.1.4.1. Control

In **Horatio**, **control** takes care of the **coindexing** of a **controlling** argument with the argument it controls, which is always the **subject** of a subordinate clause which belongs to the same **arglist** as the controller.

In generation, the task consists in **ghosting** the controlled subject, i.e. depriving it of lexical material, so that nothing is generated.

4.2.1.4.1.1. Parsing

Consider the case of verbs such as **WANT** constructed with an np followed by a to-infinitive. The control relation is between the surface object of WANT and the subject of the infinitive. In a sentence such as *The woman wants the teacher to teach*, *the teacher* will be assigned as **running subject**[10] (i.e. subject of any vp further to the right) when it is parsed to satisfy the np argument in the **arglist** of WANT:

```
m_verb(vinf,_,want_1,want,want,want,wants,wanting,
       wanted,wanted,wanted,trans,living,
       [np(opt,posprec(1,Wnp),subject_inf,_),
        np_vp(oblig,to_inf,object)]).
```

The running subject assignment will be done by the `satisfy` predicate:

```
satisfy(P0,P1,Gap,Pref,posprec(Pos,Prec),Rel,Intrel,
             [Function,Rest],
             np(Type,posprec(Pos,Prec),
             Function,Semvp),
             subject(SUBJ,Semsubj),
             subject([subject,Rest],Semvp)):-
   nsubject(Function),
   xnounphrase(P0,P1,Gap,index(J),
         Prefnp,Prec,Rel,Intrel,
          Function,
          [Function,Rest],
          Number,Person,Sem),
   sfok(Semvp,Sem),
   Pref is Prefnp + 4.
```

The first **subject** functor in the `satisfy` argument list points to the running subject on entering and the second to the running subject on exiting the procedure. The parse tree returned by the `xnounphrase` is unified with the first argument of the second subject functor, except for the function, which is set to **subject** in the second subject functor.

`Nsubject` checks that the function is one that can yield the running subject. It is defined by the following code:

```
nsubject(subject_inf).
nsubject(subject_ing).
nsubject(surf_subject).
nsubject(object).
nsubject(surf_object).
nsubject(indirect_object).
```

Now that the **running subject** has been set, it can be assigned as subject to the to-infinitive. This is done when the to-infinitive argument is satisfied by a call to the `satisfy` predicate:

```
satisfy([to|P0],P1,Gap,Prefgen,Posprec,Rel,Intrel,
             [Function,[clause,Tree]],
             np_vp(Type,to_inf,Function),
             subject([Sfunction,Treesubj],Semsubj1),
             subject([Sfunction,Treesubj],Semsubj1)):-
  xverbphrase(P0,P1,subject([subject,Treesubj],Semsubj2),
            Gap,Pref,Tree,Rel,
            Intrel,infinitive,Tense,
            aspect(Aspect),Modality,
            Number,Person,Voice,
            nsubject(NSUBJ,Nsem)),
  checksem(Nsem,Semsubj1),
  Prefgen is Pref + 4.
```

4.2.1.4.1.2. Generation

In generation, the controlled subject needs to be **ghosted**, i.e. deprived of lexical material, so that it does not appear in the generated string. We need several clauses for the predicate `control`, depending on the structural positions of the controller and of the clause with the controlled subject. Let us examine the one for controller as first argument and clause containing the controlled subject as second:

```
control([H2,
[pred_arg_mod_structure,Prop1,[predicate(Pred1,AgrPred1)],
   [[Controller,[nounphrase,index(I),AgrNP|Rest1]],
   [H1,[clause,[pred_arg_mod_structure,Prop2,[predicate(Pred,agr(Agr))],
   [[Subject,[nounphrase,index(I),AgrNP|Restsubj]]|Otherargs]]]]|R1]]],

[H2,[pred_arg_mod_structure,Prop1,[predicate(Pred1,AgrPred1)],
   [[Controller,[nounphrase,index(I),AgrNP|Rest1]],
   [H1,[clause,
[pred_arg_mod_structure,Propnew2,[predicate(Pred,agr(Agr))],
   [[Subject,[nounphrase,index(I),AgrNP|VAR]]|Otherargs]]]]|R1]]]):-

nonvar(I),
allsubject(Subject),
cv(Pred1,Requires),
second_header(H2),
first_header(H1),
controller(Controller),
nonfinite(Agr),
Prop2 = prop(Voice,Aspect,Mod,Tns),
Propnew2 = prop(Voice,Aspect,Mod,tns:Requires).
```

We first check that the index is **instantiated**. We do not want `control` to be responsible for instantiation through unification.

This is followed by a series of other checks on the environment of the rule. We make sure that the variable **Subject** refers to a subject function (the code for `allsubject` is given on page 113), and that the **clause headers** are as expected (see page 110). We also check that **Controller** points to a function that can control. The code for the `controller` predicate is the following:

```
controller(subject).
controller(object).
controller(subject_inf).
controller(subject_pass).
controller(indirect_object).
```

The `cv` clause makes a call on the lexicon (macro clause `m_verb`) and checks that the verb class is that of a control verb with the help of the predicate `cvclass`. The predicate `cvclass` also gives the nature of what will remain from the controlled clause: it will be a vp, and `cvclass` indicates whether it will be **gerundive** or **infinitive**. This value returned by `cvclass` will come to occupy the position for the tense value, which does not apply to non-finite clause. This mechanism is further explained on page 114.

The last check is on the agreement feature of the predicate in the controlled clause. It cannot be a finite clause, as control does not apply to finite clauses (rather pronominalization does).

The ghosting job consists in replacing **Restsubj**, the body of the subject in the parse tree, by **VAR**, an uninstantiated variable.

The generator has a clause which ensures that **ghosted** nps (and other ghosted elements) do not generate any output. It simply stipulates that **uninstantiated** variables generate the empty list, and this empty list disappears in the list appending and flattening processes which complete generation:

```
gen(X,[]):- var(X), !.
```

4.2.1.4.2. Raising

We need to distinguish **Subject-to-Object** raising and **Subject-to-Subject** raising. The first case is exemplified by the relation between the string *I believe John to teach linguistics* and the parse produced, revealing that the surface object is the deep subject of the subordinate clause (something linearizable as **I believe [John teach linguistics]**.) The second case is exemplified by the surface string *John seems to teach linguistics* and its "source" (in transformational terms - merely metaphorical here, since **horatio** does not assume the existence of **transformations**, but is only interested in revealing **relations**): **[John teach linguistics] seems**.

4.2.1.4.2.1. Parsing

4.2.1.4.2.1.1. Subject-to-Object Raising

In the lexicon subject-to-object raising verbs belong to the **`vinf`** class. Consider the entry for **BELIEVE** under the relevant reading :

```
m_verb(vinf,_,believe_1,believe,believe,believe,believes,believing,
      believed,believed,believed,trans,human,
      [np(oblig,posprec(1,Wnp),surf_object,_),
       np_vp(oblig,to_inf,object)]).

/* he believes him to teach linguistics */
```

When the arglist is satisfied, the np (such as **him** in the example) will be assigned **surf_object** as function. This function is an **athematic** one, and the np will therefore not appear in the parse tree, on account of the following defining clause for the predicate **`drop`**: **`drop([surf_object,Rest])`**.

However, this np will have been made **running subject** by the **`satisfy`** predicate, since **surf_object** is one of the functions accepted by **`nsubject`**: **`nsubject(surf_object).`**

```
satisfy(P0,P1,Gap,Pref,posprec(Pos,Prec),Rel,Intrel,
            [Function,Rest],
            np(Type,posprec(Pos,Prec),
            Function,Semvp),
            subject(SUBJ,Semsubj),
            subject([subject,Rest],Semvp)):-
   nsubject(Function),
   xnounphrase(P0,P1,Gap,index(J),
         Prefnp,Prec,Rel,Intrel,
          Function,
          [Function,Rest],
          Number,Person,Sem),
   sfok(Semvp,Sem),
   Pref is Prefnp + 4.
```

When the **np_vp** construction in the arglist is satisfied, it is assigned the running subject as subject:

```
satisfy([to|P0],P1,Gap,Prefgen,Posprec,Rel,Intrel,
        [Function,[clause,Tree]],
        np_vp(Type,to_inf,Function),
        subject([Sfunction,Treesubj],Semsubj1),
        subject([Sfunction,Treesubj],Semsubj1)):-
  xverbphrase(P0,P1,subject([subject,Treesubj],Semsubj2),
        Gap,Pref,Tree,Rel,
        Intrel,infinitive,Tense,
        aspect(Aspect),Modality,
        Number,Person,Voice,
        nsubject(NSUBJ,Nsem)),
  checksem(Nsem,Semsubj1),
  Prefgen is Pref + 4.
```

The function of the whole clause is read off the lexicon (`object`). We therefore end up with the clausal object we need: **him to teach**, where **him** is assigned subject role.

4.2.1.4.2.1.2. Subject-to-Subject Raising

Subject-to-Subject Raising verbs are assigned the **sraising** class. Consider the entry for **SEEM** under the relevant reading in the lexicon:

```
m_verb(sraising,_,seem_1,seem,seem,seem,seems,seeming,
    seemed,seemed,seemed,intrans,_,
    [vp(oblig,subject)]).
/* he seems to have taught linguistics */
```

The **vp** structure in the arglist will be satisfied by the following defining clause for the predicate `satisfy`:

```
satisfy([to|P0],P1,Gap,Prefgen,Posprec,Rel,Intrel,
        [Function,[clause,Tree]],
        vp(Type,Function),
        subject([Sfunction,Treesubj],Semsubj1),
        subject([Sfunction,Treesubj],Semsubj1)):-
  xverbphrase(P0,P1,subject([subject,Treesubj],Semsubj2),
        Gap,Pref,Tree,Rel,
        Intrel,infinitive,Tense,
        aspect(Aspect),Modality,
        Number,Person,Voice,
        nsubject(NSUBJ,Semsubj1)),
  Prefgen is Pref + 2.
```

The whole infinitive clause is assigned the subject role (read off the lexical entry). Here too the **running subject** (in this case the subject of SEEM) is assigned as subject of the infinitive clause. But it also needs to be made athematic in the higher clause, so that it does not appear as subject in the parse tree, the real subject being clausal. This demotion is accomplished by `reog`:

```
reog(Voice,Class,subject([subject,Rest],_),List,
    nsubject([surf_subject,Rest],_),List,Func):-
     ( Class = sraising; Class = extrapos ),
     var(Voice),!.
```

4.2.1.4.2.2.1. Subject-to-Object Raising

Subject-to-Object raising in generation is accomplished by the predicate **`oraising`**. It is examined in detail in the section on the **cycle**, where the sentence we follow through the cycle is one where subject-to-object raising applies, namely *I believe him to have been killed*. The reader is referred to page 112 and following.

4.2.1.4.2.2.2. Subject-to-Subject Raising

In generation terms **Subject-to-Subject** raising, as in *we tend to like students*, applies to subordinate clauses in subject function. Their own subject is promoted to subject of the main clause, leaving a subjectless subordinate clause which will be non-finite (infinitive).

The predicate taking care of subject-to-subject raising in generation is **`sraising`**. It has a clause which succeeds trivially (returns its first arg in its second arg) in those cases where the transformation is not applicable, i.e. where the predicate of the main clause is not of the appropriate class:

```
sraising([clause,[pred_arg_mod_structure,Prop1,
        [predicate(Notraising,Agr1)]|R1]|R2],
        [clause,[pred_arg_mod_structure,Prop1,
        [predicate(Notraising,Agr1)]|R1]|R2]):- not(sraise(Notraising,_)), !.
```

The predicate **`sraise`** calls on the lexicon to check whether the verb is a subject-to-subject raising one; the appropriate verb class is the **sraising** one:

```
sraise(Sraisingverb,to):-
m_verb(sraising,_,Sraisingverb,_,_,_,_,_,_,_,_,_,_).
```

Let us now turn to the case where subject-to-subject raising is applicable. The defining clause is as follows:

```
sraising([clause,
[pred_arg_mod_structure,Prop1,[predicate(Sraising,Agr1)],
   [[Subject1,[clause,[pred_arg_mod_structure,Prop2,Pred,
   [[Subject2|Rest]|Otherargs]]]]]]],

[clause,[pred_arg_mod_structure,Prop1,[predicate(Sraising,Agr2)],
   [[Subject2|Rest],
[Subject1,[clause,[pred_arg_mod_structure,Propnew2,Pred,
   [Otherargs]]]]]]]):-

sraise(Sraising,Requires),
allsubject(Subject1),
allsubject(Subject2),
 ( Agr1 = agr(Type,Tns,Number,Person), Agr2 = agr(Type,Tns,_,_);
   Agr2 = Agr1 ),
Prop2 = prop(Voice,Aspect,Mod,tns:present),
Propnew2 = prop(Voice,Aspect,Mod,tns:Requires).
```

Apart from the manipulation of the syntactic relations performed in the head of the clause, the following points should be noted:

1) The **agreement functor** of the main predicate is changed. We do not keep the **Number** and **Person** values, if we have them. If we do have them, we disinstantiate them; if we do not have them (the **agr**

functor is not the 4-place one), we can copy the old **agr** functor (**Agr1**) into the new one (**Agr2**). The reason why we cannot keep the **Person** and **Number** values is obvious: agreement will have to be made with the new subject, the promoted one.

2) To simplify generation we stipulate that Subject-to-Subject raising cannot apply to subordinate clauses that are **not present tense**. If tense is past, **extraposition** should be allowed to apply, not subject-to-subject raising. Compare:

(a) it appears that he **interviewed** browne
(b) he appears to be interviewing browne

In this way we do not need to introduce a **perfect** value in the **aspect** slot of **Propnew2** when tense is non-present in **Prop2** (to generate *he appears to have interviewed browne* as variant of (a)). Therefore in **Prop2** (the **Property** field of the subordinate clause), we check that the tense value is present before overwriting it with the non-finite value returned by the **`sraise`** predicate.

4.2.1.4.3. Extraposition

Extraposition is the end-placing of a subject clause and the filling of the vacated subject position with place-filler *IT*. It is the relation between the surface string *It seems that John teaches linguistics* and **[John teach linguistics] seems**, which can also be related to the surface string *John seems to teach linguistics* by raising.

4.2.1.4.3.1. Parsing

In analysis the surface subject needs to be demoted. This is accomplished by the **`reog`** clause that we have looked at in the preceding section.

In the lexicon extraposition verbs are assigned the **extrapos** class:

```
m_verb(extrapos,_,seem_1,seem,seem,seem,seems,seeming,
      seemed,seemed,seemed,intrans,it,
      [s(oblig,posprec(1,W),subject)]).
/* it seems that he has taught linguistics[11] */
```

The **arglist** contains an S that is assigned the subject role. It is satisfied by the following defining clause of the predicate **`satisfy`**:

```
satisfy(P0,P1,Gap,Prefgen,posprec(Pos,4),
          Rel,Intrel,[Function,Tree],
        s(Type,posprec(Pos,4),Function),
        subject(SUBJ,Semsubj),
        subject(SUBJ,Semsubj)):-
       xsentence(P0,P1,Gap,Pref,Tree,finite,Person,Number,Voice),
       Prefgen is Pref + 4.
```

4.2.1.4.3.2. Generation

In generation we need to put the place-filler **IT** back into the parse, so that it gets generated as surface subject of **SEEM**. This is accomplished by the predicate **`extrapos`**:

```
extrapos([clause,[pred_arg_mod_structure,Prop1,
[predicate(Extrapos,Agr1)],
   [[subject,[clause|Rest]]|Otherargs]]],
[clause,[pred_arg_mod_structure,Prop1,

[predicate(Extrapos,Agr1)],
   [[subject,[nounphrase,index($index$),agr(third,sing),
                                   ppro(third,sing,neuter)]],
    [object,[clause|Rest]]|Otherargs]]]):-
expos(Extrapos,Requires).
```

The body of the clause simply checks that the verb belongs to the right class. The clausal subject is transferred to the object argument (it will be generated as if it were an object, which it is positionally), and a new subject is created with the appropriate agreement functor (**agr(third,sing)**) and the appropriate body (**ppro(third,sing,neuter)**).

4.2.2. Non-Declarative Main Clauses

4.2.2.1. Yes-No Questions

The basic format of a YES-NO question is the following:

Operator - Sentence with incomplete verb group

The **operator** is the first auxiliary of the verb group if the latter includes any auxiliary, or DO otherwise. The operator must be **finite**.

4.2.2.1.1. Parsing

In **horatio** we check that the operator and the remainder of the verb group build up a licit verb group (each auxiliary has a **Required** value, which indicates the **Type** of the verb form that can be found to its right).

```
yesnoquestion(P1,P3,Prefs,[yes_no_question,
        [clause,
            [P,prop(vce:V,asp:A2,mod:Modality,tns:Tense) |Rest]]
   ]):-
    xaux(P1,P2,Modality,finite,Required,Number,Person,Tense),
    checkaux3(Asp,Required,V),
    xsentence(P2,P3,
           [],
           Prefs,
           [clause,
            [P,prop(vce:V,asp:A,mod:M,tns:T) |Rest]],
           Required,Person,Number,V),

    (nonvar(Modality); var(Modality),Modality = none ),

/* we have a value none in case there is no modal auxiliary */

    myappend(Asp,A,Aspect),

/* myappend appends the aspect value contributed by the auxiliary parsed by xaux to the aspect value
returned by the parsing of the S (xsentence); the code for myappend is to be found on page 66 */

    (nonvar(Aspect); var(Aspect),A2 = none ),
    (Aspect = [] , A2 = none; Aspect \= [],A2 = Aspect ).
```

/* the returned value is **none** if Aspect is uninstantiated or the empty list */

The value for **Voice** is computed by **`checkaux3`**, whose code is the following (further discussed on page 65):

```
checkaux3([perfect],en_active,_).

checkaux3([progressive],ing,_).

checkaux3([],Required,_):-
                        Required \= en_passive,
                        Required \= en_active,
                        Required \= ing.

checkaux3([],en_passive,passive).
```

The first argument is the contribution the auxiliary makes to the **Aspect** list; the second argument is the **Required** feature; the third argument is **Voice**, which is to be left **uninstantiated** if voice is active.

Looking at auxiliaries in the lexicon, we find both a **Required** (fifth argument) and a **Type** (fourth argument) value:

aux([does|X],X,_,**finite**,**infinitive**,sing,thirdsg,present).
aux([been|X],X,_,**en_active**,**ing**,_,_,_).

Does is **finite** and requires an **infinitive**; **been** can be the **active past participle** and requires an **ing-form**:

Does he know you ?
He has been reading a book.

The treatment of YES-NO questions needs to be different when the question opens with a **non-auxiliary**, i.e. **copula BE**. We do not have a verb group in the remaining S, and we need to look at the whole YES-NO question as having the following structure:

Copula - Subject - Copula's arglist

Is	the teacher	a genius ?
Are	they	aware of the problems ?

We therefore have the following piece of code in **horatio:**

```
yesnoquestion(P1,P4,Prefint,[yes_no_question,
      [clause,
         [pred_arg_mod_structure,
          prop(vce:active,asp:none,mod:none,tns:Tense),
      VERB,[SUBJ,Parse]]]
  ]):-
  verb(P1,P2,cop,VERB,finite,Tense,
       Number,Personvp,Semsubj,Args) ,

/* cop is verb class of non-auxiliary BE (copula verb ) */

  xnounphrase(P2,P3,[],index(I),Prefnp,Weight,Rel1,
           Intrel1,subject,SUBJ,Number,Personnp,Semsubjnp),
  **agree(Personnp,Number,Personvp),**
  arglist(P3,P4,[],Status,Prefarg,Preclist,Rel,
         Intrel,Voice,[Parse],vp,
         **Args**,Func,subject(SUBJ,Semsubj),
       nsubject(NSUBJ,NSemsubj), cop),
  accu(Prefint,[Prefnp,Prefarg,4]).
```

Note that we must perform the agreement check between the copula and the subject np (call to the **`agree`** predicate). The arglist of the copula is parsed by a call to the **arglist** predicate. As far as the verbal group properties are concerned, we know that we have no aspectual or modality values (there would be an auxiliary in that case), and that voice is active.

4.2.2.1.2. Generation

When we have **copula BE** as main verb, we need to generate the copula, generate the subject, and then generate the remainder, i.e. the copula's arglist and modifiers. The generator therefore looks into the parse tree to make sure that the predicate is BE. It also needs to isolate the first element in the arglist, and make sure that it is the subject. Note that we know that voice is active, and that there is no auxiliary for either aspect or modality:

```
gen([yes_no_question,[clause,[pred_arg_mod_structure,
     prop(vce:active,asp:none,mod:none,tns:Tns),
     [predicate(be,Agr)],
     [[Subject,[nounphrase,I,agr(Person,Number)|Restsubj]]|Otherargs]]]],
      Sentence):-

subject(Subject),
gen([Subject,[nounphrase,I,agr(Person,Number)|Restsubj]],Gensubject),
gen([predicate(be,Agr)],Vform,Part),
append([Part],Otherargs,Args),
insort(Args,Sortedargs),
gen(Sortedargs,Genargs),
append([Vform],[Gensubject],List1),
append(List1,[Genargs],Sentence).
```

In all other cases we do have one or more auxiliaries. We need to isolate the first auxiliary, because it must appear in front, before the subject. The generation of auxiliaries is carried out by the **`genyesno`** predicate, whose code is examined in the section on the **verb phrase** (see p.65 and following). It returns a **list** of auxiliaries, and we append the first in front of the clause, insert the subject, and then append the remaining auxiliaries (there may be none), the verb, and finally the remaining args and modifiers:

```
gen([yes_no_question,[clause,[pred_arg_mod_structure,
     Properties,[predicate(Predicate,Agr)],
     [[Subject,[nounphrase,I,agr(Person,Number)|Restsubj]]|Otherargs]]]],
      Sentence):-

subject(Subject),
gen([Subject,[nounphrase,I,agr(Person,Number)|Restsubj]],Gensubject),
genyesno(Properties,Agr,Person,Number,[First|Others]),
gen([predicate(Predicate,Agr)],Vform,Part),
append([Part],Otherargs,Args),
insort(Args,Sortedargs),
gen(Sortedargs,Genargs),
append([First],[Gensubject],List1),
append(Others,[Vform],List2),
append(List2,[Genargs],List3),
append(List1,List3,Sentence).
```

4.2.2.2. WH Questions

They are made up of a **wh-group** and a clause with a **gap** which can be filled by that wh-group.

4.2.2.2.1. Parsing

We need to distinguish two cases:

a) the wh-group is the subject

We parse the sentence as a **wh-group** followed by a **sentence** which lacks a **subject**. The link between the two parts is provided by **coindexing**.

```
/* NP wh in subject position; the S does or does not feature an auxiliary:

who knows the teacher ?
who has seen the teacher ?
who is the teacher ? */

whquestion(P0,P2,Prefint,[wh_question,NP,S]):-
 nounphrase(P0,P1,[],index(I),Prefnp,Weight,
            Semrelint,int,Functioninint,
            NP,sing,third,Semwholenp),

/* this np must have int as 8th arg, i.e. it must be or contain an interrogative element */

    nonvar(Semrelint),
    xsentence(P1,P2,
            [gap(_,npgap,Npsem,_,Functioninint,_,index(I),_)],
            Prefs,S,finite,thirdsg,sing,Voice),

/* only singular subject is catered for here: * who teach linguistics ? */

    nonvar(Functioninint),
/* the gap is a real one */

    Functioninint = subject,
/* the function of the missing element is subject */

    NP \=  [_,[_,_,_,[interrogative(whom)]]],
/* we can't have whom as subject */

    sfok(Npsem,Semwholenp),
    accu(Prefint,[Prefnp,Prefs]).
```

b) the wh-group is not the subject

Examples:

  A book about whom has he read ?
  Whom has the teacher seen ?

Here we need to be able to look into the parse trees, to account for the auxiliary which is detached from the remainder of the verb phrase, being separated from it by the subject. The relevant piece of code is the following:

```
whquestion(P0,P3,Prefint,[wh_question,NP,
        [clause,
        [P,prop(vce:V,asp:A2,mod:Modality,tns:Tense) |Rest]]
    ]):-

    nounphrase(P0,P1,[],index(I),Prefnp,Weight,
            Semrelint,int,Functioninint,
            NP,Number,third,Semwholenp),
    nonvar(Semrelint),
    xaux(P1,P2,Modality,finite,Required,Numbers,Person,Tense),

/* the auxiliary contributes the modality and tense features */

    checkaux3(Asp,Required,V),
    xsentence(P2,P3,
             [gap(_,npgap,Npsem,_,Functioninint,_,index(I),_)],
           Prefs,
            [clause,
             [P,prop(vce:V,asp:A,mod:M,tns:T) |Rest]],
             Required,Person,Numbers,V),
    (nonvar(Modality); var(Modality),Modality = none ),
    myappend(Asp,A,Aspect),
    (nonvar(Aspect); var(Aspect),A2 = none ),
    (Aspect = [] , A2 = none; Aspect \= [],A2 = Aspect ),
    nonvar(Functioninint),
    Functioninint \= subject,
    sfok(Npsem,Semwholenp),
    accu(Prefint,[Prefnp,Prefs]).
```

Note the sharing of the function **Functioninint** between the opening **wh-group** and the **gap** in the S. Note also the semantic check between **wh-group** and **gap** (call to **`sfok`**).

PP gaps receive a similar treatment.

4.2.2.2.2. Generation

In the case of a **subject** WH-group, we generate the WH-group and then generate the remainder as a normal clause (recall that it is a gapped clause, since it lacks a subject):

```
generate([wh_question,Whgroup,Clause],Sentence):-
Whgroup = [subject|Rest],
gen(Whgroup,Whgroupgen),
generate(Clause,Ynogen),
append([Whgroupgen],[Ynogen], Sentence).
```

Note the distinction between the two generation predicates. **`Gen`** generates directly, whereas **`generate`** first makes sure that **control** and other function-changing processes (**raising**, **passivization**, etc.) have been undone (See section on the **cycle**, p.108) by calling **`prepgen`**:

```
generate(Tree,Sentence):-
   prepgen(Tree,Treeprep),
   gen(Treeprep,List),
   flatten(List,Sentence).
```

When the WH-group is **not** the subject, we generate the WH-group and then the remainder as a **YES-NO** question. Generating the clausal part as a YES-NO question ensures that the clause begins with an auxiliary, and that DO is inserted if no auxiliary is present.

```
generate([wh_question,Whgroup,Clause],Sentence):-
Whgroup \= [subject|Rest],
gen(Whgroup,Whgroupgen),
generate([yes_no_question,Clause],Ynogen),
append([Whgroupgen],[Ynogen], Sentence).
```

4.3. The Phrase Level

4.3.1. Noun Phrases

4.3.1.1. Parsing

In the parsing of noun phrases we have to pay attention to the problem of **structural weight**, as this is a determining element in the position of nps with respect to the verb, each other and other phrasal elements such as pps.

The lightest nps are of course **np gaps**. They are assigned a weight of zero. Next in order of increasing weight are **personal, relative** and **interrogative pronouns**, which are assigned a weight of 1, which ensures that they stay close to the verb. **Core nps** get a weight of 2, provided the noun or adjective they contain does not have an **instantiated** arglist. In the latter case they get a weight of 3, like other **heavy** nps, such as those that include a **relative clause**.

From the preceding paragraph it is clear that the process which computes weight must be sensitive to whether the **arglist** of a constituent **argbearer** has been satisfied. This is possible because the predicate **`satisfylist`** returns in its fourth argument a **Status**, i.e. an indication whether any argument has been satisfied or a modifier found (**Status** is set to 1; otherwise it is left uninstantiated).

In the parsing of core nps we include an **`ifthenelse`**: if the **Status** variable returned by **`satisfylist`** is **uninstantiated**, we assign 2 as weight, otherwise we assign 3. Here is the relevant piece of code (the case envisaged here is that of an np which does not include adjective pre- or postmodification and **`satisfylist`** therefore applies to the head noun's **arglist**):

```
corenounphrase(P0,P3,Gap,index(I),Pref,Weight,Rel,Intrel,
        Function,
              [Function,[nounphrase,index(I),
                   agr(third,Number),DET,N,Parse]],
             Number,third,Sem):-
   determiner(P0,P1,DET,Number,Rel1,Intrel1) ,
   xnoun(P1,P2,N,Number,Sem,Arglist),
   satisfylist(P2,P3,Gap,Status,Pref,Prec,Rel2,Intrel2,_,
           Parse,np,Arglist,Func,
           subject([subject,[nounphrase,index(I),
                 agr(third,Number),DET,N]],
         Verbsem)),
   sfok(Verbsem,Sem),
   ( var(Rel1),var(Intrel1),Rel=Rel2,Intrel=Intrel2;
    nonvar(Rel1),nonvar(Intrel1),Rel=Rel1,Intrel=Intrel1 ),
   ifthenelse(var(Status),Weight=2,Weight=3).
```

The **`ifthenelse(Condition, Yescase, Nocase)`** predicate is an Arity Prolog extension to standard Edinburgh Prolog; if missing in a Prolog implementation, it can readily be simulated by the following disjunction:

```
( var(Status), Weight = 2; nonvar(Status), Weight = 3 )
```

The **weight** value is passed to the **posprec** functor and exploited by the `precede` relation to determine the relative order of arguments and modifiers. See the relevant section, p. 91.

Coordinated nps present two problems:

1) computing the weight of the resulting coordination: heuristically it has been decided to add the weights of the constituent nps, divide the result by 2 and add 1.

2) determining the person of the resulting np: this question is discussed on p. 35.

Np gaps are discussed on p. 99.

Most of the information necessary for the parsing of **pronouns** is simply read off the lexicon. Of course, the **function** played by the pronoun cannot be determined ahead of parsing, and it is not read off the lexicon, although unification provides a check on the possible function value stored in the lexicon and the one assigned by the parsing process.

Let us take personal pronouns first. The features copied from the lexicon are highlighted:

```
nounphrase(P0,P1,[],index(I),0,1,Rel,Intrel,
           Function,
           [Function,
     [nounphrase,index(I),agr(Person,Number),Tree]],
            Number,Person,Sem):-
   pp(P0,P1,Tree,Person,Number,Gender,Function,Sem).
```

Note that **Gender** would also have to be percolated to the **noun phrase level** if we wanted to deal with **reflexive** pronouns. In the parsing of

She behaved herself.

we would have to check that gender is the same in the subject and the reflexive object.

Relative and **interrogative** pronouns have two arguments that we need to go into a little, namely the seventh and the eighth.

Relative pronouns use the **7th** position to store the **semantic value** of the pronoun. If no relative is present in a noun phrase, the 7th position of the `nounphrase` predicate will be left **uninstantiated**. We cannot therefore use an uninstantiated variable to mean that the relative does not set any semantic restriction, and we use the value **norestriction** instead. The predicate performing the semantic check (**`sfok`**) knows how to deal with that value (see p. 87).

The code for noun phrases consisting of a relative pronoun is the following:

```
nounphrase(P0,P1,[],index(I),
        0,1,Semnp,rel(I),Function,
        [Function,
     [nounphrase,index(I),agr(third,Number),Tree]],
        Number,third,[Semnp]):-
relative(P0,P1,Tree,Semnp,Sempp,np).
```

Note that the **Semnp** feature is inserted in a list when passed to the noun phrase level, because at that level np semantics must be a list, as noun semantics is a list.

The `relative` predicate is a lexical one; the last three arguments are as follows:

Semnp: semantic features associated with the relative (which has np value); the string **norestriction** is used to indicate that no semantic feature is assigned, e.g.

relative([who|X],X,[relative(who)]**,human**,_,np).

vs

relative([that|X],X,[relative(that)],**norestriction**,_,np).

Sempp: semantic features associated with the relative (which has pp value), e.g.

relative([why|X],X,[relative(why)],norestriction,**[reason]**,pp).

The last argument stipulates whether the relative pronoun stands for an np or an adverbial (**np** and **pp** values).

The **eighth** position is used to keep **relative** and **interrogative** pronouns apart. In the case of **interrogatives**, it is set to the string value **int**. In the case of relatives it is used to enable coindexing of the relative pronoun when the latter does not constitute the whole np but is embedded somewhere in it, as in *a book about whom*. In such a case we need an index for the whole np and one for relative *whom*. The latter is provided by the **rel(I)** functor, where **I** is meant to carry an index.

Here is the code for interrogative pronouns:

```
nounphrase(P0,P1,[],index(I),
        0,1,Semnp,int,Function,
        [Function,
    [nounphrase,index(I),agr(third,Number),Tree]],
        Number,third,[Semnp]):-
 interrogative(P0,P1,Tree,Semnp,Sempp,np).
```

The parsing of nps containing a **relative clause** must ensure that at the higher level the **seventh** and **eighth** positions are left **uninstantiated**, and that the relative clause (in particular, the gap within it) and the antecedent share what they must share. Note that the **function** is NOT shared between the antecedent and the gap in the relative clause: we can have an antecedent **subject** coindexed with an **object** gap in the relative clause, as in *The teacher* (subject) *you like* (object gap) *teaches mathematics*. The code is the following:

```
/* Weight is 3, i.e. the resulting np is heavy */
  nounphrase(P0,P2,[],index(I),
      Preftop,3,_,_,Function,
      [Function,Newrest],
      Number,third,Sem):-
 corenounphrase(P0,P1,[],index(I),
                Pref,Weight,Rel,Intrel,Function,
                [Function,Rest],
                Number,third,NounSem),
 ( var(Intrel); Intrel \= int ),      /* not an interrogative */

 var(Rel),
/* the antecedent itself must feature a free Rel variable */

 xrelclause(P1,P2,Prefrel,RELCL,
             [gap(_,_,Npsem,_,_,_,index(I),_)],
             Personinrel,Funcinrel,NounSem,Number),
 sfok(Npsem,NounSem),
 append(Rest,[RELCL],Newrest),
 accu(Preftop,[Pref,Prefrel]).
```

4.3.1.2. Generation

Generating nps is considerably simpler than parsing them. We consider three cases. The first is that of **np gaps** and of nps that have been **ghosted** (deprived of lexical material) by the generator in its treatment of **control relations** (see p. 41). Such nps are represented by a parse tree where the variable named **NP** in the following piece of code is an **uninstantiated** variable. The generator produces an empty list, which will disappear in the list appending and flattening operations at the end of the generation process.

```
gen([Function,[nounphrase,index(I),agr(P,N)|NP]],[]):-  var(NP),!.
```

The second case is that of **personal pronouns**. They are retrieved form the lexicon on the basis of their **person**, **number**, **gender** and **function**. Note that these values appear both inside and outside the **ppro** functor. We give below the rule in the generator and the lexical clause for **object** *her*:

```
gen([Function,[nounphrase,index(I),
     agr(Person,Number),ppro(Person,Number,Gender)]],PPform):-pp([PPform|
_],_,ppro(Person,Number,Gender),Person,Number,Gender,Function,_).

pp([her|X],X,ppro(third,sing,fem),third,sing,fem,object,[human]).
```

The third case is the default case. We skip the information pieces that precede the lexical material in the parse tree, and generate from that material:

```
gen([Function,[nounphrase,index(I),agr(P,N)|NP]],Gennp):-
   NP \= ppro(_,_,_),
   gen(NP,Gennp).
```

The call on **gen** will unify with its defining clause for generating the various elements of a list:

```
gen([H|T],[Hgen|Tgen]):-
    gen(H,Hgen),
    genlist(T,Tgen).

genlist([],[]).

genlist([H|T],[Hgen|Tgen]):-
       gen(H,Hgen),
       genlist(T,Tgen).
```

Gen will then generate the determiner (if any), adjective (if any), the noun and its arguments and modifiers (if any). To do so, it will call on the **gen** clauses for **leaves**, such as the following for the **determiner**:

```
gen([det(Det),index(I)],Det).
gen([det(zero)],[]).
gen([det(Det)],Det):- Det \= zero.
```

4.3.2. Adjective Phrases

An adjective modifying a noun generally precedes that noun; however, if the adjective has an **arglist** that is partly or totally satisfied, the adjective follows the noun. Contrast:

Each participant was asked an easy question.
* Each participant was asked a question easy.
A question easy for you to answer is not necessarily an easy question.
* An easy for you to answer question[12]

In **horatio** we use the **Status** variable returned by **satisfylist** to check whether the **arglist** of the adjective has been partly or totally satisfied. If the adjective precedes the noun, the adjective's arglist must be unsatisfied (the noun's arglist can of course be satisfied: *an easy book on linguistics*). If the adjective follows, the noun's arglist cannot be satisfied, and the adjective's must be (* *a book easy for you to read on linguistics*; * *a book easy on linguistics*).

Here is the code for the case where the adjective precedes the noun it modifies :

```
corenounphrase(P0,P4,Gap,index(I),Pref,Weight,Rel,Intrel,Function,
                      [Function,
                  [nounphrase,index(I),agr(third,Number),
                                          DET,ADJ,N,Parse]],
                    Number,third,Sem):-
   determiner(P0,P1,DET,Number,Rel1,Intrel1) ,
   xadjphrase(P1,P2,[],subject(SUBJ,Semsubjadj),
          Prefadj,_,modifier,Statadj,
          ADJ,Semadj),
   var(Statadj),
   xnoun(P2,P3,N,Number,Sem,Arglist),
   sfok(Semadj,Sem),
   satisfylist(P3,P4,Gap,Status,Pref,Prec,Rel2,Intrel2,_,
          Parse,np,Arglist,Func,
          subject([subject,[nounphrase,index(I),agr(third,Number),
               DET,ADJ,N]],
          Vpsem)),
    sfok(Vpsem,Sem),
    ( var(Rel1),var(Intrel1),Rel=Rel2,Intrel=Intrel2;
    nonvar(Rel1),nonvar(Intrel1),Rel=Rel1,Intrel=Intrel1 ),
   ifthenelse(var(Status),Weight=2,Weight=3).
```

Phrases such as *an easy man to please* cannot be parsed by the mechanisms explained above: the adjective's **arglist** is separated from the adjective by the noun. In order to parse them, we need to isolate the adjective classes for which this construction is possible. In **horatio** such adjectives are recognized by looking at their **arglist**, which is **[to_vp(oblig,object)]**, i.e. the noun modified by the adjective is the object of the infinitive in the **to-phrase**. The code is the following:

```
corenounphrase(P0,P4,[],index(I),Pref,3,Rel,Intrel,Function,
                 [Function,[nounphrase,index(I),agr(third,Number),
                    DET,ADJ,N,Parse]],
                 Number,third,Sem):-
    determiner(P0,P1,DET,Number,Rel1,Intrel1) ,
    adj(P1,P2,ADJ,_,[to_vp(oblig,object)]),
    xnoun(P2,P3,N,Number,Sem,Arglist),
    satisfy(P3,P4,[],Pref,Prec,Rel2,Intrel2,
           Parse,to_vp(oblig,object),
           subject([_,[nounphrase,index(I),agr(third,Number),
                 DET,ADJ,N]],Semsubj),
           subject([_,[nounphrase,index(I),agr(third,Number),
                 DET,ADJ,N]],Semsubj)),
    ( var(Rel1),var(Intrel1),Rel=Rel2,Intrel=Intrel2;
     nonvar(Rel1),nonvar(Intrel1),Rel=Rel1,Intrel=Intrel1 ).
```

Note that the subject functor passed to the `satisfy` predicate will **not** be used for subject control of the **to-phrase**; in the absence of a **FOR NP** the subject of the infinitive is not **syntactically** retrievable. The structure passed in the subject functor is used to **coindex** the gap that the to-phrase needs to display, namely an **object** gap (object is not to be taken strictly here; it simply means an **oblique** function). The defining clause for the `satisfy` predicate as used here is the following:

```
satisfy([to|P0],P1,[],Prefgen,Posprec,Rel,Intrel,
              [adj_arg,Tree],
              to_vp(Type,object),
              subject(SUBJ,Semsubj),
              subject(SUBJ,Semsubj)):-

  SUBJ=[X,[Y,index(I)|Remainder]],
/* the "subject" of the adj (index(I)) is an object gap in the vp */

  xverbphrase(P0,P1,
             subject(NP,Sem), /* no subject control */
    [gap(Object,npgap,Semsubj,_,Func,_,index(I),_)],
    Pref,Tree,Rel,
    Intrel,toinfinitive,Tense,
    aspect(Aspect),Modality,
    Number,Person,Voice,nsubject(NNP,Nsem)),
    nonvar(Func),
    ( Func=object; Func=indirect_object; Func=np_arg_of_prep ),

/* these are the possible functions for the gap:
     hard to see; hard to give a book to; hard to borrow from */

    Prefgen is Pref + 3.
```

The code for the parsing of an adjective phrase does not require a lot of explanations:

```
adjphrase(P0,P2,Gap,subject(SUBJ,Sem),Pref,3,
          Function,Status,
              [Function,[adjectivephrase,Adj,Parse]],Semadj):-
  adj(P0,P1,Adj,Semadj,Arglist),
  satisfylist(P1,P2,Gap,Status,Pref,Prec,Rel2,_,_,
              Parse,_,Arglist,Func,
              subject(SUBJ,Semadj)).
```

Note only that we need to have the **subject** functor for cases such as *reluctant to go*, where we need the subject (either the modified noun or the subject in predicative constructions) to process the **TO** phrase.

4.3.3. Prepositional Phrases

Prepositional phrases can appear as arguments of nouns. Nouns may then determine either the semantics of the whole prepositional phrase, or the exact preposition to be used. The noun **HYPOTHESIS** is of the first type; it specifies that its optional argument pp must be a topic. We have the following clause in the lexicon:

```
m_noun(hypothesis_1,hypothesis,[abstract],[pp(opt,n:topic),s(opt)]).
```

The noun **trouble** is of the second type. It specifies that its optional argument pp must be headed by the preposition *with*:

```
m_noun(trouble_1,trouble,[abstract],[pp(opt,prep:with)]).
```

Accordingly, prepositional phrases in the grammar must have these two specifications at the highest level. A number of other pieces of information are also needed, among which **two** indices, one for the whole pp and one for the np inside. We need to be able to coindex the np inside the pp with an antecedent:

The man about whom I have read a book is a teacher.

But we also need an index for the whole pp, to be referred to in the gapped argument of *book*.

We use the following code to parse "normal" (i.e. neither gapped nor coordinated) pps:

```
prepphrase(P0,P2,Gap,index(J),
        npindex(I),Prefnp,3,PREP,Rel,Intrel,Function,
            [Function,[prepphrase,index(J),prep(PREP),NP]],
            Sempp,Semnp):-
   prep(P0,P1,[prep(PREP)],Sempp) ,
   nounphrase(P1,P2,Gap,index(I),Prefnp,Precnp,Rel,
          Intrel,np_arg_of_prep,NP,Number,Person,Semnp).
```

We note that the semantics of the **embedded np** is also percolated to **pp** level. It is needed in the parsing of **by-phrases**, where it must be checked that the **np** within the **pp** has the appropriate semantics for the subject role. The check is performed in the following piece of code:

```
satisfy(P0,P1,Gap,Pref,posprec(_,3),Rel,Intrel,
          [subject,[nounphrase,index(I)|X]],
          byphrase(opt,posprec(_,3),subject,Semsubj),
          subject(SUBJ,Sem),subject(SUBJ,Sem)):-
  xprepphrase(P0,P1,Gap,index(J),npindex(I),
           Prefpp,Precpp,by,Rel,Intrel,
             subject,
       [subject,[prepphrase,index(J),prep(by),
             [np_arg_of_prep,[nounphrase,index(I)|X]]]],
          PPseminpp,Semsubjinpp),
     sfok(Semsubj,Semsubjinpp),
      Pref is Prefpp + 4.
```

Gapped pps are discussed on page 99 and following.

We distinguish two cases of **coordinated pps**. In the first one two **full pps** are coordinated, as in *with the teacher and for the teacher*. In the second, the first pp is reduced to a preposition, the np being gapped, as in *with and for the teacher*. An alternative solution would be to parse a coordinated preposition, rather than whole pp, in the second case.

The first case is dealt with by the following code:

```
xprepphrase(P0,P2,[],index(I),npindex(I2),
         Pref,Weight,PREP,Rel,Intrel,Function,
         [Function,[and_prepphrase,PP1,PP2]],
         Sempp,Semnp):-
 inlist(and,P0),
 prepphrase(P0,[and|P1],[],index(I),
      npindex(I2),Pref1,Weight1,PREP,
      Rel,Intrel,Function,
      PP1,Sempp,Semnp),
 xprepphrase(P1,P2,[],index(J),
          npindex(J2),Pref2,Weight2,Prep2,
      Rel,Intrel,Function,
      PP2,Sempp2,Semnp2),
 accu(Pref,[Pref1,Pref2]),
 accu(Weightaccu,[Weight1,Weight2]),
 Weight is (Weightaccu/2) + 1.
```

Note that the two pps must share the **Rel** and **Function** values: if one contains a relative, so must the other: *about whom and for whom*; * *about whom and for the teacher*.

The second case is taken care of by the following code. Note that we must look into the returned parse trees to restore the missing np to the first conjunct:

```
xprepphrase(P0,P2,[],index(I),npindex(I2),
            Pref,Weight,PREP,
      Rel,Intrel,Function,
         [Function,[and_prepphrase,
               [prepphrase,index(J),prep(PREP),NP],
               [prepphrase,index(I),prep(Prep2),NP]
         ]],
         Sempp,Semnp):-
 inlist(and,P0),
 prep(P0,[and|P1],[prep(PREP)],Sempp),
 prepphrase(P1,P2,[],index(I),npindex(I2),
      Pref,Weight,Prep2,
      Rel,Intrel,Function,
      [Function,[prepphrase,index(I),prep(Prep2),NP]],
      Sempp2,Semnp).
```

4.3.4. Verb Phrases

4.3.4.1. Parsing

4.3.4.1.1. The Auxiliary Group

We first discuss the treatment of the **auxiliary group**. We need to take into account the well-known **precedence relations** within the group. To do so, we assign two features to each auxiliary, one being its type (**finite**, **ing**, **en-passive**, etc.), the other the type that it requires the next auxiliary (or main verb) to be. In the lexicon, we therefore have such clauses as:

aux([may|X],X,may,**finite**,**infinitive**,_,_,present).
aux([is|X],X,_,**finite**,**ing**,sing,thirdsg,present).
aux([is|X],X,_,**finite**,**en_passive**,sing,thirdsg,present).
aux([have|X],X,_,**infinitive**,**en_active**,_,_,_).

The fourth argument of the lexical predicate **aux** is the **Type** of the auxiliary, the fifth one is the **Required** feature, i.e. the type of the next verb form to its right. The third clause in the sample applies to BE as **passive** auxiliary, the fourth to HAVE as **perfect** auxiliary.

We also need to specify what each auxiliary contributes to the **property** field of the clause. Such a property field has the following form:

prop(vce:Voice, asp:Aspect, mod:Modality,tns:Tense),

where **vce**, **asp**, **mod** and **tns** are markers, and the capitalized items are variables. **Voice** ranges over two values: **passive** and **uninstantiated** (represented by an uninstantiated variable). **Tense** ranges over **past** and **present**, and is set by the first element of the verb group, be it an auxiliary or the main verb. **Aspect** is a feature **list**, whose possible members are **progressive** and **perfect**. **Modality** is also a feature **list**, whose possible members are the various modal auxiliaries (**shall**, **will**, **may**, **can**, etc.; the treatment is admittedly very surfacy). We also make use of the special value **none** for Modality and Aspect; it is equivalent to an empty feature list.

We define the predicate **`checkaux3`** to specify the contribution made by each auxiliary to the **aspect** and **voice** fields. The first argument is the **aspect feature list**, the second the **Required** feature of the auxiliary, and the third the value for **Voice**. The code is the following:

```
checkaux3([perfect],en_active,_).
checkaux3([progressive],ing,_).
checkaux3([],Required,_):-
                    Required \= en_passive,
                    Required \= en_active,
                    Required \= ing.
checkaux3([],en_passive,passive).
```

Note that the value for Voice is the **anonymous** variable when Voice is not passive, i.e. active.

Since we do not know the number of auxiliaries that a given verb phrase will have, we define the **`verbphrase`** predicate as recursive: it calls the **`xverbphrase`** predicate (recall that the **x** prefix indicates that the predicate can be coordinated). We call a special version of the **`append`** predicate (**`myappend`**) to build up the **aspect** value. **`Myappend`** is able to deal with the **none** value. Here is the relevant piece of code:

```
verbphrase(P0,P2,subject(SUBJ,Semsubj),Gap,Preflist,
         VG,
         Rel,Intrel,Type,Tense,
         aspect(A1),Modality,
         Number,Person,Voice,
         nsubject(NSUBJ,Nsem)):-
     xaux(P0,P1,Modality,Type,Required,Number,Person,Tense),
     checkaux3(A2,Required,Voice),
     myappend(A1,A2,A),
     xverbphrase(P1,P2,subject(SUBJ,Semsubj),Gap,Preflist,
            VG,Rel,Intrel,
            Required,Tense,aspect(A),Modality,
            Number,Person,Voice,
            nsubject(NSUBJ,Nsem)).
```

`Myappend` is defined by the following clauses:

/* the value **none** - either isolated or as first element of a one-element list - receives the same treatment as the **empty list** */

```
myappend([],X,X).
myappend(X,none,X):-!.
myappend(X,[none],X):-!.
myappend(X,Y,Y):-   var(X),  !.
myappend([Head|L1],L2,[Head|L3]):- myappend(L1,L2,L3).
```

4.3.4.1.2. Coordinated Verb Phrases

In coordinated verb phrases we make sure that the **arglist** is appropriate to both conjuncts. For instance, in

He looked at and liked the girl

the phrase *the girl* is the arglist of both **LOOK AT** and **LIKE**.

The check is implemented by feeding the **arglist** predicate the same word list. In our example, *the girl* is parsed as **arglist** of LOOK AT and then reparsed as **arglist** of LIKE. An additional problem arises out of the presence of the preposition or particle, which remains close to its verb, instead of joining the **arglist**. Indeed, we do not have

* *He looked and liked at the girl*

We must therefore make sure that the particle is found to the immediate right of the first coordinated verb. The following piece of code takes care of the example:

```
/* 1  he looked at and liked the girl */
verbphrase(P0,P4,subject(SUBJ,Semsubj2),Gap,Pref,
         [pred_arg_mod_structure,
          prop(vce:V,asp:A,mod:Modality,tns:Tense),
          [and_predicate,VERB1,VERB2],SParse],
          Rel,Intrel,Type,Tense,aspect(Aspect),Modality,
          Number,Person,Voice,nsubject(NSUBJ,Nsem2)):-
   verb(P0,P1,Class1,VERB1,Type,Tense,Number,
        Person,Semsubj1,Args1) ,
   Args1 = [string(X,Y,Z)|Rest],
/* X is Status, Y is posprec and Z is the string itself in list format, e.g. [away,with] */
   append(Z,[and|P2],P1),
/* the string + and + the remainder is what follows the verb */
   verb(P2,P3,Class2,VERB2,Type,Tense2,Number,Person,Semsubj2,Args2),
   arglist(P3,P4,Gap,Status1,Pref1,Preclist1,Rel,
           Intrel,Voice,Parse1,vp,
           Rest,Func,subject(SUBJ,Semsubj1),
           nsubject(NSUBJ,Nsem1),
           Class1),
   arglist(P3,P4,Gap,Status2,Pref,Preclist,Rel,
           Intrel,Voice,Parse2,vp,
            Args2,Func,subject(SUBJ,Semsubj2),
            nsubject(NSUBJ,Nsem2),
            Class2),
/* note P3,P4 twice: the same list must satisfy both the remaining args of the first verb and those of the
second */
    (nonvar(Aspect); var(Aspect),A = none ),
    (Aspect = [] , A = none; Aspect \= [],A = Aspect ),
    (nonvar(Modality); var(Modality),Modality = none ),
    (nonvar(Tense); var(Tense), Tense = present ),
    (nonvar(Voice), V = Voice; var(Voice),V = active ),
    append([NSUBJ],Parse2,AParse2),
    insort(AParse2,SParse),  /* to get canonical word order */
    Status1 = Status2. /* the two Status values must unify */
```

Note that the **verb class** (**Class1** and **Class2**) is passed as an argument to **arglist**, but cannot be used as the basis on which to determine whether the two verbs can be coordinated: we need to parse the remaining word list. It would be counter-productive to have to assign the same word class to two verbs merely on the basis of the observation that the two verbs can be coordinated. In our example we do not want to assign LOOK AT and LIKE to the same word class, as this would go against the spirit of consistency checks and template sharing that we might want to place on the lexicon.

4.3.4.2. Generation

Generating the verbal group divides naturally into two tasks: generation of the auxiliary group (whose membership may be nil) and generation of the main verb.

4.3.4.2.1. The Auxiliary Group

We shall first consider the generation of the auxiliary group. The **gen** predicate dedicated to this task needs to have access to the **Property** field (**voice**, **aspect**, **modality** and **tense**), and to **agreement values** of the subject, i.e. **Person** (NP1 in the code cited below) and **Number** (NP2). It returns a **list** of auxiliaries.

This **gen** predicate must also be sensitive to the required auxiliary order in the clause: **modality**, aspect (**perfect**), aspect (**progressive**) and **voice**, as in:

*He **may have been being** interviewed.*

Once we have generated a candidate auxiliary list, we check that the first auxiliary agrees in person and number with the subject. If it does not, we have to backtrack, but we can skip the part between **snips** (**[! !]**) (snips are a useful Arity Prolog extension to standard Prolog: the predicates between the two snips are skipped on backtracking[13]). Here attempting to redo the list **appending** would be useless, and attempting to redo the list **flattening** would lead to trouble.

The relevant piece of code is the following:

```
gen(prop(vce:Voice,asp:Asp,mod:Mod,tns:Tns),NP1,NP2,Alist2):-

gen(Mod,Tns,Modaux,Req1),           /* Modality */
genasp1(Asp,Aspaux1,Req1,Req2),       /* Aspect: perfect */
genasp2(Asp,Aspaux2,Req2,Req3),       /* Aspect: progressive */
gen(Voice,Voiceaux,Req3,Req4),        /* Voice */
/* the Required feature of the preceding Aux is found in last position but one in the next, i.e. the one
housing its Type */
```

```
[! append([Modaux],[Aspaux1],List1),
append([Aspaux2],[Voiceaux],List2),
append(List1,List2,List3),
flatten(List3,Auxlist),
first(Auxlist,Auxfirst) !] ,
getagr(Auxfirst,Auxfirstagree,Tns,To),
/* the last argument indicates whether a TO is needed in front of the auxiliary list */
[! append(To,Auxlist,Alist),
flatten(Alist,Alist2) !] ,
agree(NP1,NP2,Auxfirstagree).
```

Various predicates need to be explained. Let us begin with the ones generating the auxiliaries.

Modal auxiliaries are very easy to generate. We call the **`gen`** predicate with four arguments: the first is the **parse tree** of the auxiliary (it reduces to a simple string indicating the **lexemic** value of the auxiliary: both *might* and *may* will yield **may**, and differ as to tense); the second is **tense**, the third will house the **generated string** and the fourth is the **type** required. In the case of modal auxiliaries, this value is always **finite**.

We need to cater for the value **none**, which generates the empty list:

```
gen(none,_,[],_).
```

If the parse tree is not **none**, we get the appropriate form from the lexicon:

```
gen(Mod,Tns,Modaux,Req1):-
            aux([Modaux|_],_,Mod,_,Req1,_,_,Tns).
```

We have two predicates for the generation of **aspectual** auxiliaries, **`genasp1`** and **`genasp2`**. This is due to the fact that the **perfect** auxiliary (HAVE) needs to be generated before the **progressive** auxiliary (BE). Of course, neither need be present.

`Genasp1` looks for the value **perfect** in the **aspect value list**. It may be the only value in the field (there is no progressive), or it may be the first of two, or the second of two. We therefore have three clauses, which we could easily reduce to one by testing if **perfect** is a member of the list passed as first argument. It is this new version of the predicate that we give here:

```
genasp1(Asplist,Auxform,Req1,en_active):-
    inlist(perfect,Asplist),
    aux([Auxform|X],X,have,Req1,en_active,_,_,_).
```

Note that **Req1** points to the type that aspectual **have** is required to have (for instance, infinitive after a modal auxiliary), and that **en_active** is the type it requires of the next auxiliary in the chain, and will therefore appear in the preceding slot in the next call in the generation of the auxiliary list, as in this fragment:

```
genasp1(Asp,Aspaux1,Req1,Req2),  /* Aspect: perfect */
genasp2(Asp,Aspaux2,Req2,Req3),  /* Aspect: progressive */
```

We also need a clause for the case where there is no perfect auxiliary:

```
genasp1(_,[],Req,Req).
```

Note the anonymous variable in the first slot, which makes it clear that rule ordering is relevant here.

The generation of the progressive auxiliary runs parallel, and will not be discussed here.

Generating the voice auxiliary is an easy task. If voice is active, we do not generate anything. If it is passive, we look in the lexicon for the appropriate form of BE:

```
gen(active,[],Req,Req).

gen(passive,Auxform,Req1,en_passive):-
       aux([Auxform|X],X,be,Req1,en_passive,_,_,_).
```

The next predicate that deserves our attention is `getagr`, whose job is to get the agreement feature of the **first** auxiliary in the auxiliary list, since this is the only auxiliary that needs to agree with the subject, being the only finite one.

We must bear in mind that we have used the **tense** feature to indicate restriction on the whole verb group in dealing with **raising** and **control**, using it to indicate for instance that the whole group must be an **infinitive** or **ing** group. For instance, consider the following definition of `oraising`, the predicate taking care of subject to object raising:

```
oraising([H1,[pred_arg_mod_structure,Prop1,
        [predicate(Pred1,AgrPred1)],
    [[object,[clause,
[pred_arg_mod_structure,Prop2,[predicate(P,agr(Agr))],
                  [[Subject|Rest]|Otherargs]]]]|R1]]],
[H1,[pred_arg_mod_structure,Prop1,
       [predicate(Pred1,AgrPred1)],
    [[object|Rest],
     [object,[clause,
[pred_arg_mod_structure,Propnew2,[predicate(P,agr(Agr))],
                  [Otherargs]]]]|R1]]]):-

allsubject(Subject),
second_header(H1) ,
oraise(Pred1,Requires),
nonfinite(Agr),
Prop2 = prop(Voice,Aspect,Mod,Tns),
Propnew2 = prop(Voice,Aspect,Mod,tns:Requires).
```

The `oraise` predicate makes a direct call on the lexicon, as we shall see. The call `oraise(Pred1,Requires)` checks that **Pred1** is a subject-to-object raising predicate, but it also determines the nature of the verbal group left after the subject of the lower clause has been promoted to object of the higher one.

The code for `oraise` is the following:

```
oraise(Oraisingverb,to):-
m_verb(vinf,_,Oraisingverb,_,_,_,_,_,_,_,_,_,_,
       [np(_,_,surf_object,_),
        np_vp(oblig,to_inf,object)]).

oraise(Oraisingverb,to):-
m_verb(vinf,_,Oraisingverb,_,_,_,_,_,_,_,_,_,_,
       [np(_,_,subject_inf,_),
        np_vp(oblig,to_inf,object)]).

oraise(Oraisingverb,ing):-
```

```
m_verb(ving,_,Oraisingverb,_,_,_,_,_,_,_,_,_,_,
       [np(_,_,subject_ing,_),
        np_vp(oblig,ing,object)]).
```

From this we see that the second argument of **`oraise`** houses a property of the remaining non-finite clause after the promotion of its subject to the higher clause: it is either **infinitive** (bare infinitive or infinitive preceded by TO) or **gerundive**.

In the code for **`oraising`**, we see that this property **overwrites** the tense value in **Prop2** to yield **Propnew2**, the property list associated with the lower clause. In dealing with the agreement feature of the first auxiliary in the auxiliary list, we also take into account the restriction that the tense feature slot can have been made to bear.

Tense occupies the third slot of **getagr**. The first is the auxiliary and the second the agreement property of the auxiliary (the one that needs to concord with the **Person** and **Number** values percolated from the subject; it is read off the lexicon). The fourth slot is a list, which will either be left empty or will house the infinitive particle TO.

Of course, it may be the case that there is no first auxiliary, simply because the auxiliary list is empty. In that case, **`getagr`** returns **noaux** as auxiliary agreement feature. The **agree** predicate will have to be able to deal with this value.

Here is the code for **`getagr`**:

```
/* there is an aux */

getagr(Aux,Auxagree,Tns,[]):-
      Tns \== to,
      Tns \== bare,
      Tns \== ing,
      aux([Aux|X],X,_,finite,_,_,Auxagree,Tns).

getagr(Aux,Auxagree,to,[to]):-
      aux([Aux|X],X,_,infinitive,_,_,Auxagree,_).

getagr(Aux,Auxagree,bare,[]):-
      aux([Aux|X],X,_,infinitive,_,_,Auxagree,_).

getagr(Aux,Auxagree,ing,[]):-
      aux([Aux|X],X,_,ing,_,_,Auxagree,_).

/* there is no aux */

getagr([],noaux,Tns,[]):-
             Tns \== to,
      Tns \== bare,
      Tns \== ing,!.

getagr([],noaux,to,[to]).
getagr([],noaux,bare,[]).
getagr([],noaux,ing,[]).
```

Note that if **Tns** really houses a **tense value**, the auxiliary needs to be **finite**. Note also that we do not really need four clauses when there is no auxiliary (two would do), but we have chosen to preserve code parallelism.

The code for **`agree`** is similar to the one used in analysis, but we need a clause for **noaux**, which we place at the end of the clause packet, and which succeeds no matter how the first two arguments (percolated from the subject) are instantiated:

```
/* agree(Personnp,Number,Personvp) */

agree(first,sing,firstsg).
agree(first,plural,other).
agree(second,sing,other).
agree(second,plural,other).
agree(third,sing,thirdsg).
agree(third,plural,other).
agree(_,_,noaux).
```

In **YES-NO questions** we use a special predicate to generate the auxiliary list (**`genyesno`**). The reason is that a form of DO needs to be generated in YES-NO questions when voice is active and there is no modal or aspectual auxiliary (and recall that what follows a non-subject WH-group is also generated as a YES-NO question). We call on the lexicon to provide us with a finite form of DO and then call on **`agree`** to check whether it is in agreement with the subject:

```
genyesno(prop(vce:active,asp:none,mod:none,tns:Tns),
         Person,Number,[Aux|[]]):-
   aux([Aux|_],_,do,finite,_,_,Agraux,Tns),
   agree(Person,Number,Agraux),!.
```

Note that the form of DO is returned as first element of a list (with empty tail), in conformity with the pattern used in the generation of auxiliaries in declarative clauses. If we do not need to generate a form of DO, **`genyesno`** can simply fall back on the version of the **`gen`** predicate as used in declarative clauses:

```
genyesno(prop(vce:Voice,asp:Asp,mod:Mod,tns:Tns),Person,Number,Auxlist):-
        gen(prop(vce:Voice,asp:Asp,mod:Mod,tns:Tns),Person,Number,Auxlist).
```

4.3.4.2.2. The Main Verb

The generation of the last element of the verb group, i.e. the **main** verb, is in most cases simply a matter of retrieving the verb form on the basis of the lexeme value and the agreement functor. However, we also need to call on the macro **`m_verb`** clause to retrieve the particle that should accompany the verb, since it is amalgamated to the lexeme in the parse tree returned by **horatio**. For instance, *He looked it up* will have a parse tree where *up* is no longer an independent element, but appears only in the lexeme value, which will be **look_up_1**, i.e. the first (and at present only) reading of a lexeme **LOOK UP**. It would also be possible to retrieve the particle by a string operation on the lexeme value in the parse tree, but this is felt to be too tightly tied to the string structure of the lexeme value to be a reasonable way of getting at the particle. We prefer to have it as a separate argument in the lexicon, even if it will often be left uninstantiated.

The default clause for the generation of the main verb runs as follows:

```
gen([predicate(Lex,Agr)],Vform,P):-
verb([Vform|_],_,_,[predicate(Lex,Agr)],_,_,_,_,_,_),
m_verb(Class,Part,Lex,_,_,_,_,_,_,_,_,_,_,_),
( var(Part),P=[]; nonvar(Part),P=[Part] ).
```

If **Part** is left uninstantiated in the lexicon, we return an empty list as particle value (last argument of the **`gen`** predicate); otherwise, we return the particle. As a matter of fact, the particle as read off the lexicon is more than the string value of the particle. It is a couple made of a **particle type** and a **string value**. And **particle** is to be understood in an extended sense, as covering various types of fixed strings associated with verbs. Consider the **`m_verb`** clause for **LOOK UP**:

```
m_verb(vphr,part1:up,look_up_1,look,look,look,looks,looking,
        looked,looked,looked,trans,human,
        [part(oblig,posprec(1,2),up),
         np(oblig,posprec(1,Wnp),object,abstract)]).

/* she looked it up */
```

In the couple **part1:up**, the first element, the particle type, is used to determine the position of the particle with respect to other elements, such as the verb's arguments. This is accomplished in the standard way, by associating a certain weight with particles and other constituents.

The relevant code is that of the **`poids`** predicate (**poids** is French for weight):

```
poids([part0:_],0).
poids([part1:_],2).

poids([F1,[nounphrase,_,_,ppro(_,_,_)]],1):-
       ( F1 = object; F1 = indirect_object; F1 = indirect_object_2).
/* personal pronouns filling these functions must stay close to the verb */

poids([F1,R1],W):- assoc(F1,W).
/* general case: we read the value in the assoc table */
```

Part0 particles are given the lightest possible weight, and so will have to remain attached to the verb. For instance, we will assign **part0** as particle type to the string *place* in the mwu **take place**. We have no problem regarding *place* as a particle, in an extended sense at least, where the word **particle** refers to fixed string elements. In the section on mwu's (see page 21) we argued that *place* in **take place** should not be considered an np or even a noun, but should be regarded as a string, from which viewpoint its resistance to morphological variation and syntactic manipulation is readily explained.

Part1 particles are given a weight of 2, and can therefore follow very light nps such as personal pronouns, which receive a weight of 1. This explains why we generate *He looked it up*, and not * *He looked up it*.

The **`assoc`** table will be further discussed on p. 80. The generation version differs slightly from the analysis version. Here is the code for generation:

```
assoc(subject,1).
assoc(subject_pass,1).
assoc(subject_inf,1).
assoc(object,3).
assoc(indirect_object,2).
assoc(indirect_object_2,2).
assoc(cplt_s,6).
assoc(subject_attribute,4).
assoc(object_attribute,4).
assoc(pp_arg,3).
assoc(vp_modifier,5).
```

Two special cases precede the default case in main verb generation. They deal with the TO that must **follow bareinf** verbs when used in the passive voice (*They saw him take it* vs. *He was seen **to** take it*), and with the TO that must **precede** the verb when its agreement functor registers the need for it (value

toinfinitive). The code for these cases runs as follows:

```
gen([predicate(Lex,agr(en_passive))],Genpred,P):-
verb([Vform|_],_,_,[predicate(Lex,agr(en_passive))],_,_,_,_,_,_),
m_verb(vbareinf,Part,Lex,_,_,_,_,_,_,_,_,_,_,_),
( var(Part),P=[]; nonvar(Part),P=[Part] ),
append([Vform],[to],Genpred).


gen([predicate(Lex,agr(toinfinitive))],Genpred,P):-
verb([Vform|_],_,_,[predicate(Lex,agr(toinfinitive))],_,_,_,_,_,_),
m_verb(Class,Part,Lex,_,_,_,_,_,_,_,_,_,_,_),
( var(Part),P=[]; nonvar(Part),P=[Part] ),
append([to],[Vform],Genpred).
```

4.4. Modifiers

In **horatio** modifiers can take the form of prepositional phrases. The main difference with **argument** pps is in the **preference** value assigned to the phrase: arguments are preferred to modifiers, according to the **densest match first** principle.

As to the internal structure of modifier pps, we distinguish between prepositions that can head modifier pps assigned to **nps** and those that can head modifier pps assigned to **vps**. Of course, some prepositions can head both np and vp modifier pps. The predicates **modppvp** and **modppnp** are used to implement the distinction. We have the following definitions for these predicates:

```
/* availability of preps in np and vp modifiers */

/* vp */

modppvp(on).
modppvp(in).
modppvp(at).
modppvp(with).
modppvp(for).
modppvp(about).

/* np */

modppnp(of). /* a book of importance */
modppnp(on). /* a book on each table */
modppnp(in). /* a bird in the tree */
modppnp(with). /* a book with a black cover */
modppnp(for).  /* a book for mary  */
modppnp(about) /* a book about the economic situation in Great-Britain */
```

Modifier pps are parsed by calls to the **xprepphrase** predicate. The reader is referred to the section on prepositional phrases, p. 62. We give here the code for **np** modifier pps:

```
modifier(P0,P1,np,Gap,Prefgen,posprec(1,Precpp),
         Rel,Intrel,Tree,subject(SUBJ,Sem)):-
       xprepphrase(P0,P1,Gap,index(J),npindex(I),Pref,Precpp,
              Prepform,Rel,
             Intrel,np_modifier,
            Tree,PPsem,PPsemnp),
       modppnp(Prepform),
       Prefgen is Pref + 1.
```

Np modifiers can also be clauses, either **ING**-clauses or **EN**-clauses (passives). Examples are:

the man reading a book is good
the book read by the man is good

These modifier clauses are parsed by calls to the `xverbphrase` predicate. The head noun must be passed to the ING or EN clause, where it will play the part of subject (active, ING-clause) or object (passive, EN-clause). Here is the code for modifier ING clauses:

```
modifier(P0,P1,np,Gap,Prefgen,posprec(1,_),Rel,Intrel,
     [np_modifier,Tree],
     subject([Function,[Cat,Index|Rest]],Sem)):-

  xverbphrase(P0,P1,subject([Function,[Cat,Index|VAR]],Semsubj),Gap,
   Pref,Tree,Rel,Intrel,
       ing,Tense,
       aspect(Aspect),Modality,
       Number,Person,
      Voice,nsubject(NSUBJ,Nsem)),
  sfok(Semsubj,Sem),
  Prefgen is Pref + 1.
```

The third argument to the modifier clause indicates that such modifiers are restricted to **np** modifiers. Note also that the body of the subject NP (**Rest**) is **ghosted** in the tree returned by the call to the `xverbphrase` predicate. Only the **index** is retained, to show **coindexing**. A semantic check is performed between the restriction placed on the subject of the verb phrase and the semantic feature list percolated from the head noun to its ING-modifier via the **subject** functor.

Such percolation is accomplished by a call to the `satisfylist` predicate. Remember that this predicate will also parse modifiers (the difference being that in the case of a modifier the noun's arglist is left untouched), so that modifiers have access to the **subject** functor:

```
corenounphrase(P0,P3,Gap,index(I),Pref,Weight,Rel,Intrel,
     Function,
          [Function,[nounphrase,index(I),
               agr(third,Number),DET,N,Parse]],
         Number,third,Sem):-
   determiner(P0,P1,DET,Number,Rel1,Intrel1) ,
   xnoun(P1,P2,N,Number,Sem,Arglist),
   satisfylist(P2,P3,Gap,Status,Pref,Prec,Rel2,Intrel2,_,
          Parse,np,Arglist,Func,
          subject([subject,[nounphrase,index(I),
               agr(third,Number),DET,N]],
         Verbsem)),
   sfok(Verbsem,Sem),
   ( var(Rel1),var(Intrel1),Rel=Rel2,Intrel=Intrel2;
    nonvar(Rel1),nonvar(Intrel1),Rel=Rel1,Intrel=Intrel1 ),
   ifthenelse(var(Status),Weight=2,Weight=3).
```

In the case of an EN-clause as modifier, we likewise use the **subject** functor in the clause. The clause is passive, and therefore the passive subject will come to occupy object position in the tree when passivization is undone in the EN-clause. The semantic check is therefore between the **Sem** feature list passed to the **subject** functor in the modifier clause and the new subject returned by the passive clause (**nsubject** functor; this would house the restriction on the object of READ, in our example).

```
modifier(P0,P1,np,Gap,Prefgen,posprec(1,_),Rel,Intrel,
          [np_modifier,Tree],
          subject([Function,[Cat,Index|Rest]],Sem)):-
   xverbphrase(P0,P1,
     subject([Function,[Cat,Index|VAR]],Semsubj),
         Gap,Pref,Tree,Rel,Intrel,
          en_passive,Tense,
          aspect(Aspect),Modality,Number,Person,
          passive,nsubject(NSUBJ,Nsem)),
   sfok(Nsem,Sem),
   Prefgen is Pref + 1.
```

In generation, modifier **EN-clauses** need a slightly special treatment: we must make sure that we do not generate the passive auxiliary. In fact, we DO generate it when we generate the passive clause, but we drop it from the string we return at the highest level, i.e. as second argument of the **gen** predicate:

```
/* written by students */

gen([[np_modifier,[pred_arg_mod_structure,prop(vce:passive,B,C,D)|R]]], Tail):-
   prepgen([clause,[pred_arg_mod_structure,prop(vce:passive,B,C,D)|R]], Prepmod),
   gen(Prepmod,Mod),
   flatten(Mod,[H|Tail]).

/* we return only the Tail; the Head (H) is the auxiliary, which does not appear in the modifier: written by students, * is written by students */
```

4.5. Canonical Order and Athematic Arguments

In the parse trees returned by **horatio** the arguments appear in a **canonical** order, and **athematic** arguments are excluded. This is achieved by a number of predicates. We begin by considering **`insort`**, which is simply an **insertion sort** that we perform on the arglist. **`Insort`** calls **`insert`**, which itself calls **`before`**. **`Insort`** also calls **`drop`**, which decides whether the argument is athematic and should be dropped.

```
insort([H|T],S):- drop(H),
                  insort(T,S).

insort([H|T],S):- not(drop(H)),
                  insort(T,L),
                  insert(H,L,S).

insort([],[]).
```

If the head of the list to be sorted is an athematic element, it is dropped and **`insort`** calls itself recursively on the tail. It the head is a thematic argument, the tail is sorted and then the head is inserted at its proper place by **`insert`**, whose code is the following:

```
insert(X,[H|T],[H|L]):-
        before(H,X),
        !,
        insert(X,T,L).

insert(X,L,[X|L]).
```

The element is inserted in the tail of the list if the head of the list comes before the element to be inserted; otherwise, the new element is inserted in front. The predicate **`before`** determines whether its first argument should come before its second argument. The decision is taken on the basis of the argument's function, which is always the head of the list representing it in the parse:

```
before([F1,R1],[F2,R2]):-
          assoc(F1,Rank1),
          assoc(F2,Rank2),
          Rank1<Rank2.
```

The predicate **`assoc`** associates a given rank with each thematic function:

```
assoc(subject,1).
assoc(object,2).
assoc(indirect_object,3).
assoc(indirect_object_2,3).
assoc(cplt_s,4).
assoc(subject_attribute,5).
assoc(object_attribute,6).
assoc(pp_arg,7).
assoc(vp_modifier,8).
```

It remains to look at the list of **athematic** functions, the ones that appear as arguments of the **`drop`** predicate :

```
drop([surf_subject,Rest]).
drop([surf_object,Rest]).
drop([subject_inf,Rest]).
drop([subject_ing,Rest]).
```

5. Semantics

horatio is very poor on semantic features. The values are organized in a hierarchy by means of Prolog clauses very much like the well-known **parent/ancestor** predicates. The semantic checks are of course performed by unification. It should be noted that nouns have **lists** of semantic features, whereas verbs place restrictions by means of **single** semantic features (one per argument, including the subject). The use of lists for nouns enables the lexicographer to maintain the principle of one macro-clause per reading. Consider the entries for COMPUTER and BOOK:

```
m_noun(computer_1,computer, computers,[thing,human],[]).
m_noun(book_1,book, books,[thing,abstract],[pp(opt,n:topic)]).
```

These entries allow us to deal with sentences such as the following where the verb or another predicative element places restrictions on the **np** whose head is a realization of the lexeme COMPUTER:

1) The computer thinks that the problem is hard to solve.
2) The computer should be repaired.

(if we accept that THINK requires a [+HUMAN] subject, and REPAIR a [+THING] object)

5.1. GF Level

It is obvious that semantic checks should not be applied at **surface level**, but at the **GF level**, where the grammatical functions corresponding to the specifications of the item's **arglist** have been retrieved.

A case in point is **passives**. In a sentence such as *The book has been read by the teacher*, passivization has demoted the **subject** to **by-phrase** status, and promoted the **object** to **subject** status. In the lexicon, however, the semantic restrictions imposed by the verb **read** are placed on the arguments as they appear before passivization reorganizes the **arglist** (to express things in chronological terms, a harmless metaphor so long as we keep in mind that for **horatio** passives and actives are simply **related**, without any priority being given to either).

Let us track the subject of the passive clause. The predicate **`reog`** (whose behaviour has been studied in the section on passives, p. 36) puts the **object** into the slots of the **nsubject** functor, which is passed on to the **`arglist`** predicate and ends up in the **`verbphrase`** predicate:

```
arglist(P0,P2,Gaps,ArgOrModFound,
           Pref,Posprec1,Rel,
           Intrel,Voice,Parse,NpOrVp,List,
           Func,subject(SUBJ,Semsubj),
           nsubject(NSUBJ,Nsem),
           Class):-
           reog(Voice,Class,subject(SUBJ,Semsubj),List,
               nsubject(NSUBJ,Nsem),Nlist,Func),
           satisfylist(P0,P2,Gaps,ArgOrModFound,Pref,Posprec1,
              Rel,Intrel,Voice,Parse,NpOrVp,Nlist,
              Func,subject(NSUBJ,Nsem)).

  verbphrase(P1,P3,subject(SUBJ,Semsubj),Gap,Pref,
            [pred_arg_mod_structure,
            prop(vce:V,asp:A,mod:Modality,tns:Tense),
            VERB,SParse],
          Rel,Intrel,Type,Tense,aspect(Aspect),Modality,
          Number,Person,Voice,nsubject(NSUBJ,Nsem)):-
   verb(P1,P2,Class,VERB,Type,Tense,
        Number,Person,Semsubj,Args) ,
   arglist(P2,P3,Gap,Status,Pref,Preclist,Rel,
            Intrel,Voice,Parse,vp,
            Args,Func,subject(SUBJ,Semsubj),
            nsubject(NSUBJ,Nsem),
            Class),
    (nonvar(Aspect); var(Aspect),A = none ),
    (Aspect = [] , A = none; Aspect \= [],A = Aspect ),
    (nonvar(Modality); var(Modality),Modality = none ),
    (nonvar(Tense); var(Tense), Tense = present ),
    (nonvar(Voice), V = Voice; var(Voice),V = active ),
    append([NSUBJ],Parse,AParse),
    insort(AParse,SParse).
```

At **sentence** level, the check is performed between the **subject np** and the semantic restriction specified by the **nsubject** functor:

```
sentence(P0,P2,Gaps,Prefs,[clause,VP],Type,Personvp,
              Number,Voice):-
append(Gapnp,Gapvp,Gaps),
xnounphrase(P0,P1,Gapnp,index(I),Prefnp,Weight,Rel1,
             Intrel1,subject,SUBJ,Number,Personnp,Semsubjnp) ,
var(Rel1),
agree(Personnp,Number,Personvp),
xverbphrase(P1,P2,subject(SUBJ,Semsubjvp),
             Gapvp,Prefvp,VP,Rel2,
             Intrel2,Type,Tense,aspect(Aspect),Modality,
             Number,Personvp,
             Voice,
             nsubject(NSUBJ,Nsemsubjvp)),
var(Rel2),
sfok(Nsemsubjvp,Semsubjnp),
accu(Prefs,[Prefnp,Prefvp,4]).
```

5.2. Inheritance

In **horatio** semantic features are organized hierarchically into classes, subclasses, sub-subclasses, etc. A subclass is meant to be able to satisfy a semantic requirement expressed in terms of its parent class.

The class/subclass relation is expressed by the predicate **up1**. Its first argument is the subclass, its second the parent class. We find:

```
up1(animal,living).
up1(human,living).
```

These two clauses are meant to convey the information that animals and humans are both living entities. More formally, that the class **animal** is a subclass of the class **living**, and that the class **human** is also a subclass of the class **living**.

The hierarchy is traversed upwards by the predicate **`up`**, whose definition is similar to that of the **`ancestor`** predicate, to be found in introductory tutorials on Prolog. The **`parent`** predicate's role is similarly played by the **`up1`** predicate:

```
up(X,Y):- up1(X,Y).
up(X,Y):- up1(X,Z),up(Z,Y).
```

When a semantic restriction is checked, the **`up`** predicate is called to see whether we do not have a subclass of the class we are looking for, in which case the semantic requirement is satisfied:

```
sfok(Sem,Semlist):- up(Sem1,Sem),inlist(Sem1,Semlist).
```

5.3. Percolation

Percolation can be defined as the copying up or down the tree of feature values. In Prolog this copying is of course done by **unification**. For instance, we need to be able to refer to the head noun's semantic feature list at the level of the whole noun phrase. Similarly, we sometimes need to refer to the semantic feature list of a noun phrase within a prepositional phrase.

Let us look at the first of these two examples of feature percolation. In the relevant defining clause for the **`corenounphrase`** predicate the two occurrences of the semantic feature list appear in **bold** type:

```
corenounphrase(P0,P3,Gap,index(I),Pref,Weight,Rel,Intrel,
        Function,
               [Function,[nounphrase,index(I),
                    agr(third,Number),DET,N,Parse]],
             Number,third,Sem):-
   determiner(P0,P1,DET,Number,Rel1,Intrel1) ,
   xnoun(P1,P2,N,Number,Sem,Arglist),
   satisfylist(P2,P3,Gap,Status,Pref,Prec,Rel2,Intrel2,_,
           Parse,np,Arglist,Func,
           subject([subject,[nounphrase,index(I),
                agr(third,Number),DET,N]],
         Verbsem)),
   sfok(Verbsem,Sem),
   ( var(Rel1),var(Intrel1),Rel=Rel2,Intrel=Intrel2;
    nonvar(Rel1),nonvar(Intrel1),Rel=Rel1,Intrel=Intrel1 ),
   ifthenelse(var(Status),Weight=2,Weight=3).
```

The **`xnoun`** predicate is itself defined in terms of the predicate **`noun`**:

```
xnoun(V1,V2,V3,V4,V5,V6):- noun(V1,V2,V3,V4,V5,V6).
```

The predicate **`noun`** itself is a macro-expansion of the lexical predicate **`m_noun`**:

```
noun([Singular|X],X,[noun(Lex,agr(sing))],sing,Sem,Arglist):-
m_noun(Lex,Singular,Plural,Sem,Arglist).
```

Here is an example of a **`m_noun`** clause, with the semantic list (fourth argument) in bold type:

```
m_noun(explanation_1,explanation,explanations,[abstract],[]).
```

5.4. From Semantic to Lexical Classes

Consider the phrase **PAY ATTENTION TO**. **Attention** is modifiable, and the np whose head it is plays a functional role, namely that of object of the verb **PAY**:

You should pay **more attention** to the problems he has mentioned.
Too much attention has been paid to these pseudo-problems.

Nevertheless **PAY ATTENTION TO** is a multi-word unit: the preposition is lexically determined and the sense of PAY is not assignable without considering the object.

The solution adopted for such mwu's in **horatio** is to use the slot reserved for **semantic** restrictions to code **lexical** restrictions. We have one entry for **PAY** (where the lexeme value is the phrase **pay_attention_1**) where we specify that the object must bear the semantic feature **attention**. We also have a reading of **attention** which bears the required feature in its semantic feature list:

```
m_verb(vobjfreepp,_,pay_attention_1,pay,pay,pay,pays,paying,
      paid,paid,paid,trans,human,
      [np(oblig,posprec(1,Wnp),object,attention),
       pp(oblig,posprec(1,Wpp),pp_arg,_,_,to)]).

/* they should pay attention to the problem he has seen */


m_noun(attention_1,attention,[attention],[]).
```

By the side of the entry for **pay_attention_1** given above, we need another one, where **attention to** is parsed as a **particle** attached to the verb. This is achieved by including the **string value** ***attention to*** in the **arglist**. This entry is needed to account for passives such as *The problem was paid attention to*, where the object is not *attention*, but *the problem*, i.e. the **object** argument in the arglist for this second entry (cf. the section on **double analysis**, p. 19):

```
m_verb(vtrphrprep,part0:'attention to', pay_attention_to_1_a,
      pay,pay,pay,pays,paying,
      paid,paid,paid,trans,human,
      [string(oblig,posprec(1,0),[attention,to]),
       np(oblig,posprec(2,Wnp),object,_)]).

/* the problem should be paid attention to */
```

We would also need another entry for **ATTENTION** to account for its uses outside the mwu **PAY ATTENTION TO**. The following would be appropriate:

```
m_noun(attention_2,attention,[abstract],[]).
```

Note that here **abstract** is a true semantic feature, not a lexical one.

5.5. Performing the Checks

`SFOK` is used to check that a semantic restriction placed by a verb or an adjective is satisfied by the noun phrase filling the verb's **argslot** or by the noun modified by the adjective.

`CHECKSEM` is used when the two values are both semantic restrictions. They are checked for compatibility, i.e. it is ascertained whether they meet somewhere by going up the semantic trees they belong to.

5.5.1. `Sfok`

To understand **`sfok`**, it is necessary to recall that nouns have semantic feature **lists**, whereas verbs and adjectives place semantic restrictions by means of **single** semantic features. For instance, the noun BOOK has the following entry in the lexicon:

```
m_noun(book_1,book,books,[thing,abstract],[pp(opt,n:topic)]).,
```

[thing,abstract] being its semantic feature list.

On the other hand, the verb READ has the following entry:

```
m_verb(verbtr,_,read_1,read,read,read,reads,reading,
      read,read,read,trans,human,
      [np(opt,posprec(1,Wnp),object,abstract)]).

/* she was reading */
```

where **human** is the semantic restriction on its subject, and **abstract** on its object.

To check whether a semantic restriction is satisfied, we try to find the feature embodying the semantic restriction placed by the verb on one of its arguments among the semantic feature list associated with the np or pp filling the argument position. The necessary list traversal is accomplished by **`inlist`**, which is basically a deterministic version of the **`member`** predicate:

```
inlist(X,X):- !.

inlist(Sem,[norestriction]):- !.

inlist(Sem,[Sem|X]):- !.

inlist(Sem,[_|X]):- inlist(Sem,X).
```

The first clause takes care of the cases where one of the two arguments, or both, is an **uninstantiated** variable. In such a case the test must succeed.

The second clause takes care of the special semantic feature list **[norestriction]** which is used with certain relative and interrogative pronouns. Here too the test succeeds, no matter what the semantic restriction placed by the verb or the adjective is.

The third and last clauses code a deterministic version of the well-known **member** predicate.

`Sfok` similarly opens with a number of clauses taking care of the special cases:

```
sfok(Sem,Sem):- !.

/* this clause is used to assign a value to a variable; necessary when the NP is a gap; the semantic
restriction in the VP will be projected onto the gap; it also takes care of cases where both arguments are
variables, e.g. checking the semantic features of an optional argument that is not realized in the S to be
parsed */

sfok(norestriction,X):- !.

sfok(X,norestriction):- !.

sfok([],X):- !.
```

The last two clauses call **`inlist`** to accomplish the list traversal of the semantic feature list associated with the np or pp argument. The last one calls **`up`**, whose role, as we have seen, is to permit inheritance:

```
sfok(Sem,Semlist):- inlist(Sem,Semlist),!.
sfok(Sem,Semlist):- up(Sem1,Sem) ,
                    inlist(Sem1,Semlist).
```

5.5.2. Checksem

As already said, we use **checksem** instead of **sfok** when we have two semantic restrictions, and want to know if they are **compatible**. This will be the case in the analysis of such sentences as *the man wants to read the book*, where what is passed on to the infinitive clause *to read the book* is the restriction that **wants** places on its subject. Consider the following piece of code:

```
satisfy([to|P0],P1,Gap,Prefgen,Posprec,Rel,Intrel,
        [Function,[clause,Tree]],
        np_vp(Type,to_inf,Function),
        subject([Sfunction,Treesubj],Semsubj1),
        subject([Sfunction,Treesubj],Semsubj1)):-
 xverbphrase(P0,P1,subject([subject,Treesubj],Semsubj2),
       Gap,Pref,Tree,Rel,
       Intrel,infinitive,Tense,
       aspect(Aspect),Modality,
       Number,Person,Voice,
       nsubject(NSUBJ,Nsem)),
 checksem(Nsem,Semsubj1),
 Prefgen is Pref + 4.
```

The **xverbphrase** predicate will put its semantic requirement on its subject in the **Nsem** variable in the **nsubject** functor (remember that it can be submitted to such argument reshuffling transformations as **passive**). The **Semsubj1** variable in the **subject** functor of the **satisfy** predicate records the semantic restriction on the subject of the verb that has already been parsed, **wants** in our case. We need to verify whether the two semantic restrictions are compatible, and we use **checksem** for this purpose.

The code for **checksem** is the following.

If we have one or two **uninstantiated** variables, or if the two variables are **unifiable**, **checksem** succeeds:

```
checksem(X,X):- !.
```

It may be the case that the second argument of **checksem** points to a list of features (this may happen through unification of semantic features), in which case we simply use **sfok**:

```
checksem(X,Y):- sfok(X,Y), !.
```

If both arguments refer to semantic restrictions, they must be **compatible**, i.e. must meet - be identical - somewhere on the **up** path in the hierarchy:

```
checksem(X,Y):- up(X,X1),
                checksem(X1,Y),
                !.
/* X is more restrictive than Y */


checksem(X,Y):- up(Y,Y1),
                        checksem(X,Y1).

 /* Y is more restrictive than X */
```

Note that in the case of **subject-to-subject raising** we do not use `checksem`, because such raising verbs (such as **SEEM**) do not place restrictions on their subjects. We simply pass the restriction placed by the **infinitive** on its subject to the higher level (by percolation), and leave the `sentence` predicate to make the appropriate call on `sfok`:

```
satisfy([to|P0],P1,Gap,Prefgen,Posprec,Rel,Intrel,
            [Function,[clause,Tree]],
            vp(Type,Function),
            subject([Sfunction,Treesubj],Semsubj1),
            subject([Sfunction,Treesubj],Semsubj1)):-
  xverbphrase(P0,P1,subject([subject,Treesubj],Semsubj2),
           Gap,Pref,Tree,Rel,
           Intrel,infinitive,Tense,
           aspect(Aspect),Modality,
           Number,Person,Voice,
           nsubject(NSUBJ,Semsubj1)),
  Prefgen is Pref + 2.
```

6. Parsing Issues

6.1. Dealing with Flexible Word Order

English is characterized by a word order that is neither wholly free nor entirely fixed. We shall therefore call it **flexible**. The first task is to isolate the relevant factors determining word order. For the cases dealt with in **horatio**, these are two: **argument canonical order** as specified in an item's **arglist**, and **structural weight**.

Structural weight has priority over argument order, although the exact weighting of these two factors is not easy to determine, and a good deal of it is heuristics, i.e. trying various possibilities and seeing how well they fit the data.

Argument order as determined in the lexical clause specifying the arglist is undoubtedly a factor. Consider the phrases built around the skeleton **consider x y**. If x and y have the same structural weight, x needs to refer to the object and y to the object's complement, not the other way round. In an interpretation where *motorists* is the object and *criminals* the complement, we can have

I consider motorists criminals

but not

* *I consider criminals motorists*

However, if the object is weightier than the complement, it will follow:

I consider criminals the motorists who drink and drive.

Sticking to the argument order specified in the rule would yield a much less acceptable sentence, also on account of the unintended object link between *drive* and *criminals*:

? I consider the motorists who drink and drive criminals.

In **horatio**, we use the **posprec** structure to house both the argument order and to leave room for the weight, to be filled when the argument is actually parsed. Consider again the entry for ALLOW, third reading, repeated here for convenience:

```
m_verb(vio,allow_3,allow,allow,allow,allows,allowing,
      allowed,allowed,allowed,allowed,trans,human,
      [np(oblig,posprec(2,Wnp1),object,thing),
       io(oblig,posprec(1,W2),indirect_object,human,_)]).
```

The first value in **posprec** can be determined in the argument list itself. Here, it is meant to convey the information that the indirect object should precede the direct object. The second value in **posprec** is left as a variable in the argument list. It will be instantiated when the actual **object np** and **io** (indirect object) are parsed. It reflects the weight of the element. For instance, a prepositional phrase will be assigned a certain weight, **nps** will differ in the weight they are assigned (for instance, an **np** consisting of a personal pronoun is very light, an **np** consisting of a head followed by a relative clause is heavy, etc.).

As already said, priority is given to the second value over the first in **posprec**, to allow for end placing of heavy elements. The relative importance to be assigned to the two values of the **posprec** structure is a matter for investigation[14].

The approach sketched here does not run into the problems that the **CLE** one (Core Language Engine) is confronted with (see **Pulman 1992**, in his section on subcategorization, p. 62 and foll.). He points out the the GPSG approach, as well as the CLE approach, does not "allow for the possibility of optional modifiers like PP or AdvP to appear between elements on a subcat list" . To remedy this problem CLE has "a version of the subcategorization schema that *splits* the list of complements and allows a modifying structure to intervene". In **Alshawi et al. 1992** we are given no further information on this splitting scheme. I believe that we should look for a solution that takes into account both arg order and weight. As has been said already, the crux lies in the weighting of these two factors.

6.2. Computing Preferences

In **horatio**, each parse is assigned a preference, expressed by a positive number. The greater the number, the higher the preference. The preference of a parse is a function of the preferences assigned to its constituents. The preference mechanism is built around the time-honoured principle of the redundancy of natural language, i.e. the **densest match - best match** principle. In practice, it means that arguments are preferred to modifiers.

Consider the following two predicate definitions. They both deal with the parsing of **prepositional phrases**. In the first case we have a pp as **argument**, in the second case as **modifier**.

```
satisfy(P0,P1,Gap,Prefgen,posprec(Pos,Precpp),Rel,Intrel,Tree,
            pp(Type,posprec(Pos,Precpp),
              Function,PPsemnp,PPsemvp,Prepform),
           subject(SUBJ,Semsubj),
           subject(SUBJ,Semsubj)):-
      xprepphrase(P0,P1,Gap,index(J),npindex(I),
                Prefpp,Precpp,Prepform,
                Rel,Intrel,Function,Tree,
                PPsem,PPsemnp),
      sfok(PPsemvp,PPsem),
      Prefgen is Prefpp + 4.


modifier(P0,P1,vp,Gap,Prefgen,posprec(1,Precpp),
      Rel,Intrel,Tree,subject(SUBJ,Sem)):-
      xprepphrase(P0,P1,Gap,index(J),npindex(I),Prefpp,Precpp,
                Prepform,Rel,
                Intrel,vp_modifier,
              Tree,PPsem,PPsemnp),
      modppvp(Prepform),
      Prefgen is Prefpp + 2.
```

Prefgen records the preference assigned to the phrase. In the first case (argument), it is the preference returned by the **xprepphrase** predicate plus 4. In the second case (modifier), the added value is only 2.

Consider now how the preference value is computed at the highest level, i.e. for whole sentences. Here is the relevant definition of the **sentence** predicate:

```
sentence(P0,P2,Gaps,Prefs,[clause,VP],Type,Personvp,
              Number,Voice):-
append(Gapnp,Gapvp,Gaps),
xnounphrase(P0,P1,Gapnp,index(I),Prefnp,Weight,Rel1,
             Intrel1,subject,SUBJ,Number,Personnp,Semsubjnp) ,
var(Rel1),
agree(Personnp,Number,Personvp),
xverbphrase(P1,P2,subject(SUBJ,Semsubjvp),
             Gapvp,Prefvp,VP,Rel2,
             Intrel2,Type,Tense,aspect(Aspect),Modality,
             Number,Personvp,
             Voice,
             nsubject(NSUBJ,Nsemsubjvp)),
var(Rel2),
sfok(Nsemsubjvp,Semsubjnp),
accu(Prefs,[Prefnp,Prefvp,4]).
```

The preference value for the whole S (**Prefs**) results from the accumulation (**accu** predicate) of the preference values of the subject np (**Prefnp**) and the verb phrase (**Prefvp**). But an S is more than the concatenation of an np and a vp. The np must be able to function as subject of the vp (in this sentence pattern) and this is ensured by the grammatical and semantic agreement checks (**agree** and **sfok** predicates). This link is the reason for the addition of the value 4 to the accumulated np and vp preference values. How do we find out that 4 is the appropriate value to add here ? The computing of preference values, like that of precedence values, is very much a question of heuristics, i.e. finding out by experiment which values give the best results.

Accu is a simple predicate which returns in its first argument the sum of the values listed in its second argument, which must be a list. The mode is the following:

```
accu(-Totalvalue,+Valuelist).
```

Accu is defined as follows:

```
accu(0,[]).

accu(Res,[Head|Tail]):-
                nonvar(Head),
                accu(Respart,Tail),
                Res is Head + Respart.

accu(Res,[Head|Tail]):-
                var(Head),
                accu(Res,Tail).
```

The first clause is the **non-recursive** case: an empty value list yields a total value of zero.

The second clause calls `accu` recursively on the tail of the value list, and adds the value of the head to the result, provided the head is instantiated.

The third clause takes care of uninstantiated heads: they are simply dropped, and `accu` is called on the tail of the list to return the final result.

6.3. Hard Coordination

horatio can deal with certain types of so-called **hard** coordination. The reader is referred to the last two sample parses (see Appendix F, page 155). The main principle is that only like elements can be conjoined, but the problem consists in determining the conditions that must be fulfilled for two elements to be regarded as similar enough to be coordinated. Here the argument list plays a crucial part, and its availability in the lexical entries themselves is a crucial condition for the strategies developed here to work.

In order to exemplify the approach taken in **horatio**, we shall look at coordination at the sentence level. We first deal with the easiest case: the two clauses do not display any gap. The only requirement is that they be of the same **Type**, either finite or non-finite.

For each linguistic phrase (from the clause level down) that can be coordinated, we define a super-predicate (by convention its name should begin with **x**), itself defined as a call to the simpler predicate, the parsing of a conjunction, and a recursive call on the complex predicate (the rules are therefore **right**-recursive). Of course, the second defining clause of the complex predicate defines it as a call on the simpler predicate, to get out of the recursion. In this way, we avoid the well-known problem of **left**-recursive rules (e.g. *np* --> *np coord np*) for a top-down parser as the one used in **horatio**.

At the clause level, we therefore have the following piece of code, where `xsentence` is the complex predicate and `c_sentence` the simpler one. Note that the gap value (third argument) is set to the empty list: the two clauses do not feature any gap.

```
xsentence(P0,P2,[],Prefs,[and_sentence,S1,S2],Type,Person,
        Number,Voice):-
inlist(and,P0),
c_sentence(P0,[and|P1],[],Pref1,S1,Type,Person,Number,
        Voice),
xsentence(P1,P2,[],Pref2,S2,Type,Person2,Number2,Voice2),
accu(Prefs,[Pref1,Pref2]).

/* getting out of the recursion */
 xsentence(A,B,C,D,E,F,G,H,I):-
   c_sentence(A,B,C,D,E,F,G,H,I).
```

We also note the simplistic call on `inlist`. It checks that the word AND is a member of the remaining word list[15]. This call is for efficiency only, and would have to be revised or dropped in a version of **horatio** that attempted to deal with other coordinating conjunctions than AND (recall that the comma is the coordinating conjunction of choice when the coordination has more than two members).

We next consider the case where we have a gap. It needs to be shared by the two conjoined clauses. Consider:

The linguistics which he thinks *he teaches and she likes ...*

The italicized bit is a conjunction of two S's sharing the same gap, namely the relative pronoun. Sharing the gap implies here that the gap is of the same type (**np** or **pp**) and points to the same antecedent (and also, of course, has the same index). We therefore have the following piece of code:

```
xsentence(P0,P2,
        [gap(_,_,_,_,FunctioninS2,_,index(I),_)],
         Prefs,[and_sentence,S1,S2],Type,Person,Number,Voice):-
  inlist(and,P0),
  c_sentence(P0,[and|P1],
          [gap(Ref,Gaptype,_,_,FunctioninS1,_,index(I),_)],
         Pref1,S1,Type,Person,Number,Voice),
  nonvar(FunctioninS1),
  xsentence(P1,P2,
        [gap(Ref,Gaptype,_,_,FunctioninS2,_,index(I),_)],
         Pref2,S2,Type,Person2,Number2,Voice2),
  nonvar(FunctioninS2),
  accu(Prefs,[Pref1,Pref2]).
```

Note here the calls on `nonvar`, to make sure that the gaps are real. Function assignment can only happen in the clause itself: if the function value is not left uninstantiated, it is evidence that a piece of structure was missing, since a function was assigned to its place-holder (the gap).

The next case we consider is that of a gap that is filled somewhere to the **right** (the gaps in the previous case were filled on the left, as in interrogative or relative clauses). An example is

John likes and Mary reads the book

which we will analyse as two gapped S's (1: *John likes* GAP; 2: *Mary reads* GAP) followed by the gap filler (*the book*).

Here we require that the gap fill the **same** function in the two coordinated S's (the object function in the example), and we check that the gap filler is semantically OK for both gaps. The gap functor has two positions for semantic restrictions (one for **np** gaps and one for **pp** gaps) and we carry out the check between the appropriate slot in the gap functor and the semantic feature list associated with the noun phrase or prepositional phrase filling the gap. Here is the code in the case of **np** gaps:

```
xsentence(P0,P3,[],Prefs,[and_sentence,S1,S2,NP],Type,
         Person,Number,Voice):-
  inlist(and,P0),
  sentence(P0,[and|P1],
     [gap(NP,npgap,Npsem1,_,FunctioninS1,_,index(I),_)],
     Pref1,S1,Type,Person,Number,Voice),
  nonvar(FunctioninS1),
  sentence(P1,P2,
     [gap(NP,npgap,Npsem2,_,FunctioninS2,_,index(I),_)],
     Pref2,S2,Type,Person2,Number2,Voice2),
  nonvar(FunctioninS2),
  FunctioninS1=FunctioninS2,
  xnounphrase(P2,P3,[],index(I),Pref,Weight,Rel,
        Intrel,**FunctioninS1**,**NP**,Numbernp,Personnp,**Sem**),
  **sfok(Npsem1,Sem),**
  **sfok(Npsem2,Sem),**
  accu(Prefs,[Pref1,Pref2]).
```

Note that at the highest level the clause resulting from the coordination of the two gapped clauses and the gap filler must feature an empty list as gap value. This is in contrast with the preceding case where the gap was percolated to the resulting clause.

The most complex case is that of clauses which are gapped in their main verb, such as

Mary teaches linguistics and ***John mathematics****.*

Matsumoto 1991 leaves such cases to be dealt with by a device which does not belong to the main parsing scheme, the Extra-Grammatical Sentence Module. He writes (page 10): "Because the Extra-Grammatical Sentence Module is activated only when no sentence structure is found, the gap type of coordination handling is never tried if there is at least one possible parse obtained from the input". One of the sentences he discusses is the following:

John will cook the meals today and Barbara tomorrow (missing vp in the second S).

If the 'cannibalistic' reading is not rejected, it will preclude finding the 'gapped' reading. Matsumoto is aware of this danger (p.11), although he has not seen that the interpretation of one of his example sentences falls prey to it.

The treatment of such gapped S's in **horatio** crucially depends on the availability of **frame information** in the lexicon. As a matter of fact, we need to go into the structure returned by the parsing of the first clause, get at the **Class** and **Lex** value for the verb, call the corresponding macro-clause (**`m_verb`**) to get at the **Args** required by the predicate, and attempt to parse what follows the subject of the second clause with a call to the **`arglist`** predicate, to which the required **Args** and **Class** values are passed. The **`m_verb`** clause also yields the semantic restriction on the subject and we use it to make sure that the subject of the second clause is semantically appropriate to its (gapped) verb. The following is the commented code for this more complex case:

```
xsentence(P0,P4,[],Prefs,
  [and_sentence,
   [clause,  [pred_arg_mod_structure,
            prop(vce:V,asp:A,mod:Modality,tns:Tense),
            [predicate(**Lex**,Agr)],
            SParse]],
   [clause,  [pred_arg_mod_structure,
            prop(vce:V,asp:A,mod:Modality,tns:Tense),
            [predicate(**Lex**,_)],
            SParse2]]
   ],
   **Type**,**Person**,**Number**,**V**):-
  inlist(and,P0),
  xnounphrase(P0,P1,[],index(I),Prefnp1,Weight1,Rel1,
             Intrel1,subject,SUBJ,Number,Personnp,Semsubjnp) ,
/* e.g. Mary */

  var(Rel1),
  agree(Personnp,Number,Person),
  verbphrase(P1,[and|P2],subject(SUBJ,Semsubj),[],Prefvp1,
            [pred_arg_mod_structure,
            prop(vce:V,asp:A,mod:Modality,tns:Tense),
              [predicate(Lex,Agr)],
             SParse],
            Relvp1,Intrelvp1,Type,Tens,aspect(Aspect),Mod,
            Number,Person,Voice,nsubject(NSUBJ,Nsem)),
/* e.g. teaches linguistics */

/* P2,P3: subject of the second S, e.g. John */

  xnounphrase(P2,P3,[],index(I2),Prefnp2,Weight2,Rel2,
           Intrel2,subject,NP,Number2,Personnp2,Sem),
  var(Rel2),
  m_verb(Class,_,Lex,
        _,_,_,_,_,_,_,_,
        Semsubj,Args),

/* P3,P4: argument list in the second S */
/* e.g. mathematics */

  arglist(P3,P4,[],Status,Prefvp2,Preclist,Relvp2,
              Intrelvp2,Voice,Parse2,vp,
              Args,Func,subject(NP,Semsubj),
              nsubject(NSUBJ2,Nsem),
              Class),
  nonvar(Status),
  append([NSUBJ2],Parse2,AParse2),

/* the append predicate is used to append the subject at the beginning of the parse tree produced by
arglist, from which the subject is excluded */

  insort(AParse2,SParse2),
  sfok(Nsem,Sem),
  accu(Prefs,[Prefnp1,Prefvp1,Prefnp2,Prefvp2]).
```

A Note on Generation

The generation of coordinate structures is straightforward: we generate the first conjunct, append the coordinating conjunction AND, and then generate the second conjunct (recall that **horatio** knows only about one coordinator, AND). But there is one case that is slightly more complex. It is due to the fact that sometimes the parser factors out a piece of information, such as for instance the **Property** field (modality, aspect, voice, tense) in the case of coordinated **YES-NO questions**. We therefore need to redistribute the factored out information. We first make sure that factorisation has taken place by checking that the factored out element is uninstantiated at the places where it will have to be redistributed. The redistribution is accomplished by the following piece of code:

```
gen([yes_no_question,[clause,[and_pred_arg_mod_structure,Properties,
[pred_arg_mod_structure,Vprop1|Rest1],
[pred_arg_mod_structure,Vprop2|Rest2]]]],
Sentence):-
    var(Vprop1),
    var(Vprop2),
    gen([yes_no_question,[clause,[pred_arg_mod_structure,
     Properties|Rest1]]],S1),
    gen([yes_no_question,[clause,[pred_arg_mod_structure,
     Properties|Rest2]]],S2),
    append(S1,[and],S1and),
    append(S1and,S2,Sentence).
```

Such redistribution also applies to the **Function** value in the case of coordinated nps:

```
gen([Function,[and_nounphrase,NP1,NP2]],Gennp):-
   gen([Function,NP1],NP1gen),
   gen([Function,NP2],NP2gen),
   append([NP1gen],[and],L1),
   append(L1,[NP2gen],Gennp).
```

6.4. Long Distance Dependencies

The treatment of long distance dependencies is similar to that suggested in **GPSG (Generalized Phrase Structure Grammar)** and **MLG**[16] **(Modular Logic Grammar)**. **Gap threading** is used (via **`APPEND`**, not via **difference list** techniques). In **horatio** we use two types of gaps: **np** gaps and **pp** gaps. Consider:

The house in which he lives is nice.
The house he lives in is nice.

In the first relative clause (*he lives*) we have a **pp** gap filled in by the relative prepositional phrase *in which* (a relative prepositional phrase is a prepositional phrase which contains a relative **np**). In the second relative clause (*he lives in*) we have an **np** gap filled by the zero relative pronoun. Coindexing will ensure that the **np** within the **pp** in the first case is coindexed with the antecedent *the house*; in the second case the zero relative will likewise be coindexed with the antecedent *the house*.

Np gaps come into existence when an **argbearer** misses an **np argument** in the word list it is attempting to parse, i.e. cannot **`satisfy`** it as a regular np and calls on the relevant clause in the **`nounphrase`** predicate, which returns something without consuming anything in the input list (hence the **P0,P0** values as **input** and **output** lists in the code quoted below). When the gap is parsed, the only value that is set is that of the **function** played by the gap. This is the reason why we often check whether this value is instantiated: if it is, we can be sure that the **argbearer** has missed one of its arguments. All the other values may come to be instantiated by **unification**, as will become clear when we discuss the **percolation** of the features in the **gap functor**. Here is the clause for the parsing of **noun phrase gaps**:

```
nounphrase(P0,P0,
      [gap(Antecedent,npgap,Npsem,_,Function,_,index(I),_)],
         index(I),0,0,Rel,Intrel,Function,
         [Function,
          [nounphrase,index(I),agr(third,Number),NP]],
         Number,third,Npsem).
```

The zero values are the **Weight** and **Preference** values. Note that in the parse tree of a gapped np, only the **function**, **index** and **agreement** value are kept; it should be noted that **NP** is an uninstantiated variable, not a copy of the antecedent.

The gap functor holds the following features:

Antecedent: this feature can hold a full copy of the antecedent of the gap;

gap type: **npgap** or **ppgap**;

Npsem: semantic restriction on the np;

PPsem: semantic restriction on the pp;

Function: function filled by the gap in clause structure;

Prep : preposition, if any;

Index: we keep an index, so that we do not need a full copy of the antecedent to show the coindexing relation;

NPindex: indexes the np within a pp; necessary for such pps as *about whom*, where the np *whom* must be coindexed with its antecedent.

A pp gap is similar. Here is the relevant clause for the **`prepphrase`** predicate:

prepphrase(P0,P0,
    [**gap(Antecedent,ppgap,Semnp,Sempp,Function,Prep,index(I),npindex(J))**],
    index(I),npindex(J),0,0,Prep,Rel,Intrel,**Function**,
    [**Function**,[prepphrase,**index(I)**,_,_]],
     Sempp,Semnp).

Note that the two index values are kept distinct. Only the index value for the whole pp is preserved in the parse tree.

If, apart from **Function**, the values in the gap functor do not come from the parsing of the missing **np** or **pp**, where *do* they come from? In order to show the instantiation of the values carried out by unification (of which percolation is a special case, being unification up or down a tree), we shall examine the treatment of relative clauses.

The passing of values is to be found at two places:

1) in a defining clause for **`nounphrase`** which accounts for nps governing a relative clause, we percolate the features from the antecedent to the gap functor carried by the relative clause predicate;

2) in the defining clauses for relative clauses, we percolate the values from the gap functor in the relative clause to the **`xsentence`** predicate that parses the body of the relative clause. It follows that the **`xsentence`** predicate must also bear a gap functor (the gap may come from the subject or from an argument in the **arglist** of an **argbearer** that is a constituent of the S).

Two things need to be kept in mind. The first is that even if we like to think of a direction in the percolation process, it has none. It is a relation between values, not even a process, and certainly not a copy from one place to another. We have already seen that the **Function** value could not be assigned before the parsing of the gap.

The second is that in the process just outlined, it is the gap functor carried by the relative clause predicate which provides the link between the antecedent and the sentence (with a missing arg) which makes up the body of the relative.

Let us consider first the **`nounphrase`** clause which takes care of nps governing a relative clause. We have already looked at it (in the section on **nps**, see p.55), but this time we shall concentrate our attention on the unification pattern between antecedent and gap functor in the relative clause predicate. Here is the code:

nounphrase(P0,P2,[],index(I),
          Preftop,3,_,_,Function,
          [Function,Newrest],
          Number,third,Sem):-
 corenounphrase(P0,P1,[],**index(I)**,
              Pref,Weight,Rel,Intrel,Function,

```
                [Function,Rest],
                Number,third,NounSem),
 ( var(Intrel); Intrel \= int ),      /* not an interrogative */
 var(Rel),
 xrelclause(P1,P2,Prefrel,RELCL,
              [gap(_,_,Npsem,_,_,_,index(I),_)],
              Personinrel,Funcinrel,NounSem,Number),
 sfok(Npsem,NounSem),
 append(Rest,[RELCL],Newrest),
 accu(Preftop,[Pref,Prefrel]).
```

The shared information covers the **index**, which is the means of indicating **coindexing** relationships in **horatio**. Note that the semantic feature list of the antecedent np (**NounSem**) is not copied to the **gap functor**. The latter will hold a semantic restriction (**Npsem**), and the call to **sfok** will make sure that the antecedent fits the semantic restriction placed in the relative clause.

We also see that **Number** and **Nounsem** are transferred from the antecedent to the relative clause predicate (`xrelclause`), but are not part of the gap functor. We shall understand the rationale behind such transfer when we look at the `xrelclause` predicate.

Let us take one type of relative clause to follow the fate of the percolated features: we shall look at relative clauses which feature a relative pronoun (non-zero relative) and an np gap. We should bear in mind that the last two features of the `relclause` predicate are copied from the antecedent: antecedent semantic feature list and antecedent number, in that order.

The code is the following:

```
/*   the man whom the woman likes
   the man the book about whom the woman likes */

relclause(P0,P2,Prefrel,[relative_clause,NP,S],
             [gap(_,npgap,Semrel,_,_,_,index(J),_)],
             Personrel,Functioninrel,NounSem,Numberant):-
    nounphrase(P0,P1,[],index(I),Prefnp,Weight,Semrel,
              rel(J),Functioninrel,
              NP,Number,third,Semwholenp),

/* bear in mind that the structure spanned by nounphrase is NOT the antecedent, but the relative pronoun (e.g. which) or the noun phrase containing a relative element (e.g. a book about whom) */

/* I and J can be the same, but need not: in a book about whom you have read I would refer to book, J to whom */

    nonvar(Semrel), /* the np must have a rel feature */
    sentence(P1,P2,
              [gap(_,npgap,Npsem,_,Functioninrel,_,index(I),_)],
              Prefs,S,finite,Personrel,Numberrel,Voice),
    nonvar(Functioninrel),
    ( Weight \= 1 , sfok(Npsem,Semwholenp),
      ifthen((Functioninrel=subject),
         agree(third,Number,Personrel))
;
      Weight = 1 , sfok(Npsem,NounSem),
       ifthen((Functioninrel=subject),
         agree(third,Numberant,Personrel))) ,
    accu(Prefrel,[Prefnp,Prefs]).
```

/* checking the **Weight** value is a slightly roundabout way of getting to know whether we have a relative pronoun alone (such as *whom* - Weight is 1) or an np containing a relative (such as *a book about whom* - Weight is heavier than 1). */

The only feature that is directly shared between the **gap functor** in the `relclause` predicate and that in the `sentence` predicate is the **gap type**: here both gaps need to be **np** gaps. However, the sharing of values is more extensive if one considers the features in the `nounphrase` predicate (parsing the relative pronoun, or the noun phrase containing a relative pronoun). First of all, an **index** is shared: the index of the relative element within the **noun phrase** (J) is to be found at the **relclause** level. Second, the **Semrel** at the **relclause** level is shared by the `nounphrase` predicate. **Semrel** will be the same as **Semwholenp** when the noun phrase consists only of a relative pronoun. The relevant piece of code for the parsing of relative pronouns is repeated below:

```
nounphrase(P0,P1,[],index(I),
            0,1,Semnp,rel(I),Function,
            [Function, [nounphrase,index(I),agr(third,Number),Tree]],
            Number,third,[Semnp]):-
 relative(P0,P1,Tree,Semnp,Sempp,np).
```

We see that here the **rel** index and the index for the whole np are also shared. The semantic feature is shared, but inserted as a one-element list to conform to **np** semantics, which is always in list format.

At the level of the `sentence` predicate, the semantic restriction on the gap is expressed in the **Npsem** feature. If we have a noun phrase that consists of a relative pronoun only (*the man **whom** he likes*), its weight will be 1, and the semantic check can be carried out between the semantic restriction placed by the `sentence` predicate on the gap (**Npsem**) and the semantics of the antecedent itself, which is housed in the last argument but one of the `relclause` predicate (**NounSem**). If the noun phrase contains a relative pronoun, but is not limited to such a pronoun (as in *the man **a book about whom** I have read*), its weight will be different from 1, and the semantic test is to be carried out with the semantics of the whole np as second argument (**Semwholenp**).

If the noun phrase consists of a relative pronoun only (**Weight** equals 1), and that relative is subject in the relative clause, we need to check subject-verb agreement in the relative clause. We can do so because the **number** of the antecedent has been percolated as last argument of the `relclause` predicate (**Numberant**), and the **person** feature is percolated to sentence level in the `sentence` predicate (**Personrel**). The relative pronoun is always third person[17], so we have a check where the first two arguments (the np ones) are **third** and **Numberant** (the number of the antecedent) and the third argument (the verbal one) is **Personrel** carried by the `sentence` predicate.

On the other hand, if the subject of the relative clause is a whole np containing a relative pronoun (**Weight** is different from 1), as in *The genius **a book about whom** has just been published*, we need to check agreement with the second argument representing the number of the whole np, not that of the relative pronoun within it. We therefore have

agree(third, Number, Personrel),

where **Number** is the number of the whole np, returned by the `nounphrase` predicate.

To illustrate pp gaps we shall look at interrogative clauses. We take as illustration finite interrogative clauses featuring a pp gap, as in

He knew in whose library she lived.

We analyse the interrogative clause as a prepositional phrase followed by a sentence featuring a pp gap, and we rely on unification to relate the pp with the pp gap. The code is the following:

```
intclause(P0,P2,Prefint,[interrogative_clause,PP,S]):-
   xprepphrase(P0,P1,[],index(I),npindex(J),Prefpp,
        Weight,Prep,
           Semrelint,int,Function,
           PP,PPsem,Semnp),
   nonvar(Semrelint),
   sentence(P1,P2,
           [gap(PP,ppgap,Npsem,PPsem,Function, P_Prep,index(I),npindex(J))],
           Prefs,S,finite,Personint,Numberint,Voice),
   nonvar(Function),
   accu(Prefint,[Prefpp,Prefs]).
```

In order to understand the need for **two** indices in pps (**index** and **npindex**), we return to the treatment of relative clauses similar to the interrogative clause we have just looked at. Consider

the man in whose library she lived

We need to establish a link between *whose* and *the man*. This link will be carried by the **rel(X) functor** within the pp.

First, we must recall the structure of the **lexical** clause accounting for **relative** *whose*:

```
 determiner([whose|X],X,[det(whose),index(I)],_,norestriction,rel(I)).
```

The last argument **(rel(I))** has a variable position for an index - **I** - which will be instantiated so as to keep track of the relation between **whose** and its antecedent. Note that this index is also returned in the parse tree associated with determiner *whose.*

Second, note that the **rel(I)** functor is percolated from the **determiner** to the **noun phrase** level when the **np** is parsed. It is called **Intrel1** at the **determiner** level in the piece of code below, and **Intrel** at the **np** level.

```
corenounphrase(P0,P3,Gap,index(I),Pref,Weight,Rel,Intrel,
        Function,
              [Function,[nounphrase,index(I),
                   agr(third,Number),DET,N,Parse]],
             Number,third,Sem):-
   determiner(P0,P1,DET,Number,Rel1,Intrel1) ,
   xnoun(P1,P2,N,Number,Sem,Arglist),
   satisfylist(P2,P3,Gap,Status,Pref,Prec,Rel2,Intrel2,_,
           Parse,np,Arglist,Func,
           subject([subject,[nounphrase,index(I),
                 agr(third,Number),DET,N]],
          Verbsem)),
   sfok(Verbsem,Sem),
   ( var(Rel1),var(Intrel1),Rel=Rel2,Intrel=Intrel2;
    nonvar(Rel1),nonvar(Intrel1),Rel=Rel1,Intrel=Intrel1 ),
   ifthenelse(var(Status),Weight=2,Weight=3).
```

From the **np** it will be percolated to the **pp**:

```
prepphrase(P0,P2,Gap,index(J),
        npindex(I),Prefnp,3,PREP,Rel,Intrel,Function,
            [Function,[prepphrase,index(J),prep(PREP),NP]],
            Sempp,Semnp):-
    prep(P0,P1,[prep(PREP)],Sempp) ,
    nounphrase(P1,P2,Gap,index(I),Prefnp,Precnp,Rel,
          Intrel,np_arg_of_prep,NP,Number,Person,Semnp).
```

We can now look at the defining clause for relative clauses where the gap is a pp gap:

```
relclause(P0,P2,Prefrel,[relative_clause,PP,S],
          [gap(_,ppgap,Semrel,PPsem,_,Prep,index(J),npindex(I2))],
          Personrel,Function,NounSem,Numberant):-
    xprepphrase(P0,P1,[],index(I),npindex(I2),
        Prefpp,Weight,Prep,
            Semrel,rel(J),
            Function,
            PP,PPsem,Semnp),
    nonvar(Semrel),
    sentence(P1,P2,
          [gap(PP,ppgap,Npsem,PPsem,Function,P_Prep, index(I),npindex(I2))],
          Prefs,S,finite,Personrel,Numberrel,Voice),
    nonvar(Function),
    accu(Prefrel,[Prefpp,Prefs]).
```

The index of the **rel** functor at the **pp** level must be the same as that of the index functor in the **relclause** gap, i.e. point to the **antecedent**. In our example sentence, **I** would refer to the whole pp (*in whose library*), **I2** would refer to the np within the pp (*whose library*), and **J** to the antecedent (*the man*).

7. Generation Issues

There is some research in generation with unification grammars which tends towards the ideal of reversibility: the same grammar is made use of in analysis and generation (cf. **Isabelle et al. 1988**; **Dymetman and Isabelle 1990**). By **same grammar** is meant here the same **code** embodying the grammar, not just the same linguistic concepts.

horgen, the generation counterpart of **horatio**, does **not** use the same grammar. The reason is that parses are supposed to be fully unambiguous and therefore generation ought to be much simpler than analysis, which is faced with potentially and actually ambiguous inputs.

In this piece of research generation is used mainly as a test, to make sure that the surface text is retrievable from the parses produced by the analyzer. Such a property of parses is crucial in machine translation.

In **horgen** generation is **not backtrackable**. Only one surface string is produced per parse. The non-backtrackability of generation implies that the clause order of the generation predicates is significant, and that the model **exceptions first, general case last** can be used without necessarily making use of the cut.

7.1. Freezing the Variables in the Parses

The parses produced by **horatio** account for **coindexing** by means of **shared** variables in the **index(VAR)** functors. We need to convert them to strings before we start generating from them, to prevent generation from running wild. The conversion of **variable** to **string** is achieved by a small script applied to the parses before feeding them to **horgen**.

This script is a **Kornshell** script executable under the **MKS** implementation of Unix utilities for OS/2 and MS-DOS. It runs as follows:

```
sed -E -e "s/index\(\_(....)\)/index\(\$\1\$\)/g"  $1 > tmp.tmp
if [ -s tmp.tmp ]
 then
    mv tmp.tmp $1
 else
    echo "Error !!!"
fi
```

SED is a call to the UNIX **stream editor**. The command is to perform a global (**g**) substitution (**s**) of the source pattern **index(_....)**, where the dots stand for alphanumeric characters. The target pattern includes the 4 alphanumeric characters (the digit 1 in the **target** pattern refers to the first parenthesized expression in the **source** pattern). They appear between dollar signs, the string delimiter in Arity Prolog. The variable marker (the underscore) in the source pattern is not copied to the target pattern. The net result is to transform variables into strings. An example is given below in the section on the cycle.

7.2. The Cycle

horatio produces parses in which the **gf's** (grammatical functions) are deep, i.e. correspond to the specifications in the **arglist** of the predicate, the argument bearer. To retrieve the surface string, this work must be undone. Since the days of standard transformational grammar, it has been known that the **control** relations (i.e. the control of the subjects of subordinate clauses by higher **nps** in argument functions), and transformations such as **passive** and **raising**, apply cyclically, to the more deeply embedded clauses first.

Even if we do not wish to stick to a transformational treatment, this insight of TG is still crucial. To retrieve the surface string, we will need to undo the work done by **horatio** in cyclic fashion. Hence the title of the appropriate section in the **horgen** code: **The Cycle**.

Consider the parse corresponding to the surface string:

I believe him to have been killed.

Schematically, it looks like this:

I believe [VAR has killed him].

We first need to apply the **`passive`** predicate in generation mode, whose job is to move the object into subject position and to create a **by-phrase** for the subject (here this by-phrase will not lead to the generation of anything, since it has no lexical material in the **np** slot). We obtain something along the lines of:

I believe [he has been killed]

We now apply **subject to object raising**: the subject of the subordinate clause is raised to object position in the main clause. At the same time the main verb of the subordinate clause is turned into an infinitive with TO:

I believe him [he to have been killed]

Control will now take care of **ghosting** (i.e. depriving of lexical content) the controlled subject.

I believe him [VAR to have been killed]

Generating the **VAR** will yield the empty list, which will later disappear by the list flattening and appending operations which complete the generation process:

I believe him to have been killed.

We will now consider the generation of the resulting sentence in somewhat more detail, as a way of illustrating the cycle principle in action.

We start from what the parser gives us. The pretty-printed parse looks like this:

```
 12
  clause
   pred_arg_mod_structure
   prop(vce: active,asp: none,mod: none,tns: present)
    predicate(believe_1,agr(finite,present,sing,first))
     subject
      nounphrase
      index(_0398)
      agr(first,sing)
      ppro(first,sing,_041C)
     object
      clause
       pred_arg_mod_structure
       prop(vce: passive,asp: [perfect],mod: none,tns: present)
        predicate(kill_1,agr(en_passive))
         object
          nounphrase
          index(_0580)
          agr(third,sing)
          ppro(third,sing,masc)
```

We see that the object of **believe** is clausal, and that **kill** has an object and no subject (passive has been undone; but the **agr** functor and the **property** list record that we have a passive clause).

Of course, the generator does not work on pretty-printed objects. Here is the **raw** object produced by the parser. It is a **Prolog term** (carriage returns have nonetheless been added to improve readability):

[12,[clause,[pred_arg_mod_structure,
prop(vce: active,asp: none,mod: none, tns: present),
[predicate(believe_1,agr(finite,present,sing,first))],
[[subject,[nounphrase,**index(_0398)**,agr(first,sing),ppro(first,sing,_041C)]],
[object,[clause,[pred_arg_mod_structure,
prop(vce: passive,asp: [perfect],mod: none,tns: present), [predicate(kill_1,agr(en_passive))],
[[object,[nounphrase,**index(_0580)**,
agr(third,sing),ppro(third,sing,masc)]]]]]]]]]].

We note that two of the three **uninstantiated** variables appear as arguments of the **index** functor. These are the ones that we need to **freeze**. We do so by converting them into **strings**, as explained in the preceding section. The result is as follows:

```
[12,[clause,[pred_arg_mod_structure,
prop(vce: active,asp: none,mod: none,tns: present),
[predicate(believe_1,agr(finite,present,sing,first))],
[[subject,[nounphrase,index($0398$),agr(first,sing),ppro(first,sing,_041C)]],
[object,[clause,[pred_arg_mod_structure,
prop(vce: passive,asp: [perfect],mod: none,tns: present),
[predicate(kill_1,agr(en_passive))],
[[object,[nounphrase,index($0580$),
agr(third,sing),ppro(third,sing,masc)]]]]]]]]]].
```

We have **two** clauses, with the **controlled** clause in **second** position in the arglist of the **controlling** clause. The following defining clause for **`prepgen`** therefore applies:

```
prepgen([H2_cl1,[pred_arg_mod_structure,Prop1,Pred1,[Firstarg_cl1,
        [H1_cl2,[clause|Rcl2]|R2]|R1]]],
Res4):-
second_header(H2_cl1),
first_header(H1_cl2),
passive([clause|Rcl2],Res1),
control([H2_cl1,[pred_arg_mod_structure,Prop1,Pred1,[Firstarg_cl1,
      [H1_cl2,Res1|R2]|R1]]],Res2),
oraising(Res2,Res3),
passive(Res3,Res4).
```

The first job carried out by **`prepgen`** is to check the **clause headers**, to make sure that we are in the right environment to apply the processes defined in **`passive`**, **`control`**, and **`oraising`**. Let us look at the check **`second_header`**. It will be instantiated in the following way when the call is made:

```
second_header(clause).
```

The call will succeed because it is a **fact** recorded in the packet of clauses for the predicate **`second_header`**, listed below:

```
second_header(clause).
second_header(np_modifier).
second_header(adj_arg).
second_header(arg).
second_header(noun_arg).
```

The next check, **`first_header`**, will consist of the following call

```
first_header(object).
```

and will also succeed, for a similar reason.

`Passive` will then try to apply in the controlled clause. The instantiation is the following when the call is made:

```
passive([clause,[pred_arg_mod_structure, prop(vce:passive, asp:[perfect], mod:none, tns:present),
[predicate(kill_1,agr(en_passive))],
[[object,[nounphrase, index($0580$),agr(third,sing),ppro(third,sing,masc)]]]]],_2888).
```

_2888 will get instantiated to the following term:

```
[clause,[pred_arg_mod_structure, prop(vce:passive, asp:[perfect], mod:none, tns:present),
[predicate(kill_1,agr(en_passive))],
[[subject,[nounphrase, index($0580$),agr(third,sing),ppro(third,sing,masc)]]]]
```

We see that the (deep) object has been turned into a (surface) subject. The job was performed by the following clause for `passive`:

```
passive([H1,[pred_arg_mod_structure,prop(vce:passive,B,C,D),Pred1,
   [[object,O]|Otherargs]]],
[H1,[pred_arg_mod_structure,prop(vce:passive,B,C,D),Pred1,
   [[subject,O]|Otherargs]]]):- second_header(H1).
```

Note that the job is carried out in the head of the clause, the body consisting only of a header check.

The next predicate to be called is **`control`**, which gets the whole structure as argument, but with the result of the call to `passive` instead of the original controlled clause. The instantiation when the call is made is the following:

```
control([clause,[pred_arg_mod_structure,
prop(vce:active,asp:none, mod:none,tns:present),
[predicate(believe_1,agr(finite,present,sing,first))],
[[subject, [nounphrase,index($0398$),agr(first,sing), ppro(first,sing,_041C)]],
[object,[clause,[pred_arg_mod_structure,
prop(vce:passive, asp: [perfect],mod:none, tns:present),
[predicate(kill_1,agr(en_passive))],
[[subject, [nounphrase, index($0580$), agr(third,sing,masc) ]]]]]]]]],_28A8).
```

As a result of the call, _28A8 is instantiated to the following term:

```
[clause,[pred_arg_mod_structure,
prop(vce:active,asp:none, mod:none,tns:present),
[predicate(believe_1,agr(finite,present,sing,first))],
[[subject, [nounphrase,index($0398$),agr(first,sing), ppro(first,sing,_041C)]],
[object,[clause,[pred_arg_mod_structure,
prop(vce:passive, asp: [perfect],mod:none, tns:present),
[predicate(kill_1,agr(en_passive))],
[[subject, [nounphrase, index($0580$), agr(third,sing,masc) ]]]]]]]]
```

i.e. the same ! `Control` had no work to do, because the controller does not occur in the controlled clause, and so does not need to be ghosted. `Control` succeeded doing nothing on account of the following of its defining clauses:

```
control(X,X).
```

The next predicate to be applied is **`oraising`**. Since `control` has not modified the structure, we get the following call:

```
oraising([clause,[pred_arg_mod_structure,
prop(vce:active,asp:none, mod:none,tns:present),
[predicate(believe_1,agr(finite,present,sing,first))],
[[subject, [nounphrase,index($0398$),agr(first,sing), ppro(first,sing,_041C)]],
[object,[clause,[pred_arg_mod_structure,
prop(vce:passive, asp: [perfect],mod:none, tns:present),
[predicate(kill_1,agr(en_passive))],
[[subject, [nounphrase, index($0580$), agr(third,sing,masc) ]]]]]]]],_2910).
```

Oraising does change the structure. The defining clause that applies is the following:

```
oraising([H1,[pred_arg_mod_structure,Prop1,
        [predicate(Pred1,AgrPred1)],
    [Arg1,
     [object,[clause,
[pred_arg_mod_structure,Prop2,[predicate(P,agr(Agr))],
                  [[Subject|Rest]|Otherargs]]]]|R]]],

H1,
[pred_arg_mod_structure,Prop1,
       [predicate(Pred1,AgrPred1)],
    [Arg1,
     [object|Rest],
     [object,[clause,
[pred_arg_mod_structure,Propnew2,[predicate(P,agr(Agr))],
                  [Otherargs]]]]|R]]]):-

allsubject(Subject),
second_header(H1) ,
oraise(Pred1,Requires),
nonfinite(Agr),
Prop2 = prop(Voice,Aspect,Mod,Tns),
Propnew2 = prop(Voice,Aspect,Mod,tns:Requires).
```

Allsubject is, like **second_header**, a check on the environment of the rule. It requires that the variable named **Subject** cover a subject function. The code for **allsubject** is the following:

```
allsubject(S):- subject(S),!.
allsubject(S):- subject_active(S).

subject(subject).
subject(subject_pass).
subject_active(subject).
subject_active(subject_inf).
```

The call succeeds in the present case since **Subject** is instantiated to **subject**.

Next, the **oraise** predicate checks that the verb of the main clause belongs to the right class, and at the same time it instantiates the **Requires** feature, which will tell us whether what will remain of the controlled clause should be infinitive or gerundive. The call here is the following:

```
oraise(believe_1,X),
```

which succeeds and instantiates X to **to. Oraise** calls on the macro **m_verb** clause:

oraise(**Oraisingverb**,to):-
m_verb(**vinf**,_,**Oraisingverb**,_,_,_,_,_,_,_,_,_,_,
        [np(_,_,surf_object,_),
         np_vp(oblig,to_inf,object)]).

The **m_verb** responsible for the success of **oraise** is the lexical clause for the relevant reading of BELIEVE, namely:

```
m_verb(vinf,_,believe_1,believe,believe,believe,believes,believing,
        believed,believed,believed,trans,human,
        [np(oblig,posprec(1,Wnp),surf_object,_),
         np_vp(oblig,to_inf,object)]).

/* he believes him to teach linguistics */
```

Next is the check on the agreement value of the predicate in the controlled clause; it cannot be a finite clause, to which **subject to object raising** could not apply. In this case the call is:

```
nonfinite(en_passive)
```

which succeeds.

`Oraise` then proceeds to copy the **Property** field of the controlled clause to a new variable, but replaces the value for **tense** (of no application in non-finite clauses) with the **Requires** feature, i.e. **to**, which will indicate to the relevant clause of the generator that an **infinitive with to** vp should be generated as the remaining vp in the controlled clause.

Let us now look at the structure returned by **`oraising`**. We see that the subject of the controlled clause is moved to the controlling clause, and its function changed to object. The controlled clause is returned as second object, but now misses its subject. Finally, the new property field is assigned to the controlled clause, which is thereby untensed. In the case of our example sentence, the arglist for the matrix clause has the following two objects:

```
[object,[nounphrase,index($0580$), agr(third,sing), ppro(third, sing, masc))),
[object,[clause,[pred_arg_mod_structure,
prop(vce:passive,asp:[perfect], mod:none, tns:to],
[predicate(kill_1, agr(en_passive))],[[]]]]]]
```

Returning to **`prepgen`**, we find that there is another call to **`passive`**, this time on the matrix clause. This call will succeed trivially, since **`passive`** does not apply there:

```
passive([H1,[pred_arg_mod_structure,prop(vce:active,B,C,D)|R1]|R2],
         [H1,[pred_arg_mod_structure,prop(vce:active,B,C,D)|R1]|R2] ):-
second_header(H1),
!.
```

`Prepgen` has now finished, and the generator will take the results it has delivered and attempt to generate a string from them:

```
generate(Tree,Sentence):-
    prepgen(Tree,Treeprep),
    gen(Treeprep,List),
    flatten(List,Sentence).
```

Sentence will be instantiated to **[i,believe,him,to,have,been,killed]** and output as:

i believe him to have been killed.

Appendix A. Non-Standard Arity Prolog Predicates

This appendix briefly describes the pre-defined Prolog predicates that are specific to Arity Prolog (or at least non-standard) and have been made use of in **horatio** and/or **horgen**. Note that + in front of an argument indicates that it should be instantiated when the call to the predicate is made; a - sign indicates that the variable should still be uninstantiated.

Execution Control

[! P1, P2, ...!]: the snip symbol (**[!!]**) isolates code (a set of goals, the **P**s here) to be skipped in the event of backtracking. Example from **horatio** on page 69.

abort(1): with 1 as argument, `abort` returns to the operating system

ifthenelse(Condition,YesAction,NoAction): If **Condition** succeeds, **YesAction** is attempted; otherwise, **NoAction** is. **Ifthenelse** can be simulated by a disjunction of goals: `(Condition, Yesaction; not(Condition), NoAction).`Example from **horatio** on page 56.

String Manipulation

concat(+String1, +String2, -String3): **String3** results from the concatenation of **String1** and **String2.** Example from **horatio**:

```
concat(Output,$.ter$,Outterm),
concat(Output,$.lst$,Outlist),
```

used to produce the names of the output files: **.ter** (raw Prolog terms) and **.lst** (pretty-printed terms)

File Operations

create(-Handle, +Filename): a file named **Filename** is created and associated with handle **Handle.** Example from **horatio**:

```
dealwith(Input,_):-
        Input=stdin,
        create(HandleIn,'bidon'),
        close(HandleIn),!.
```

open(-Handle, +Filename, +Mode): Mode is **r** for reading, **w** for writing or **a** for appending; a file with name **Filename** is opened and associated with handle **Handle**, for the operation specified in **Mode**. Example from **horatio**:

```
dealwith(FileIn,HandleIn):-
        open(HandleIn,FileIn,r).
```

close(+Handle): the file associated with handle **Handle** is closed. Example from **horatio**:

```
dealwith(Input,_):-
        Input=stdin,
        create(HandleIn,'bidon'),
        close(HandleIn),!.
```

read_line(+Handle, -String): **Handle 0** is standard input; **String** is read as a one-line string from the input stream associated with **Handle** (the end of the string is indicated by a carriage return). **Read_line** offers minimal editing facilities (backspacing for deletion). Example from **horatio**:

```
  start(Outterm,Outlist,_,stdin):-
           open(Y,Outterm,w),
           open(Z,Outlist,w),
           repeat,
           nl,
           write($Please key in your sentence or stop. to stop$),
           nl,
           read_line(0,S),
           open(HandleIn,'bidon',w),  /* for writing */
           write(HandleIn,S),
           close(HandleIn),
           open(Handlenew,'bidon',r),    /* for reading */
           getsentence(Sentence,Handlenew),
           close(Handlenew),
           nl,
           write(Sentence),
           process(Sentence,Y,Z).
```

Appendix B. File Organization and Compilation Directives

File Organization

Both the parser (**horatio.exe**) and the generator (**horgen.exe**) result from compiling and linking several source files. The compiling and linking is achieved by calls to the **Arity Prolog Compiler** and the **Microsoft Linker** by two **.BAT** or **.CMD** files, i.e. batch files for MS-DOS or for OS/2. Except for the file name extension the files are the same for DOS and OS/2

Moratio.cmd

Moratio.cmd is an OS/2 batch file used to generate and link the object modules, to create **horatio.exe**, the executable file for the parser; **horatio.ari**, **lexatio1.ari**, **lexatio2.ari**, **lexatio3.ari** are text files containing the source code of the program; **horatio.ari** is the grammar proper, **lexatiox.ari** are the files holding the lexicon; the full commented source is available on the companion disk.

apc horatio,,/n

/* compiling the grammar, generating the data base */

/* **APC.EXE** is the **Arity Prolog compiler** */

/* **horatio.ari** contains the grammar for the parser */

/* it is compiled and produces **horatio.idb** as data base */

/* **/n** indicates that a new Prolog data base needs to be created; its name defaults to that of the source file, but the extension is **.idb** instead of **.ari**, the default file extension for Arity Prolog source code */

/* the object file's name defaults to **horatio.obj**, since the second argument is left empty (hence the two commas with nothing in between) */

apc lexatio1,,horatio

/* compiling the lexicon (part 1), adding to the same data base (last argument) */

apc lexatio2,,horatio /* idem (part 2) */
apc lexatio3,,horatio /* idem (part 3) */
link code lexio1 lexio2 lexio3 horatio lexatio1 lexatio2 lexatio3, horatio,, arity doscalls,,

//* linking the code modules; **horatio.obj**, **lexatio1.obj**, **lexatio2.obj**, **lexatio3.obj** result from the compiling process carried out in the preceding lines by the Arity Prolog compiler. **Code** is part of the Arity Prolog delivery and is to be used as the first object module for all Arity Prolog applications; **lexio1** to **3** are cloned (with the utility **clone.exe**, also part of the Arity Prolog delivery) so as to be able to make use of **far Prolog**. **Arity** and **doscalls** are the libraries. Under DOS there is no need for the **doscalls** library. The second argument is the name of the **.exe** file; here it is **horatio.exe** */

Morgen.cmd

/* similar to MORATIO.CMD; the first lexicon file (**lexgen1.ari**) differs slightly from the one used in analysis, namely **lexatio1.ari**; the executable file for the generator is called **horgen.exe** */

apc horgen,,/n
apc lexgen1,,horgen
apc lexatio2,,horgen
apc lexatio3,,horgen
link code lexio1 lexio2 lexio3 horgen lexgen1 lexatio2 lexatio3, horgen,, arity doscalls,,

Arity Prolog Compilation Directives

When several files are meant to be compiled and linked together to yield a single executable file, the first code file to be compiled needs a declaration for a predicate called **main**, of zero arity and to be declared **public**. We therefore find in the file **horatio.ari**:

:- public main/0.

The **`main`** predicate is itself defined by a call to the **`go`** predicate, which provides the GO step.

All lexical clauses in both the parser and the generator are declared **external** and **far**. For instance, in **horatio.ari**, we have compilation directives such as

:- extrn verb/10:far.

This means that the predicate **verb** is to be found in an **external** file (*in casu* **lexatio2.ari**), has **arity** 10 (needs 10 **arguments**), and is to be compiled as **far**[18] code.

In the file **lexatio2.ari**, we find the following compilation directive

:- public verb/10:far.

making the predicate **`verb`** public, i.e. callable from other modules.

Besides, each module containing **far** code needs to have its own segment declaration. For instance, in **lexatio2.ari** we find

```
:- segment(lexio2).
```

Lexio2.obj is obtained by an appropriate call on the **clone.exe** program, part of the Arity package, namely

```
clone lexio2
```

Appendix C. Input and Output

Input

In **horatio** input is either from the keyboard (standard input, **stdin** in **horatio** code), or from a text file. In the latter case, the text must be pure **ASCII**, with one sentence per line, the end of the file being marked by a line containing only the word **stop**.

In the former case, the input is read by the predicate **`read_line`** (which provides minimal editing facilities, such as the use of **backspace** to delete characters already entered at the keyboard) and saved into a dummy file called **bidon**. Then the dummy file is opened and input proceeds in the way described in the previous paragraph.

The **`read_line`** predicate is an Arity Prolog extension to standard Prolog. It takes two arguments: the first is the read **handle**, which is 0 in the case of standard input. The second is the string to be read from the input stream associated with the handle. The string is read as a single line.

The next job in input is to convert the string to a word list. This is achieved by a standard **string to word list converter** program. We use the one given in **Bratko 1990**.

The resulting word list is then submitted to the parsing process.

It should be noted that in **horatio** input is within a loop. We keep reading in sentences until we hit the word list **[stop]**, resulting from the end-of-file marker **stop**. When we do, we call the **`abort`** predicate to return to the operating system.

In **parsing** we need to get at all the possible parses in the case of ambiguous input. This is why we **`fail`** at the end of the parsing process for a given sentence, whereas in generation we succeed (the generator does not backtrack once it has produced a string corresponding to a given parse).

In the generator **horgen** input is never from the keyboard, as the expected input is a parse. Such parses are read in with the standard Prolog **`read`** predicate, which is able to read any well-formed Prolog term.

Output

In **horatio** we need to produce two types of output:

1) **raw** objects: these are the Prolog **terms** which result from the parsing process. They are feedable to the

generator **horgen** (after a transformation into strings of the uninstantiated variables in the **index** functors - this transformation is described above, on page 107). Such parses go into a file with a **.ter** extension (for terms). The file name is created by concatenating the extension **.ter** to the base file name elicited from the user. We make sure that the parses end with a dot, so that they can be read in by the `read` predicate in the generator.

2) **pretty-printed** objects: these are meant to be examined by the grammar-designer for debugging and other purposes. They are collected into a file which gets an **.lst** extension. The pretty-printer used is elementary and standard. It is a slight adaptation of the one to be found in **Clocksin and Mellish 1981** (the adaptation concerns the pretty-printing of uninstantiated variables).

horgen produces strings as output. It is a trivial extension to add the capability of capitalizing the first word of the output, as well as proper names occurring in the string. It is equally trivial to ensure that questions end with a question mark. These extensions are left as exercises for the reader.

Appendix D. Selective Lexicon Downloading

Getvoc.awk

```
BEGIN {RS="."}

{ if (FILENAME==ARGV[1])
      for (i=1;i<=NF;i++)  x[tolower($i)]= tolower($i)
}

{ if (FILENAME==ARGV[2]) {  RS= "@";FS=",";
                      for (i=2;i<=10;i++)
                              {
                              if ( tolower($i) in x &&  !($0 in y))
                                        {print  > "voc.ari"
                                         y[$0]="in"
                                         }
                              }
                    }
}
```

The first argument is the text to be parsed, the second the lexicon to be searched. In the first file (the text to be parsed) we use a period (.) as record separator (**RS** is set to dot). In the second file (the lexicon to be used) we separate records with an **@**, which we therefore assign as new value for **RS**.

The individual words are saved into the **x** array, and the lexicon is then searched for matches in the lexeme and morphological variant fields. The selected entries are saved to a file, **voc.ari**; the second condition in the last **IF clause** prevents repetitions in the **voc.ari** file. A batch procedure will then take care of the compilation of **voc.ari**, and of the linking of the resulting object code with that produced for the grammar files, in order to build the required executable. Under OS/2, the batch file is the following:

```
echo %1. %1.  >filin
awkl -f getvoc.awk %1 lex.ari
copy lexdcl.ari+voc.ari lexsel.ari
apc lexsel,,horatio
link code lexio1 lexio2 horatio lexatio1 lexsel , horex,,arity doscalls,,
horex < filin
```

(**%1** stands for the text file; **awkl** invokes the large model of **awk** in the **MKS** OS/2 implementation[19];

lex.ari houses the complete **horatio** lexicon; **lexdcl.ari** contains the necessary compiler declarations for Arity Prolog; **apc** calls the Arity Prolog compiler; the linker (**link**) produces the executable file **horex**, which is fed the name of the text file twice, so that it can produce a **%1.lst** and a **%1.ter** file, the latter containing the raw objects, and the former the pretty-printed ones).

Appendix E. Importing Lexical Entries from Ldoce

Introduction

It should be stated from the outset that a certain familiarity with at least the published version of LDOCE is presupposed in this appendix. A whole book (namely **Boguraev and Briscoe 1989**) has recently been devoted to computer applications of this dictionary data base, and the reader is referred to the first four chapters of that book, and the references given there, for more information on LDOCE, particularly on its grammar coding system. The published version of LDOCE contains explanatory prefatory material and a table of the grammar codes used in the dictionary, but does not explain the semantic or field codes used in the computer tape.

There is now a lot of interest in the reuse of available lexical resources. However, much of the available literature on the topic is speculative and programmatic in nature. This appendix, on the contrary, is unashamedly pragmatic. It reports on an experiment that has been carried out, going down to the level of specifying the queries on the data base and the **awk**[20] programs used to reformat the imported data.

The application programs for retrieving material from the lexical data base and the **awk** programs all run on a PC under MS-DOS.

Description of the Liège Ldoce Data Base

LDOCE in Liège is in relational data base format. The retrieval software is made up of a series of application programs written in **Clipper**, whose data definition and query language can be defined as an enhanced **dBase** language. The next section looks at the major tables and their fields. The data base design and the transformation of the LDOCE computer tape into the relational tables was carried out by Jacques Jansen. The application programs were written by the author.

Ldoce Data Base Design

Lemma Data Base: Coword

Field Name	Type	Nature	Width
ENTRYKEY	Char	shared by all LDOCE-derived dbf as a link to this dbf	4
HEADCLAS	Char	simple, compound or run-on: S,C,R	1

DEFINUM	Char	for run-on entries: link to the relevant definition number in the main entry	2
VARISUF	Char	flag: is there a variable suffix? (e.g.. ic/ical, ise/ize)	1
FLAGCV	Char	included in LDOCE's controlled vocabulary ?	1
FLAGIF	Char	does final consonant redoubling apply (red -> redder)?	1
POS	Char	Part of speech: up to five different POS, each coded in one byte	5
NOTE	Char	reference to grammatical note	3
WORD	Char	lemma	34

Structure for ncoword.dbf

Ncoword is similar to **coword**, but also includes lexicographical WEIGHT of the lemma, measured in terms of number of definitions, examples, different grammatical codes and idiomatic structures associated with the entry.

DEFNU	Num	Number of definitions associated with the entry	2
IDIONU	Num	Number of idioms associated with the entry	2
EXNU	Num	Number of examples associated with the entry	3
GRNU	Num	Number of grammatical codes associated with the entry	3

Definition Data Base: Codefi

Field Name	Type	Nature	Width
ENTRYKEY	Char	cf coword	4
HEADCLAS	Char	**I** identifies an idiomatic structure	1
DEFINUM	Char	sequential number of the definition	2
DEFILET	Char	small letter associated with a given definum: a, b, c, ...	1
DEFIMAT	Char	field codes: a four-byte field; codes the subject matter(s) (1 or 2) to which the lexical item under the given definition belongs	4
DEFISEM	Char	semantic codes: a ten-byte field; codes such properties as the semantic requirements to be placed on the deep subject (byte 5) and objects of verbs (bytes 10 and 8); also codes inherent semantic properties for nouns and selection restrictions for adjectives (byte 5)	10
LINECNT	Char	sequential value of definition line (each defiline is 76 char long, and several may be needed to cover a	1

		single definition; the @ sign identifies the first line)	
DEFILINE	Char	text of one line of a given definition	76

Idiom Data Base: Coidio

Field Name	Type	Nature	Width
ENTRYKEY	Char	cf coword	4
HEADCLAS	Char	**I** identifies idioms or idiomatic structures	1
DEFINUM	Char	cf codefi	2
VARISUF	Char	cf coword	1
IDIOM	Char	text of the idiom	60

Example Data Base: Coexam

Examples are associated with definitions and idioms

Field Name	Type	Nature	Width
ENTRYKEY	Char	cf coword	4
HEADCLAS	Char	cf codefi	1
DEFINUM	Char	cf codefi	2

DEFILET	Char	cf codefi	1
EXAMNUM	Char	sequential number of the example (associated with a given word sense or idiom)	2
DCODNUM	Char	hand-coded: relation between example and grammatical code	2
LINECNT	Char	cf codefi	1
EXAMLINE	Char	line of example text	76

Grammatical Code Data Base: Codcod

Field Name	Type	Nature	Width
ENTRYKEY	Char	cf coword	4
HEADCLAS	Char	cf codefi	1
DEFINUM	Char	cf codefi	2
DCODNUM	Char	sequential number of grammatical code associated with a given definition	2
LEFTLINK	Char	type of link: **optional**, **'esp**.', **obligatory**; this field codes the cohesiveness of the link between the lexical item and its lexical left environment	1
LEFTTYPO	Char	type of font: **italic**, **bold**,...: this field can be used to retrieve the nature of the coded lexical left environment; for instance, prepositions will be coded **italic** and adverbial particles **bold**	1
LEFTCTX	Char	word or word group specified as lexical left environment	3
GRAMCODE	Char	three-byte grammatical code; consult the LDOCE Table of Codes (in the printed form of the dictionary)	3
RGHTLINK	Char	cf leftlink	1
RGHTTYPO	Char	cf lefttypo	1
RGHTCTX	Char	cf leftctx	3
GRAMCOMM	Char	grammatical comment (partly formalized)	3

It might be useful to look at the representation of a given lexical item in the Liège LDOCE data base, so that the reader can build for himself a more concrete picture of what the data base looks like. I have taken the entry for the verb **believe**, because it is short and illustrates the results of the decompaction process that has been applied to the LDOCE grammar fields, a process that needed to be carried out if the data base was to prove usable for the retrieval of grammatical information, as illustrated in this appendix. The need for a decompaction procedure is also discussed in Michiels 1982.

Believe in Ldoce

believe / pron / *v* [Wv6] **1** [I0] to have a firm religious faith **2** [T1] to consider to be true or honest: *to believe someone* | *to believe someone's reports* **3** [T5a,b;V3;X (*to be*) 1, (*to be*) 7] to hold as an opinion; suppose: *I believe he has come.* | *He has come, I believe.*| *"Has he come?" "I believe so"* | *I believe him to have done it.* | *I believe him (to be) honest* -- see unbelief (USAGE)

In the Liège data bases:

In COWORD

Entrykey	B@ZV
Headclas	S
Flagcv	1
Flagif	0
Pos	v
Word	believe

In CODCOD

Entrykey	B@ZV	B@ZV	B@ZV	B@ZV	B@ZV	B@ZV	B@ZV
Headclas	S	S	S	S	S	S	S

Definum	1	2	3	3	3	3	3
Dcodnum	1	1	1	2	3	4	5
Gramcode	I0	T1	T5a	T5b	V3	X1e	X7e

In CODEFI

Entrykey	B@ZV	B@ZV	B@ZV
Headclas	S	S	S
Definum	1	2	3
Defimat	RLRN[21]	....	....
Defisem	....H.....[22]	....H....X[23]	....H....T[24]
Linecnt	@	@	@
Defiline	to have a firm re-ligious faith	to consider to be true or honest	to hold as an opinion, suppose

In COEXAM

(first three examples only)

Entrykey	B@ZV	B@ZV	B@ZV
Headclas	S	S	S
Definum	2	2	3
Examnum	1	2	1
Dcodnum	1		1

Linecnt	@	@	@
Examline	to believe someone	to believe some-one's reports	I believe he has come

A good deal of criticism has been leveled recently against the use of the relational data base model for the implementation of lexical data bases. It is pointed out that the restrictions of the relational model (fixed number of fields, fixed field length) make it extremely difficult to implement a lexical data base in such a model. For instance, lexical items have different number of homographs; homographs have different numbers of associated definitions; definitions have different numbers of associated examples; definitions and examples are of varying length, from a single word to a full sentence. However, it is in the very essence of the relational model to work with a series of tables, rather than a single one. Consequently, the fact that one definition has one example, whereas the following has six, is not really a problem if the definition table and the example table are distinct tables related by one or several common fields, as they are in the Liège LDOCE data base.

Clearly, efficiency considerations make the relational model unusable for real time retrieval in NLP tasks such as machine translation. But then again, the full power of the relational model is not necessarily needed in such cases, as retrieval is most often to be carried out on the basis of the morphological variant or lemma, and not on the basis of the complex queries on grammatical, semantic and field code information that the relational model can handle. The power of the latter, however, can be put to good use in a lexicographer's workbench.

The Importation Process from Ldoce into Horatio's Lexicon

In its present state of development, **horatio** works with 18 verb classes. The formats of the classes selected for the importation experiment are briefly described below. For each verb class, the filter used on the Liège LDOCE data base is described, as well as the raw data from the Liège data base and the AWK program used to convert to the format used by **horatio** .

An essential feature of **horatio** is that it goes down to the level of **reading**, i.e. a selected interpretation of a lexical item according to properties of its environment. Such a level of delicacy is crucial for most NLP tasks, machine translation being a prime example. As can be guessed from the sample entries reproduced in this appendix, the importation from LDOCE concerns mainly entries beginning with **a**, and no effort has been made to select entries on the basis of frequency, lexicographical weight or membership of a domain-related sublanguage (lexicographical weight could have been assessed on the basis of the information contained in **ncoword** and domain-relatedness captured by means of the subject field codes in **defimat** in **codefi**).

We shall concentrate on showing how the **m_verb** clauses for various verb classes (indicated by the first argument of the **m_verb** clause) can be derived fom LDOCE by a three-step process:

a) obtain the relevant material from LDOCE (this is accomplished by EXPORT, a Clipper application which enables the linguist to select lexical entries - at the reading level - according to entry-level, semantic and grammatical properties; irregular inflectional morphology is taken care of by retrieval from a specialized data base; regular inflectional morphology is generated by a procedure belonging to EXPORT)

b) reformat the material obtained in a) by means of the appropriate AWK program

c) hand-check the results (the assignment of LDOCE semantic codes is often debatable; inflectional variants also have to be checked)

Intransitive Verbs

A. FILTERS ON LDOCE

on ENTRIES (**coword**)

'v' $ pos .and. headclas ="S"

(the **POS** field **includes** (operator **$**) "v" - i.e. it is a verb - and **headclas** is "**S**"- i.e. it is not a compound, multi-word unit or a run-on entry - "**C**" or "**R**" values)

on SEMANTICS (**codefi**)

headclas # "I"

Headclas must be **different from** (# operator) "I", i.e. must not be an idiom.

on GRAMMAR (**codcod**)

gramcode= "I0 " .and. rghtlink = " "

Gramcode is a three-byte code; **rghtlink** gives the nature of the link with a lexical element attached to the item on the right. By specifying an empty **rghtlink** we make sure that we do not select prepositional or phrasal verbs (for which see below, p. 138 and 141).

B. RAW IMPORTED MATERIAL

As pointed out above, the application programs include a rather elementary generator of inflectional variants; this generator has access to a data base of irregular forms and uses the flag for consonant redoubling of **coword** (**FLAGIF** field - see above, p. 126).

(Rnu = LDOCE reading number)

```
LEX          Vs      Ving     Ved    Ven    Pos  Gr   Sem    Rnu
abate        abates abating abated abated v     I0  ....T.....  1
DEF   (of winds, storms, disease, pain, etc.) to become less strong; decrease
EX    The ship waited till the storm abated before sailing out to sea

abrade       abrades abrading abraded abraded v     I0  ....Z....5  1a
DEF   (esp. of skin) to wear away by hard rubbing; to cause (esp. skin) to wear away by hard rubbing

accelerate   accelerates accelerating accelerated accelerated v     I0  ....Z....Z  1
DEF   to (cause to) move faster

accept       accepts accepting accepted accepted v     I0  ....H....Z  1
DEF   to take or receive (something offered or given), esp. willingly; receive with favour
EX    I cannot accept your gift. He asked her to marry him and she accepted (him)
```

C. AWK PROGRAM TO CONVERT RAW MATERIAL TO HORATIO FORMAT

The **if**-clauses test the value of the fifth byte in the 10-byte field allocated to the semantic codes; this byte houses the restrictions on the subject: **J** codes "**Movable and Solid**", deemed to roughly correspond to the feature "**thing**" in **horatio**; **H** is "**human**" and **T** is "**abstract**"; absence of semantic restriction on the subject is coded in **horatio** by means of Prolog's **anonymous variable (_)**.

```
BEGIN {nu=1}

# nu is the entry counter

{  semsubj="_"

  # semsubj is assigned the anonymous variable unless J, H or T is found at the appropriate place in
  # the semantic info field

  if (substr($8,5,1)=="J") semsubj = "thing"
  if (substr($8,5,1)=="H") semsubj = "human"
  if (substr($8,5,1)=="T") semsubj = "abstract"

  if (($1 ~ /DEF/ || $1 ~ /EX/) && NF>1)
       printf("/* %s */\n",$0)

  # the first part of the IF clause deals with definition and example lines; these appear as comments
  # in the Prolog clauses

    else
       { if (NF>1)
       {printf("\n/* NU %d */\n",nu)
       printf("m_verb(verbintr,%s_%s,%s,%s,%s,%s,%s,\n%s,%s,%s,intrans,%s,\n[]).\n",
       $1,$9,$1,$1,$1,$2,$3,$4,$4,$5,semsubj)

            # $1 = Lex
            # $9 = Rnu
            # $2 = Vs
            # $3 = Ving
            # $4 = Ved
            # $5 = Ven

       nu++}}
  }
```

D. SAMPLE ENTRIES IMPORTED FROM LDOCE

```
/* NU 1 */
m_verb(verbintr,abate_1,abate,abate,abate,abates,abating,
abated,abated,abated,intrans,abstract,
[]).
/* DEF   (of winds, storms, disease, pain, etc.) to become less strong; decrease */
/* EX    The ship waited till the storm abated before sailing out to sea */
```

Note

/* The **T** feature transcribed as **abstract** is made more precise by the list of typical subjects headed by **OF**; such lists are best read as lists of **thesauric heads**; they are hard to make use of */

```
/* NU 2 */
m_verb(verbintr,abrade_1a,abrade,abrade,abrade,abrades,abrading,
abraded,abraded,abraded,intrans,_,
[]).
/* DEF   (esp. of skin) to wear away by hard rubbing; to cause (esp. skin) to wear away by hard rubbing
*/
```

Note

/* The **Z** value transcribed as the **anonymous variable** is hardly what we need: the selectional restriction is very strong, but is once again buried in the OF list */

```
/* NU 3 */
m_verb(verbintr,accelerate_1,accelerate,accelerate,accelerate,accelerates,accelerating,
accelerated,accelerated,accelerated,intrans,_,
[]).
/* DEF   to (cause to) move faster */

/* NU 4 */
m_verb(verbintr,accept_1,accept,accept,accept,accepts,accepting,
accepted,accepted,accepted,intrans,human,
[]).
/* DEF   to take or receive (something offered or given), esp. willingly; receive with favour */
/* EX    I cannot accept your gift. He asked her to marry him and she accepted (him) */
```

The main problem with intransitive verbs is how to distinguish them from transitive verbs that can have a zero (understood) object, either definite or indefinite.

Mono-Transitive Verbs

A. FILTERS ON LDOCE

ENTRY LEVEL	**'v' $ pos .and. headclas = "S"**
SEMANTICS	**headclas # "I"**
GRAMMAR	**gramcode = "T1 " .and. rghtlink = " "**

B. RAW IMPORTED MATERIAL

```
abandon          abandons abandoning abandoned abandoned v    T1  ....H....T  1
DEF   to leave completely and for ever; desert
EX    The sailors abandoned the burning ship

abandon          abandons abandoning abandoned abandoned v    T1  ..D.H....H  2
DEF   to leave (a relation or friend) in a thoughtless or cruel way
EX    He abandoned his wife and went away with all their money

abandon          abandons abandoning abandoned abandoned v    T1  ....H....T  3
DEF   to give up, esp. without finishing
EX    The search was abandoned when night came, even though the child had not been found

abate          abates abating abated abated v    T1  ....T....T  2
DEF   lit to make less
EX    His pride was not abated by his many mistakes

abate          abates abating abated abated v    T1  ....H....T  3
```

DEF   law to bring to an end (esp. in the phr. abate a nuisance)

C. AWK PROGRAM

Note that byte 10 in **defisem** (field 8) codes the semantic restriction on the **object** of a transitive verb.

```
BEGIN {nu=1}
{  semsubj="_"
   if (substr($8,5,1)=="J") semsubj = "thing"
   if (substr($8,5,1)=="H") semsubj = "human"
   if (substr($8,5,1)=="T") semsubj = "abstract"

   semobj = "_"
   if (substr($8,10,1)=="J") semobj = "thing"
   if (substr($8,10,1)=="H") semobj = "human"
   if (substr($8,10,1)=="T") semobj = "abstract"

   if (($1 ~ /DEF/ || $1 ~ /EX/) && NF>1)
          printf("/* %s */\n",$0)
      else
          { if (NF>1)
          {printf("\n/* NU %d */\n",nu)
          printf("m_verb(verbtr,%s_%s,%s,%s,%s,%s,%s,\n%s,%s,%s,trans,
%s,\n[np(oblig,posprec(1,Wnp),object,%s)]).\n",$1,$9,$1,$1,$1,$2,$3,$4,$4,$5,semsubj,semobj)
          nu++}}
   }
```

D. SAMPLE ENTRIES IMPORTED FROM LDOCE

```
/* NU 1 */
m_verb(verbtr,abandon_1,abandon,abandon,abandon,abandons,abandoning,
abandoned,abandoned,abandoned,trans,human,
[np(oblig,posprec(1,Wnp),object,abstract)]).
/* DEF   to leave completely and for ever; desert */
/* EX    The sailors abandoned the burning ship */
```

Note

/* The **T** value giving rise to the **abstract** value results from a miscoding on the part of the LDOCE lexicographers */

```
/* NU 2 */
m_verb(verbtr,abandon_2,abandon,abandon,abandon,abandons,abandoning,
abandoned,abandoned,abandoned,trans,human,
[np(oblig,posprec(1,Wnp),object,human)]).
/* DEF   to leave (a relation or friend) in a thoughtless or cruel way */
/* EX    He abandoned his wife and went away with all their money */

/* NU 3 */
m_verb(verbtr,abandon_3,abandon,abandon,abandon,abandons,abandoning,
abandoned,abandoned,abandoned,trans,human,
[np(oblig,posprec(1,Wnp),object,abstract)]).
/* DEF   to give up, esp. without finishing */
/* EX    The search was abandoned when night came, even though the child had not been found */
/* NU 6 */
m_verb(verbtr,abate_2,abate,abate,abate,abates,abating,
abated,abated,abated,trans,abstract,
[np(oblig,posprec(1,Wnp),object,abstract)]).
/* DEF   lit to make less */
/* EX    His pride was not abated by his many mistakes */

/* NU 7 */
m_verb(verbtr,abate_3,abate,abate,abate,abates,abating,
abated,abated,abated,trans,human,
[np(oblig,posprec(1,Wnp),object,abstract)]).
/* DEF   law to bring to an end (esp. in the phr. abate a nuisance) */
```

Note
/*The **T** feature is again much too general */

Prepositional Verbs

Examples: **look at**, **listen to**: strongly bound preposition

A. FILTERS ON LDOCE

ENTRY **'y' $ pos .and. headclas ="C"**

(**'y'** is the POS assigned to prepositional verbs; **headclas** is now "**C**", i.e. **compound** (more than one word))

SEM **headclas # "I"**

GRAMMAR **substr(gramcode,1,2) ="T1"**

B. RAW IMPORTED MATERIAL

```
abide by        abides abiding abided abided y    T1  ....H....T  1
DEF   to be faithful to; obey (laws, agreements, etc.)
EX    If you join the club you must abide by its rules
abide by        abides abiding abided abided y    T1  ....H....T  2
DEF   to wait for or accept
EX    You must abide by the results of your mistakes
account for     accounts accounting accounted accounted y    T1  ....H...YT  1  -I to
DEF   to give a statement showing how money or goods left in one's care have been dealt with
EX    He has to account to the chairman for all the money he spends
account for     accounts accounting accounted accounted y    T1  ....H....T  2  -I to
DEF   to give an explanation or reason for
EX    He could not account for his foolish mistake
account for     accounts accounting accounted accounted y    T1  ..I.H....O  3
DEF   infml to kill, shoot, or catch
EX    I think I accounted for 3 of the attackers
```

C. AWK PROGRAM

This program generates two clauses for each selected reading, in accordance with Quirk et al.'s concept of **multiple analysis** for such verbs, to which the reader is referred (see **Quirk et al. 1985**, §§ 2.61, 16.5 and here p. 19).

```
BEGIN {nu=1}
{  semsubj="_"
   if (substr($9,5,1)=="J") semsubj = "thing"
   if (substr($9,5,1)=="H") semsubj = "human"
   if (substr($9,5,1)=="T") semsubj = "abstract"

   semobj = "_"
   if (substr($9,10,1)=="J") semobj = "thing"
   if (substr($9,10,1)=="H") semobj = "human"
   if (substr($9,10,1)=="T") semobj = "abstract"

   if (($1 ~ /DEF/ || $1 ~ /EX/) && NF>1)
          printf("/* %s */\n",$0)
      else
          { if (NF>1)
          {printf("\n/* NU %d */\n",nu)
          printf("m_verb(vtrprep,%s_%s_%s,%s,%s,%s,%s,%s,\n%s,%s,%s,trans,
%s,\n[pp(oblig,posprec(1,Wpp),pp_arg,%s,_,%s)]).\n",
          $1,$2,$10,$1,$1,$1,$3,$4,$5,$5,$6,semsubj,semobj,$2)

# $2 is the preposition
 printf("m_verb(vtrprep,%s_%s_%s,%s,%s,%s,%s,%s,\n%s,%s,%s,trans,%s,\n
[string(oblig,posprec(1,0),[%s]),\nnp(oblig,posprec(2,Wnp),object,%s)]).\n",
          $1,$2,$10,$1,$1,$1,$3,$4,$5,$5,$6,semsubj,$2,semobj)
          nu++}}
   }
```

D. SAMPLE ENTRIES IMPORTED FROM LDOCE

```
/* NU 1 */
m_verb(vtrprep,abide_by_1,abide,abide,abide,abides,abiding,
abided,abided,abided,trans,human,
[pp(oblig,posprec(1,Wpp),pp_arg,abstract,_,by)]).
m_verb(vtrprep,abide_by_1,abide,abide,abide,abides,abiding,
abided,abided,abided,trans,human,
[string(oblig,posprec(1,0),[by]),
np(oblig,posprec(2,Wnp),object,abstract)]).
/* DEF   to be faithful to; obey (laws, agreements, etc.) */
/* EX    If you join the club you must abide by its rules */
```

Note
/* Here too the specification of the thesauric type of the object is much more precise than the **T** code */

```
/* NU 2 */
m_verb(vtrprep,abide_by_2,abide,abide,abide,abides,abiding,
abided,abided,abided,trans,human,
[pp(oblig,posprec(1,Wpp),pp_arg,abstract,_,by)]).
m_verb(vtrprep,abide_by_2,abide,abide,abide,abides,abiding,
abided,abided,abided,trans,human,
[string(oblig,posprec(1,0),[by]),
np(oblig,posprec(2,Wnp),object,abstract)]).
/* DEF   to wait for or accept */
/* EX    You must abide by the results of your mistakes */

/* NU 5 */
m_verb(vtrprep,account_for_1,account,account,account,accounts,accounting,
accounted,accounted,accounted,trans,human,
[pp(oblig,posprec(1,Wpp),pp_arg,abstract,_,for)]).
m_verb(vtrprep,account_for_1,account,account,account,accounts,accounting,
accounted,accounted,accounted,trans,human,
[string(oblig,posprec(1,0),[for]),
np(oblig,posprec(2,Wnp),object,abstract)]).
/* DEF   to give a statement showing how money or goods left in one's care have been dealt with */
/* EX    He has to account to the chairman for all the money he spends */

/* NU 6 */
m_verb(vtrprep,account_for_2,account,account,account,accounts,accounting,
accounted,accounted,accounted,trans,human,
[pp(oblig,posprec(1,Wpp),pp_arg,abstract,_,for)]).
m_verb(vtrprep,account_for_2,account,account,account,accounts,accounting,
accounted,accounted,accounted,trans,human,
[string(oblig,posprec(1,0),[for]),
np(oblig,posprec(2,Wnp),object,abstract)]).
/* DEF   to give an explanation or reason for */
/* EX    He could not account for his foolish mistake */

/* NU 7 */
m_verb(vtrprep,account_for_3,account,account,account,accounts,accounting,
accounted,accounted,accounted,trans,human,
[pp(oblig,posprec(1,Wpp),pp_arg,_,_,for)]).
m_verb(vtrprep,account_for_3,account,account,account,accounts,accounting,
accounted,accounted,accounted,trans,human,
[string(oblig,posprec(1,0),[for]),
np(oblig,posprec(2,Wnp),object,_)]).
/* DEF   infml to kill, shoot, or catch */
/* EX    I think I accounted for 3 of the attackers */
```

Transitive Phrasal Verbs

Examples: **look up**, **put off**: transitive phrasal verbs: *look it up*, **look up it*

A. FILTERS ON LDOCE

ENTRY **'z' $ pos .and. headclas = "C"**

(**'z'** is the **pos** associated with phrasal verbs in LDOCE)

SEM **headclas # "I"**

GRAMMAR **substr(gramcode,1,2) ="T1"**

B. RAW IMPORTED MATERIAL

```
act out          acts acting acted acted z     T1  ....H....T  1
DEF   to express (thoughts, unconscious fears, etc.) in actions and behaviour rather than in words
add up           adds adding added added z     T1  ....H....T  2
DEF   to add (numbers) together to get a total
```

C. AWK PROGRAM

```
BEGIN {nu=1}
```

```
{  semsubj="_"
   if (substr($9,5,1)=="J") semsubj = "thing"
   if (substr($9,5,1)=="H") semsubj = "human"
   if (substr($9,5,1)=="T") semsubj = "abstract"

   semobj = "_"
   if (substr($9,10,1)=="J") semobj = "thing"
   if (substr($9,10,1)=="H") semobj = "human"
   if (substr($9,10,1)=="T") semobj = "abstract"

   if (($1 ~ /DEF/ || $1 ~ /EX/) && NF>1)
          printf("/* %s */\n",$0)
      else
          { if (NF>1)
          {printf("\n/* NU %d */\n",nu)
          printf("m_verb(vphr,%s_%s_%s,%s,%s,%s,%s,%s,\n%s,%s,%s,trans,
%s,\n[part(oblig,posprec(1,2),[%s]),\nnp(oblig,posprec(1,Wnp),object,%s)]).\n",
          $1,$2,$10,$1,$1,$1,$3,$4,$5,$5,$6,semsubj,$2,semobj)
          nu++}}
   }
```

D. SAMPLE ENTRIES IMPORTED FROM LDOCE

```
/* NU 1 */
m_verb(vphr,act_out_1,act,act,act,acts,acting,
acted,acted,acted,trans,human,
[part(oblig,posprec(1,2),[out]),
np(oblig,posprec(1,Wnp),object,abstract)]).
/* DEF   to express (thoughts, unconscious fears, etc.) in actions and behaviour rather than in words */

/* NU 2 */
m_verb(vphr,add_up_2,add,add,add,adds,adding,
added,added,added,trans,human,
[part(oblig,posprec(1,2),[up]),
np(oblig,posprec(1,Wnp),object,abstract)]).
/* DEF   to add (numbers) together to get a total */
```

Note
/* In both these entries the bracketed material in the definition is again much more precise than the semantic code; the two cases are different, however: in **act_out,** the list is open-ended and has exemplificatory value only; in **add_up**, the one-member list is best read as head of a thesauric class */

Verbs Taking an Object and an Object Complement

A. Object complement is a noun phrase

Example: **consider**: *he considers the teacher a genius*

A. FILTERS ON LDOCE

ENTRY	**'v' $ pos .and. headclas = "S"**
SEM	**headclas # "I"**
GRAMMAR	**gramcode="X1 " .or. gramcode="X1e"**

(in the Liège LDOCE data base, **X1e** corresponds to **X (to be) 1** in the printed version of LDOCE)

B. RAW IMPORTED MATERIAL

```
acclaim            acclaims acclaiming acclaimed acclaimed v     X1  ....H..T.H  2
DEF   to declare to be or publicly recognize as, esp. with loud shouts of approval or praise
EX    They acclaimed him as the best writer of the year. They acclaimed her their leader
account            accounts accounting accounted accounted v     X1  ....H....Z  1
DEF   to consider
EX    He was accounted a wise man. He accounted himself lucky to be alive
acknowledge         acknowledges acknowledging acknowledged acknowledged v     X1e ....H....H  2
DEF   to recognize, accept, or admit (as)
EX    He was acknowledged to be the best player. He was acknowledged as their leader. They
acknowledged themselves (to be) defeated
```

C. AWK PROGRAM

```
BEGIN {nu=1}
{  semsubj="_"
   if (substr($8,5,1)=="J") semsubj = "thing"
   if (substr($8,5,1)=="H") semsubj = "human"
   if (substr($8,5,1)=="T") semsubj = "abstract"

   semobj = "_"
   if (substr($8,10,1)=="J") semobj = "thing"
   if (substr($8,10,1)=="H") semobj = "human"
   if (substr($8,10,1)=="T") semobj = "abstract"

 # semattr houses restrictions on the object complement
   semattr = "_"
   if (substr($8,8,1)=="J") semattr = "thing"
   if (substr($8,8,1)=="H") semattr = "human"
   if (substr($8,8,1)=="T") semattr = "abstract"

   if (($1 ~ /DEF/ || $1 ~ /EX/) && NF>1)
          printf("/* %s */\n",$0)
      else
          { if (NF>1)
          {printf("\n/* NU %d */\n",nu)
          printf("m_verb(vcomp,%s_%s,%s,%s,%s,%s,%s,\n%s,%s,%s,trans,
%s,\n[np(oblig,posprec(1,Wnp),object,%s),\nnp(oblig,posprec(2,Wnp2),object_attribute,%s)]).\n",
          $1,$9,$1,$1,$1,$2,$3,$4,$4,$5,semsubj,semobj,semattr)
          nu++}}
   }
```

D. SAMPLE ENTRIES IMPORTED FROM LDOCE

```
/* NU 1 */
m_verb(vcomp,acclaim_2,acclaim,acclaim,acclaim,acclaims,acclaiming,
acclaimed,acclaimed,acclaimed,trans,human,
[np(oblig,posprec(1,Wnp),object,human),
np(oblig,posprec(2,Wnp2),object_attribute,abstract)]).
/* DEF   to declare to be or publicly recognize as, esp. with loud shouts of approval or praise */
/* EX    They acclaimed him as the best writer of the year. They acclaimed her their leader */

/* NU 2 */
```

```
m_verb(vcomp,account_1,account,account,account,accounts,accounting,
accounted,accounted,accounted,trans,human,
[np(oblig,posprec(1,Wnp),object,_),
np(oblig,posprec(2,Wnp2),object_attribute,_)]).
/* DEF   to consider */
/* EX    He was accounted a wise man. He accounted himself lucky to be alive */

/* NU 3 */
m_verb(vcomp,acknowledge_2,acknowledge,acknowledge,acknowledge,acknowledges,acknowledging,
acknowledged,acknowledged,acknowledged,trans,human,
[np(oblig,posprec(1,Wnp),object,human),
np(oblig,posprec(2,Wnp2),object_attribute,_)]).
/* DEF   to recognize, accept, or admit (as) */
/* EX    He was acknowledged to be the best player. He was acknowledged as their leader. They
acknowledged themselves (to be) defeated */
```

Note
/* The [**hu**] feature on the object is too restrictive: *The problem was acknowledged one of the hardest in the field* */

B. Object complement is an adjective phrase

Example: **consider**: *he considers the teacher very intelligent*

A. FILTERS ON LDOCE

ENTRY **'v' $ pos .and. headclas = "S"**

SEM **headclas # "I"**

GRAMMAR **gramcode="X7 " .or. gramcode="X7e"**

(in the Liège LDOCE data base, **X7e** corresponds to **X (to be) 7** in the printed version of LDOCE)

B. RAW IMPORTED MATERIAL

```
account          accounts accounting accounted accounted v    X7  ....H....Z  1
DEF   to consider
EX    He was accounted a wise man. He accounted himself lucky to be alive
acknowledge         acknowledges acknowledging acknowledged acknowledged v    X7e ....H....H  2
DEF   to recognize, accept, or admit (as)
EX    He was acknowledged to be the best player. He was acknowledged as their leader. They
acknowledged themselves (to be) defeated
```

C. AWK PROGRAM

```
BEGIN {nu=1}
{  semsubj="_"
   if (substr($8,5,1)=="J") semsubj = "thing"
   if (substr($8,5,1)=="H") semsubj = "human"
   if (substr($8,5,1)=="T") semsubj = "abstract"

   semobj = "_"
   if (substr($8,10,1)=="J") semobj = "thing"
   if (substr($8,10,1)=="H") semobj = "human"
   if (substr($8,10,1)=="T") semobj = "abstract"
```

```
  semattr = "_"
  if (substr($8,8,1)=="J") semattr = "thing"
  if (substr($8,8,1)=="H") semattr = "human"
  if (substr($8,8,1)=="T") semattr = "abstract"

  if (($1 ~ /DEF/ || $1 ~ /EX/) && NF>1)
       printf("/* %s */\n",$0)
    else
       { if (NF>1)
       {printf("\n/* NU %d */\n",nu)
       printf("m_verb(vcomp,%s_%s,%s,%s,%s,%s,%s,\n%s,%s,%s,trans,
%s,\n[np(oblig,posprec(1,Wnp),object,%s),\nadjp(oblig,posprec(2,W),object_attribute,%s)]).\n",
       $1,$9,$1,$1,$1,$2,$3,$4,$4,$5,semsubj,semobj,semattr)
       nu++}}
  }
```

D. SAMPLE ENTRIES IMPORTED FROM LDOCE

```
/* NU 1 */
m_verb(vcomp,account_1,account,account,account,accounts,accounting,
accounted,accounted,accounted,trans,human,
[np(oblig,posprec(1,Wnp),object,_),
adjp(oblig,posprec(2,W),object_attribute,_)]).
/* DEF   to consider */
/* EX    He was accounted a wise man. He accounted himself lucky to be alive */

/* NU 2 */
m_verb(vcomp,acknowledge_2,acknowledge,acknowledge,acknowledge,acknowledges,acknowledging,
acknowledged,acknowledged,acknowledged,trans,human,
[np(oblig,posprec(1,Wnp),object,human),
adjp(oblig,posprec(2,W),object_attribute,_)]).
/* DEF   to recognize, accept, or admit (as) */
/* EX    He was acknowledged to be the best player. He was acknowledged as their leader. They
acknowledged themselves (to be) defeated */
```

Ditransitive Verbs

A. FILTERS ON LDOCE

ENTRY **'v' $ pos .and. headclas = "S"**

SEM **headclas # "I"**

GRAMMAR **gramcode="D1 "**

B. RAW IMPORTED MATERIAL

```
accord          accords according accorded accorded v     D1  ..F.H..T.H  2  -I to
DEF   fml to give; allow
EX    He was accorded permission to use the library
afford          affords affording afforded afforded v     D1  ...BZ....T  3
DEF   fml & lit to provide with; supply with; give
EX    The tree afforded us shelter from the rain
allocate        allocates allocating allocated allocated v     D1  ....H..TAU  2
```

```
DEF   to give as a share
EX    We allocated the society some money
allocate            allocates allocating allocated allocated v     D1  ....H...AT  3
DEF   to set apart for somebody or some purpose
EX    That space has already been allocated for building a new hospital
```

C. AWK PROGRAM

Paul Procter's memorandum to the LDOCE editors dated 16/9/1974 specifies that byte 10 of what is for us **defisem** should be used to place semantic restrictions on the **first** object or nominal complement of a ditransitive verb. Byte 8 should be used for the **second** object. However, there is some ambiguity in these specifications, in so far as the indirect object can be **positionally** the first object, but **conceptually** the second. As a matter of fact, the LDOCE lexicographers have been rather inconsistent in their coding of semantic restrictions on **vio** verbs. The **awk** program caters for the most frequent type of coding, i.e. byte 8 for the direct object and byte 10 for the indirect one.

```
BEGIN {nu=1}
{  semsubj="_"
   if (substr($8,5,1)=="J") semsubj = "thing"
   if (substr($8,5,1)=="H") semsubj = "human"
   if (substr($8,5,1)=="T") semsubj = "abstract"

   semiobj = "_"
   if (substr($8,10,1)=="J") semiobj = "thing"
   if (substr($8,10,1)=="H") semiobj = "human"
   if (substr($8,10,1)=="T") semiobj = "abstract"

   semobj = "_"
   if (substr($8,8,1)=="J") semobj = "thing"
   if (substr($8,8,1)=="H") semobj = "human"
   if (substr($8,8,1)=="T") semobj = "abstract"

   if (($1 ~ /DEF/ || $1 ~ /EX/) && NF>1)
        printf("/* %s */\n",$0)
     else
        { if (NF>1)
        {printf("\n/* NU %d */\n",nu)
        printf("m_verb(vio,%s_%s,%s,%s,%s,%s,%s,\n%s,%s,%s,trans,
%s,\n[np(oblig,posprec(2,Wnp1),object,%s),\nio(opt,posprec(1,W2),indirect_object,%s,_)]).\n",
        $1,$9,$1,$1,$1,$2,$3,$4,$4,$5,semsubj,semobj,semiobj)
        nu++}}
   }
```

D. SAMPLE ENTRIES IMPORTED FROM LDOCE

```
/* NU 1 */
m_verb(vio,accord_2,accord,accord,accord,accords,according,
accorded,accorded,accorded,trans,human,
[np(oblig,posprec(2,Wnp1),object,abstract),
io(opt,posprec(1,W2),indirect_object,human,_)]).
/* DEF   fml to give; allow */
/* EX    He was accorded permission to use the library */

/* NU 2 */
m_verb(vio,afford_3,afford,afford,afford,affords,affording,
afforded,afforded,afforded,trans,_,
[np(oblig,posprec(2,Wnp1),object,_),
io(opt,posprec(1,W2),indirect_object,abstract,_)]).
/* DEF   fml & lit to provide with; supply with; give */
/* EX    The tree afforded us shelter from the rain */
```

Note
/* The **T** feature is misplaced: it should be placed on the direct object, not the indirect one */

```
/* NU 3 */
m_verb(vio,allocate_2,allocate,allocate,allocate,allocates,allocating,
allocated,allocated,allocated,trans,human,
[np(oblig,posprec(2,Wnp1),object,abstract),
io(opt,posprec(1,W2),indirect_object,_,_)]).
/* DEF   to give as a share */
/* EX    We allocated the society some money */

/* NU 4 */
m_verb(vio,allocate_3,allocate,allocate,allocate,allocates,allocating,
allocated,allocated,allocated,trans,human,
[np(oblig,posprec(2,Wnp1),object,_),
io(opt,posprec(1,W2),indirect_object,abstract,_)]).
/* DEF   to set apart for somebody or some purpose */
/* EX    That space has already been allocated for building a new hospital */
```

Note
/* Again, **T** is misplaced */

Verbs Taking an Object and a That-Clause

Example: **tell**: *he told the teacher that he had taught linguistics*

A. FILTERS ON LDOCE

ENTRY **'v' $ pos .and. headclas = "S"**

SEM **headclas # "I"**

GRAMMAR **substr(gramcode,1,2) = "D5"**

B. RAW IMPORTED MATERIAL

advise advises advising advised advised v D5 ....H..T.X 1
DEF to tell (somebody) what one thinks should be done; give advice to (somebody)
EX I advise waiting till the proper time. I will do as you advise. I advised her that she should wait. I advised her where to stay. I advise you to leave now

advise advises advising advised advised v D5 ..F.H..T.X 2
DEF fml to give notice to; inform
EX I have advised her that we are coming. Will you advise us (of) when the bags should arrive?

assure assures assuring assured assured v D5a ....H..Z.H 1
DEF to try to cause to believe or trust in something; promise; try to persuade
EX I assure you that this medicine cannot harm you. He assured us of his ability to work
assure assures assuring assured assured v D5a ....H..Z.H 2
DEF to make (oneself) sure or certain

EX   Before going to bed she assured herself that the door was locked

C. AWK PROGRAM

```
BEGIN {nu=1}
{  semsubj="_"
   if (substr($8,5,1)=="J") semsubj = "thing"
   if (substr($8,5,1)=="H") semsubj = "human"
   if (substr($8,5,1)=="T") semsubj = "abstract"

   semiobj = "_"
   if (substr($8,10,1)=="J") semiobj = "thing"
   if (substr($8,10,1)=="H") semiobj = "human"
   if (substr($8,10,1)=="T") semiobj = "abstract"


   if (($1 ~ /DEF/ || $1 ~ /EX/) && NF>1)
          printf("/* %s */\n",$0)
      else
          { if (NF>1)
          {printf("\n/* NU %d */\n",nu)
          printf("m_verb(viothat,%s_%s,%s,%s,%s,%s,%s,\n%s,%s,%s,trans,
%s,\n[io(opt,posprec(1,W2),indirect_object,%s,np),\ns(oblig,posprec(2,W),object]).\n",
          $1,$9,$1,$1,$1,$2,$3,$4,$4,$5,semsubj,semiobj)
          nu++}}
   }
```

D. SAMPLE ENTRIES IMPORTED FROM LDOCE

```
/* NU 1 */
m_verb(viothat,advise_1,advise,advise,advise,advises,advising,
advised,advised,advised,trans,human,
[io(opt,posprec(1,W2),indirect_object,_,np),
s(oblig,posprec(2,W),object]).
/* DEF   to tell (somebody) what one thinks should be done; give advice to (somebody) */
/* EX    I advise waiting till the proper time. I will do as you advise. I advised her that she should wait. I
advised her where to stay. I advise you to leave now */

/* NU 2 */
m_verb(viothat,advise_2,advise,advise,advise,advises,advising,
advised,advised,advised,trans,human,
[io(opt,posprec(1,W2),indirect_object,_,np),
s(oblig,posprec(2,W),object]).
/* DEF   fml to give notice to; inform */
/* EX    I have advised her that we are coming. Will you advise us (of) when the bags should arrive? */

/* NU 3 */
m_verb(viothat,assure_1,assure,assure,assure,assures,assuring,
assured,assured,assured,trans,human,
[io(opt,posprec(1,W2),indirect_object,human,np),
s(oblig,posprec(2,W),object]).
/* DEF   to try to cause to believe or trust in something; promise; try to persuade */
/* EX    I assure you that this medicine cannot harm you. He assured us of his ability to work */

/* NU 4 */
m_verb(viothat,assure_2,assure,assure,assure,assures,assuring,
assured,assured,assured,trans,human,
[io(opt,posprec(1,W2),indirect_object,human,np),
s(oblig,posprec(2,W),object]).
/* DEF   to make (oneself) sure or certain */
/* EX    Before going to bed she assured herself that the door was locked */
```

Note
/* This second reading should be restricted to **assure oneself** */

Verbs Taking an Object and a Prepositional Object

A. FILTERS ON LDOCE

ENTRY **'v' $ pos .and. headclas = "S"**

SEM **headclas # "I"**

GRAMMAR **substr(gramcode,1,2) = "T1" .and. righttypo="I"**

(if **righttypo** = "**I**" (italics) then there is a lexical item in the righthand-side environment and it is a preposition: recall that **particles** are in **bold** and *prepositions* in *italics* in the **rghtctx** field)

B. RAW IMPORTED MATERIAL

```
abandon        abandons abandoning abandoned abandoned v    T1  ....H....H  4  -I to
DEF   to give (oneself) up completely to a feeling, desire, etc.
EX    He abandoned himself to grief. abandoned behaviour
abet           abets abetting abetted abetted v    T1  ....H....X  1  -I in
DEF   to encourage or give help to (a crime or criminal)
EX    He abetted the thief in robbing the bank
absent         absents absenting absented absented v    T1  ..F.H....H  1  -I fro
DEF   to keep (oneself) away
EX    He absented himself from the meeting
absolve        absolves absolving absolved absolved v    T1  ....X....H  2  -I fro
DEF   to free (someone) from fulfilling a promise or a duty, or from having to suffer for wrongdoing
abstract       abstracts abstracting abstracted abstracted v    T1  ....H....C  1  -I fro
DEF   tech to remove by drawing out gently; separate
abstract       abstracts abstracting abstracted abstracted v    T1  ..E.H....C  2  -I fro
DEF   euph to steal
acclaim        acclaims acclaiming acclaimed acclaimed v    T1  ....H..T.H  2  -I as
DEF   to declare to be or publicly recognize as, esp. with loud shouts of approval or praise
EX    They acclaimed him as the best writer of the year. They acclaimed her their leader
```

C. AWK PROGRAM

Here the semantic restrictions on the direct object are in byte 10; byte 8 codes restrictions on the prepositional object.

```
BEGIN {nu=1}
{  semsubj="_"
   if (substr($8,5,1)=="J") semsubj = "thing"
   if (substr($8,5,1)=="H") semsubj = "human"
   if (substr($8,5,1)=="T") semsubj = "abstract"

   semobj = "_"
   if (substr($8,10,1)=="J") semobj = "thing"
   if (substr($8,10,1)=="H") semobj = "human"
   if (substr($8,10,1)=="T") semobj = "abstract"

  # sempobj houses semantic restrictions on the prepositional object
   sempobj = "_"
   if (substr($8,8,1)=="J") sempobj = "thing"
   if (substr($8,8,1)=="H") sempobj = "human"
   if (substr($8,8,1)=="T") sempobj = "abstract"

   # the following if-clauses expand the abbreviations used for prepositions in the right context field
   if ($11 == "fro") $11="from"
   if ($11 == "wit") $11="with"
   if ($11 == "til") $11="till"
   if ($11 == "unt") $11="until"
   if ($11 == "int") $11="into"
   if ($11 == "amo") $11="among"
   if ($11 == "bet") $11="between"
   if ($11 == "ouf") $11="out_of"
   if ($11 == "uon") $11="upon"

   if (($1 ~ /DEF/ || $1 ~ /EX/) && NF>1)
        printf("/* %s */\n",$0)
     else
        { if (NF>1)
        {printf("\n/* NU %d */\n",nu)
        printf("m_verb(vobjfreepp,%s_%s,%s,%s,%s,%s,%s,\n%s,%s,%s,trans,
%s,\n[np(oblig,posprec(1,Wnp),object,%s),\npp(oblig,posprec(1,Wpp),pp_arg,%s,_,%s)]).\n",
        $1,$9,$1,$1,$1,$2,$3,$4,$4,$5,semsubj,semobj,sempobj,$11)
        nu++}}
   }
```

D. SAMPLE ENTRIES IMPORTED FROM LDOCE

```
/* NU 1 */
m_verb(vobjfreepp,abandon_4,abandon,abandon,abandon,abandons,abandoning,
abandoned,abandoned,abandoned,trans,human,
[np(oblig,posprec(1,Wnp),object,human),
pp(oblig,posprec(1,Wpp),pp_arg,_,_,to)]).
/* DEF   to give (oneself) up completely to a feeling, desire, etc. */
/* EX    He abandoned himself to grief. abandoned behaviour */
```

Note
/* The adjective **abandoned** is best dealt with as a separate entry; the entry here should be **abandon oneself to** */

```
/* NU 2 */
m_verb(vobjfreepp,abet_1,abet,abet,abet,abets,abetting,
abetted,abetted,abetted,trans,human,
[np(oblig,posprec(1,Wnp),object,_),
pp(oblig,posprec(1,Wpp),pp_arg,_,_,in)]).
/* DEF   to encourage or give help to (a crime or criminal) */
/* EX    He abetted the thief in robbing the bank */

/* NU 3 */
m_verb(vobjfreepp,absent_1,absent,absent,absent,absents,absenting,
absented,absented,absented,trans,human,
[np(oblig,posprec(1,Wnp),object,human),
pp(oblig,posprec(1,Wpp),pp_arg,_,_,from)]).
/* DEF   to keep (oneself) away */
/* EX    He absented himself from the meeting */
```

Note
/* The entry should read **absent oneself from**; *He absented her from the meeting* is jocular at best*/

```
/* NU 4 */
m_verb(vobjfrepp,absolve_2,absolve,absolve,absolve,absolves,absolving,
absolved,absolved,absolved,trans,_,
[np(oblig,posprec(1,Wnp),object,human),
pp(oblig,posprec(1,Wpp),pp_arg,_,_,from)]).
/* DEF   to free (someone) from fulfilling a promise or a duty, or from having to suffer for wrongdoing */

/* NU 5 */
m_verb(vobjfreepp,abstract_1,abstract,abstract,abstract,abstracts,abstracting,
abstracted,abstracted,abstracted,trans,human,
[np(oblig,posprec(1,Wnp),object,_),
pp(oblig,posprec(1,Wpp),pp_arg,_,_,from)]).
/* DEF   tech to remove by drawing out gently; separate */

/* NU 6 */
m_verb(vobjfreepp,abstract_2,abstract,abstract,abstract,abstracts,abstracting,
abstracted,abstracted,abstracted,trans,human,
[np(oblig,posprec(1,Wnp),object,_),
pp(oblig,posprec(1,Wpp),pp_arg,_,_,from)]).
/* DEF   euph to steal */

/* NU 7 */
m_verb(vobjfreepp,acclaim_2,acclaim,acclaim,acclaim,acclaims,acclaiming,
acclaimed,acclaimed,acclaimed,trans,human,
[np(oblig,posprec(1,Wnp),object,human),
pp(oblig,posprec(1,Wpp),pp_arg,abstract,_,as)]).
/* DEF   to declare to be or publicly recognize as, esp. with loud shouts of approval or praise */
/* EX    They acclaimed him as the best writer of the year. They acclaimed her their leader */
```

Appendix F. Sample Parses

The backbone of the parses is the **predicate-argument-modifier** model. Properties of the clause (voice, aspect, modality, tense) are recorded in the **prop** structure. The arguments are given in **canonical order** according to their deep **gf**. **Athematic** elements do not appear in the parses. The sharing of the variable within the **index** structure is used to indicate coindexing.

1. He was eager to back down.

```
/* control  */
/* mwu */
Parse:
[he,was,eager,to,back,down]

 7                    /* preference index */
  clause
```

```
   pred_arg_mod_structure
   prop(vce: active,asp: none,mod: none,tns: past)
              /* Voice, Aspect, Modality, Tense */
    predicate(be,agr(finite,past,sing,third))
     subject
      nounphrase
      index(_02B0)       /* shared: used to indicate coreference */
      ppro(third,sing,masc)
     subject_attribute
      adjectivephrase
       adjective(eager_1)
        adj_arg
         pred_arg_mod_structure
         prop(vce: active,asp: none,mod: none,tns: present)
          predicate(back_down_1,agr(infinitive))
           subject
            nounphrase
            index(_02B0)       /* shared: used to indicate coreference */
            ppro(third,sing,masc)
```

2. They took the problems he had seen into account.

```
/* mwu */
/* linear precedence */
/* long distance dependencies */
Parse:
[they,took,the,problems,he,had,seen,into,account]

 16
  clause
   pred_arg_mod_structure
   prop(vce: active,asp: none,mod: none,tns: past)
    predicate(take_into_account_1,agr(finite,past,plural_or_second))/* mwu */
     subject
      nounphrase
      index(_0360)
      ppro(third,plural)
     object
      nounphrase
      index(_0544)
       det(the)
       noun(problem_1,agr(plural))
       relative_clause
        clause
         pred_arg_mod_structure
         prop(vce: active,asp: [perfect],mod: none,tns: past)
          predicate(see_1,agr(en_active))
           subject
            nounphrase
            index(_06BC)
            ppro(third,sing,masc)
           object
            gapped_nounphrase
            index(_0544)
```

3. John wants to appear to be loved by Mary.

```
/* control and raising */
 /* passive */
Parse:
[john,wants,to,appear,to,be,loved,by,mary]

 14
  clause
   pred_arg_mod_structure
   prop(vce: active,asp: none,mod: none,tns: present)
    predicate(want_1,agr(finite,present,sing,third))
     subject
      nounphrase
      index(_0334)
       noun(john,agr(sing))
     object          /* the object of WANT is clausal */
      clause
       pred_arg_mod_structure
       prop(vce: active,asp: none,mod: none,tns: present)
        predicate(appear_1,agr(infinitive))
         subject             /* the subject of APPEAR is clausal */
          clause
           pred_arg_mod_structure
           prop(vce: passive,asp: none,mod: none,tns: present)
            predicate(love_1,agr(en_passive))
             subject             /* passive undone */
              nounphrase
              index(_08AC)
               noun(mary,agr(sing))
             object
              nounphrase
              index(_0334)
               noun(john,agr(sing))
```

4. John appears to want to be loved by Mary.

```
/* control and raising */
/* passive */
Parse:
[john,appears,to,want,to,be,loved,by,mary]

 14
  clause
   pred_arg_mod_structure
   prop(vce: active,asp: none,mod: none,tns: present)
    predicate(appear_1,agr(finite,present,sing,third))
     subject
      clause         /* the subject of APPEAR is clausal */
       pred_arg_mod_structure
       prop(vce: active,asp: none,mod: none,tns: present)
        predicate(want_1,agr(infinitive))
         subject
          nounphrase
          index(_0334)
           noun(john,agr(sing))
         object
          clause        /* the object of WANT is clausal */
         pred_arg_mod_structure
         prop(vce: passive,asp: none,mod: none,tns: present)
          predicate(love_1,agr(en_passive))
           subject       /* passive is undone */
            nounphrase
            index(_08AC)
             noun(mary,agr(sing))
           object
            nounphrase
            index(_0334)
             noun(john,agr(sing))
```

5. The genius a book about whom he has read teaches mathematics.

/* long distance dependencies */
Parse:
[the,genius,a,book,about,whom,he,has,read,teaches,mathematics]

```
 19
  clause
   pred_arg_mod_structure
   prop(vce: active,asp: none,mod: none,tns: present)
    predicate(teach_1,agr(finite,present,sing,third))
     subject
      nounphrase
      index(_03C8)                /* coref 1 */
       det(the)
       noun(genius_1,agr(sing))
       relative_clause
        object
         nounphrase
         index(_051C)                /* coref 2 */
          det(a)
          noun(book_1,agr(sing))
           np_arg_of_prep
            prepphrase
            index(_0668)
            prep(about)
             np_arg_of_prep
              nounphrase
              index(_03C8)        /* coref 1 */
               relative(whom)
        clause
         pred_arg_mod_structure
         prop(vce: active,asp: [perfect],mod: none,tns: present)
          predicate(read_1,agr(en_active))
           subject
            nounphrase
            index(_075C)
            ppro(third,sing,masc)
           object
            gapped_nounphrase
            index(_051C)                /* coref 2 */
     object
      nounphrase
      index(_0B68)
```

```
       det(zero)
       noun(mathematics_1,agr(sing_uncountable))
```

6. I am reading in the library a book the students want me to read.

```
/* linear precedence */
/* control */
/* long distance dependencies */
Parse:
[i,am,reading,in,the,library,a,book,the,students,want,me,to,read]

 26
  clause
   pred_arg_mod_structure
   prop(vce: active,asp: [progressive],mod: none,tns: present)
    predicate(read_1,agr(ing))
     subject
      nounphrase
      index(_03E0)
      ppro(first,sing)
     object
      nounphrase
      index(_06E0)
       det(a)
       noun(book_1,agr(sing))
       relative_clause
        clause
         pred_arg_mod_structure
         prop(vce: active,asp: none,mod: none,tns: present)
          predicate(want_1,agr(finite,present,plural_or_second))
           subject
            nounphrase
            index(_0860)
             det(the)
             noun(student_1,agr(plural))
           object
            clause
             pred_arg_mod_structure
             prop(vce: active,asp: none,mod: none,tns: present)
              predicate(read_1,agr(infinitive))
               subject
                nounphrase
                index(_0A98)
                ppro(first,sing)
               object
                gapped_nounphrase
                index(_06E0)
     vp_modifier
      prepphrase
      index(_056C)
      prep(in)
       np_arg_of_prep
        nounphrase
        index(_05C0)
         det(the)
         noun(library_1,agr(sing))
```

7. The teacher has been given a book and the students a library.

```
/* gapping */
/* passive */
Parse:
[the,teacher,has,been,given,a,book,and,the,students,a,library]

 8
  and_sentence
   clause
    pred_arg_mod_structure
    prop(vce: passive,asp: [perfect],mod: none,tns: present)
     predicate(give_1,agr(en_passive))
      object
       nounphrase
       index(_075C)
        det(a)
        noun(book_1,agr(sing))
      indirect_object
       nounphrase
       index(_049C)
        det(the)
        noun(teacher_1,agr(sing))
   clause
    pred_arg_mod_structure
    prop(vce: passive,asp: [perfect],mod: none,tns: present)
       /* gapped v */
     predicate(give_1,_0494)
      object
       nounphrase
       index(_0A7C)
        det(a)
        noun(library_1,agr(sing))
      indirect_object
       nounphrase
       index(_04CC)
        det(the)
        noun(student_1,agr(plural))
```

8. She likes the books that I have written and you have put into the library.

```
/* coordination with across-the-board deletions */
Parse:
[she,likes,the,books,that,i,have,written,and,you,have,put,into,the,library]

 28
  clause
   pred_arg_mod_structure
   prop(vce: active,asp: none,mod: none,tns: present)
    predicate(like_1,agr(finite,present,sing,third))
     subject
      nounphrase
      index(_0434)
      ppro(third,sing,fem)
     object
      nounphrase
      index(_05EC)              /* coref */
       det(the)
       noun(book_1,agr(plural))
       and_relative_clause
        relative_clause
         object
          nounphrase
          index(_05EC)              /* coref */
           relative(that)
         clause
          pred_arg_mod_structure
          prop(vce: active,asp: [perfect],mod: none,tns: present)
           predicate(write_1,agr(en_active))
            subject
             nounphrase
             index(_0844)
             ppro(first,sing)
            object
             gapped_nounphrase
             index(_05EC)        /* coref */
        relative_clause
         clause
          pred_arg_mod_structure
          prop(vce: active,asp: [perfect],mod: none,tns: present)
           predicate(put_1,agr(en_active))
            subject
             nounphrase
             index(_0B2C)
             ppro(second)
            object
             gapped_nounphrase
             index(_05EC)                    /* coref */
            pp_arg
             prepphrase
             index(_0D74)
             prep(into)
              np_arg_of_prep
               nounphrase
               index(_0DC8)
                det(the)
                noun(library_1,agr(sing))
```

Appendix G. Test Suites

For Analysis

Designed for Horatio

They failed.
He was eager to back down.
Do the facts allow the explanation he gave to the students ?

They should back up the teacher.
They should back the good teachers up.
They should back up the teacher they like.
The teacher should have been backed up.
She must allow that John is a good teacher.
She must allow John is a bad teacher.
You must allow for the oversimplifications he has made.
The teacher allows the boys money for books.
He told her that he loved Mary.
She told him what to see.
John has alienated the students from the teacher.
He allowed the students into the library.
The students he had allowed into the library were reading books.
They are teachers.
He is reluctant to go into the library.
The problem is that she knows him.
We have been in the library.
He has become a good teacher.
The books belong in the library.
The girl went to the library.
He brought the books he had liked to the library.
He brought to the library the books he liked.
He considers the claim she has made an oversimplification.
They declared the claim valid.
They will decide where to go.
They did away with the bad teachers.
They want him to kick the bucket.
They should pay attention to the problems he saw.
Great attention was paid to the problems he had seen.
The students had been put at risk.
They took the problems he had seen into account.
They took into account the problems they had seen.
He should take them into account.
The workshop will take place in the library.
They were shooting the breeze.
They allowed her to teach linguistics.
She was allowed to teach linguistics.
They wanted to teach linguistics.
He wanted them to put the workshop off.
John tried to teach linguistics.
* John tried them to teach linguistics.
They persuaded her to teach linguistics.
She was persuaded to teach linguistics.
They expected her to teach linguistics.
She was expected to teach linguistics.
Mary is expected to be elected.
She promised to teach linguistics.
She promised them to teach linguistics.
She seems to have taught linguistics.
It seems that she has taught mathematics.
The book seems to have been read by the students.
The book was expected to have been read.
She is eager to teach.
She is easy to please.
She is an easy woman to please.
The teacher was seen to read a bad book.

The students saw John teach mathematics.
Teachers avoid reading books.
She wants to avoid their reading bad books.
They believed him to have killed a student.
He was believed to have killed a student.
* The book seems to read.
The book seems to be read.
The book seems to have been read by the student.
* The book was seen to read.
The book is believed to have been read.
The man is believed to have read the book.
* The man is believed to have been read.
Mary tends to be annoyed by John.
John tends to annoy Mary.
John tries to annoy Mary.
Mary tries to be annoyed by John.
John wants to appear to be loved by Mary.
John appears to want to be loved by Mary.
When Mary saw John she told him that she wanted him to meet the teacher.
He warned her that she had been seen before she went to the library.
If he saw her he must have seen her before she went into the library.
The teacher who teaches linguistics is good.
The workshop that he wants to put off will fail.
The genius a book about whom he has read teaches mathematics.
She likes the town in which she lives.
She likes the town which she lives in.
She likes the town that she lives in.
She likes the town she lives in.
She likes the town where she lives.
The teacher whose books she likes thinks that she is a good student.
I know the university which she tells him she knows he wants her to go to.
Who knew that John expected her to break down ?
What might the man have been looking at ?
On which table has he put the books ?
Which table has he put the books on ?
Where did he go ?
Have you met Mary ?
Do I know him ?
Are they the teachers who taught you linguistics ?
I knew where he wanted to go.
You must decide which books the students should read.
I told him where to go.
He must have been told where to go.
Might he have been writing a book ?
Does he believe her to have gone in for linguistics ?
Have you read the letter to the teacher about the library ?
The problem with you is that you know me.
Do you back up the decision to give him money ?
They are easy to teach.
John is reluctant to teach linguistics.
John is black.
John has seen a black dog.
He is sure to tell them what to read.
He is sure I will tell them what to read.
Mary is an easy woman to please.
The man reading a book in the library is a teacher.

I want to read a book written by a student.
He went to the library with Mary.
Mary was reading a book about linguistics in the library.
The woman is reading a book in the library.
* The woman is reading in the library a book.
I am reading in the library a book the students want me to read.
She gave books to the students.
* She gave to the students books.
She gave the students good books.
* She gave books the students.
She gave the students the books she wanted them to read.
The teacher took the problems into account.
The teacher took into account the problems.
The teacher took them into account.
* The teacher took into account them.
The teacher took into account the problems the students had seen.
Do you like books about linguistics ?
The man reading a book in the library is a good teacher.
He considers the claim she made an oversimplification.
The students were persuaded to read the books in the library.
He had been looked down on.
Mary has been given a book.
A good book has been given to Mary.
A book has been given to Mary by the student.
The students are expected to read books about linguistics.
The teacher was seen to read a book about women.
Books should be read.
The student was declared a genius.
The problems were paid attention to.
Great attention was paid to the problems the students had seen.
The books they said they liked were put in the library.
He had been told where to meet her.
He was believed to have killed a bad student.
The good books seem to have been read by the students.
The teacher whose books I told her I liked knows the university I have persuaded her to go to.
The students like the books the teacher wants them to read.
What does the teacher think the student is learning ?
Who is the man the woman has been looking for listening to ?
On which table might the man have put the books ?
Which table might the man have put the books on ?
I decided what to tell her I believed her to like.

With coordination:

Mary teaches linguistics and John is learning mathematics.
Mary teaches linguistics and John mathematics.
Mary is and John wants to be in the library.
Mary is in and John wants to be in the library.
Mary went to the library and John to the workshop.
The teacher has been given a book and the students a library.
John and the students want to put off the workshop.
John likes dogs and black cats.
He looked at the teacher and the students.
She made a valid and true claim.
The teacher turned up and broke down.
She declares and considers him a genius.

She told him where to go and what to see.
He can and should see her.
He relied on and liked the students.
He liked and relied on the students.
He backed up and liked the decision to give them money.
He liked and backed up the decision to give them money.
She likes the books that I have written and you have put into the library.
They had been tripping the light fantastic and shooting the breeze.
They tripped the light fantastic and shot the breeze.
They may trip the light fantastic and shoot the breeze.

Based on Flickinger et al. 1987

Abrams works.
Abrams hired Browne.
Abrams showed the office to Browne.
Abrams showed Chiang the office.
Abrams became competent.
Abrams became a manager.
* Abrams became in the office.
* Abrams became working.
Abrams is interviewing an applicant.
Abrams is interviewing.
Abrams is working for Browne.
Abrams is working.
Abrams works for Browne.
* Abrams works of Browne.
Abrams approves of Browne.
* Abrams approves for Browne.
She hired him.
He hired her.
* She hired he.
* He hired she.
* Her hired he.
* Him hired she.
* Her hired him.
* Him hired her.
He showed it to her.
He showed her it.
He showed her an office.
He showed it to Chiang.
* He showed Chiang it.
He interviewed them.
* He interviewed they.
They interviewed him.
* Them interviewed him.
I interviewed Abrams.
Abrams interviewed me.
* Me interviewed Abrams.
* Abrams interviewed I.
We interviewed Abrams.
Abrams interviewed us.
* Us interviewed Abrams.
* Abrams interviewed we.

You interviewed Abrams.
Abrams interviewed you.
She and I interviewed Abrams.
I and she interviewed Abrams.
* Me and her interviewed Abrams.
* Her and me interviewed Abrams.
* Her and I interviewed Abrams.
* She and me interviewed Abrams.
* Me and she interviewed Abrams.
* I and her interviewed Abrams.
Abrams interviewed her and me.
* Abrams interviewed she and I.
Abrams interviewed me and her.
* Abrams interviewed I and she.
* Abrams interviewed her and I.
* Abrams interviewed she and me.
* Abrams interviewed me and she.
* Abrams interviewed I and her.
Whom does she work for ?
Who does she work for ?
For whom does she work ?
Who hired Browne ?
* Whom hired Browne ?
A manager works.
Managers work.
* A manager work.
* Managers works.
I work.
* I works.
You work.
* You works.
He works.
* He work.
We work.
* We works.
They work.
* They works.
They list women who have bookcases.
* They list women which have bookcases.
* They list bookcases who women have.
They list bookcases which women have.
A manager is an employee.
Managers are employees.
Managers are a problem.
Abrams may hire Browne.
Abrams might hire Browne.
Abrams can hire Browne.
Abrams shall hire Browne.
Abrams should hire Browne.
Abrams will hire Browne.
Abrams would hire Browne.
Abrams must hire Browne.
Abrams has hired Browne.
Abrams is hiring Browne.
Abrams could have hired Browne.
Abrams could be hiring Browne.

Abrams could have been hiring Browne.
Browne was interviewed by Abrams.
Browne could be interviewed by Abrams.
Browne has been interviewed by Abrams.
Browne is being interviewed by Abrams.
Browne could have been interviewed by Abrams.
Browne could be being interviewed by Abrams.
Browne has been being interviewed by Abrams.
Browne could have been being interviewed by Abrams.
* Abrams could hired Browne.
* Abrams could hiring Browne.
* Abrams has hire Browne.
* Abrams has hiring Browne.
* Abrams is hire Browne.
* Abrams is hired Browne.
* Abrams has could hire Browne.
* Abrams is coulding hire Browne.
* Abrams is having hired Browne.
* Abrams has could be hiring Browne.
* Abrams could be having hired Browne.
* Abrams could may hire Browne.
* Abrams has had hired Browne.
* Abrams is being hiring Browne.
* Abrams did could hire Browne.
* Browne is could interviewed by Abrams.
* Browne is had interviewed by Abrams.
* Browne did be interviewed by Abrams.
* Browne is did interviewed by Abrams.
* Browne is been interviewed by Abrams.
Abrams knew who to hire.
Abrams knew who Browne hired.
Abrams knew that Browne hired Chiang.
Abrams laid off a programmer.
Abrams laid a programmer off.
Abrams managed to hire Browne.
Abrams promised Browne to hire Chiang.
Abrams promised Browne to be interviewed by Chiang.
Browne was promised by Abrams to hire Chiang.
Browne was promised to hire Chiang.
Abrams urged Browne to hire Chiang.
Browne was urged to hire Chiang.
Abrams urged Browne to be interviewed.
Abrams was urged to be interviewed.
Abrams failed to hire Browne.
Abrams failed to be interviewed by Browne.
Abrams was hired by Browne.
Abrams was hired.
He was hired by her.
* Him was hired by her.
* He was hired by she.
An office was shown to Abrams by Chiang.
An office was shown to Abrams.
An office was shown by Abrams to Browne.
Devito was shown an office by Abrams.
An office was shown Chiang by Abrams.
Abrams was urged to hire Browne by Chiang.

Abrams was urged by Devito to hire Browne.
Abrams was urged to hire Browne.
Abrams was known to be interviewing Browne.
Abrams was known by Chiang to be interviewing Browne.
Abrams was approved of by Chiang.
Abrams was approved of.
Abrams is competent.
Abrams is a manager.
Abrams is in the office.
Abrams hired a woman who was competent.
Abrams hired women who were competent.
* Abrams hired a woman who were competent.
* Abrams hired women who was competent.
Abrams hired women whose manager was competent.
Abrams hired a woman who Browne interviewed.
Abrams hired a woman who Browne approved of.
Abrams hired a woman who Browne knew Chiang interviewed.
Abrams has a bookcase which is heavy.
Abrams has a bookcase that is heavy.
* Abrams has a bookcase what is heavy.
Abrams has an office that Browne showed Chiang.
Abrams has an office which Browne showed Chiang.
Abrams has an office Browne showed Devito.
Abrams has an office that Browne showed to Devito.
Abrams interviewed a woman who Browne showed an office to.
Abrams has an office Devito showed to Chiang.
Abrams hired a woman that was competent.
Abrams hired a woman was competent.
Abrams hired a woman that Browne interviewed.
Abrams hired a woman Browne interviewed.
Abrams hired a woman that Browne approved of.
Abrams hired a woman Devito approved of.
Abrams hired a woman that Browne knew Chiang interviewed.
Abrams hired a woman Browne knew Devito interviewed.
Abrams hired a woman interviewed by Chiang.
Abrams hired a woman working for Chiang.
Abrams hired a woman of whom Chiang approved.
Abrams has a bookcase of which Chiang approved.
Abrams hired a woman the manager of whom Chiang had interviewed.
Abrams hired a woman whose manager Chiang had interviewed.
Abrams interviewed programmers whose manager was Chiang.
* Abrams interviewed programmers whose manager were Chiang.
* Which project is Abrams a programmer and the manager of ?
* Which project does Abrams manage the department and ?
* Which manager did Abrams have an office that Chiang showed to ?
* Which projects does Abrams know who had worked on ?
* Which managers does Abrams know which offices Browne had shown to ?
* Which programmer did Abrams know which office had ?
* Which programmer was that Abrams had interviewed known by Browne ?
Is Abrams a manager ?
Is Abrams competent ?
* Are Abrams competent ?
Does Abrams work for Browne ?
Could Abrams work for Browne ?
Is Abrams working for Browne ?
Has Abrams worked for Browne ?

Could Abrams be working for Browne ?
Could Abrams have worked for Browne ?
Has Abrams been working for Browne ?
Could Abrams have been working for Browne ?
* Has been Abrams working for Browne ?
* Has Abrams is working for Browne ?
* Has Abrams be working for Browne ?
* Have Abrams could be working for Browne ?
Could Abrams have been interviewed by Browne ?
Has Abrams been interviewed by Browne ?
Was Abrams interviewed by Browne ?
Was Abrams interviewed ?
Was Abrams being interviewed by Browne ?
* Did Abrams be interviewed by Browne ?
* Was Abrams been interviewing by Browne ?
Who works for Chiang ?
Who is a manager ?
Who will be a manager ?
Who is managed by Abrams ?
What was shown to Browne ?
Which programmer works for Abrams ?
Which programmer is a manager ?
Which programmer will be a manager ?
Which programmer was hired by Abrams ?
Who does Chiang employ ?
What did Abrams show Browne ?
What did Abrams show to Chiang ?
What was Abrams shown ?
Which programmer did Abrams interview ?
What project does Abrams manage ?
Which office was Abrams shown ?
Where does Abrams work ?
Where was Abrams interviewed ?
Who does Abrams work for ?
Which office does Abrams work in ?
Whom did Abrams show an office to ?
Who is Browne managed by ?
Which department does Abrams know the manager of ?
Whose department does Abrams work in ?
Of whom does Abrams approve ?
In which office does Abrams work ?
To whom did Abrams show an office ?
Who did Abrams show an office ?
Who was shown an office by Abrams ?
By whom is Browne managed ?
Of which department is Browne the manager ?
In whose department does Chiang work ?
Whose manager did Abrams hire ?
* Whose did Abrams hire manager ?
Abrams knows who hired Browne.
Abrams knows who was hired by Browne.
Abrams knows which managers interviewed Browne.
Abrams knows who Browne hired.
Abrams knows who showed Browne an office.
Abrams knows who Browne was hired by.
Abrams knows whom Browne was hired by.

Abrams knows by whom Browne was hired.
Abrams knows which programmers Browne hired.
Abrams knows where Browne works.
The managers of the projects are trustworthy.
The consultants to the managers were hired by Abrams.
The programmer who was hired by Abrams manages the project.
Abrams manages the project that has consultants.
* Abrams manages the project has consultants.
* The programmer was hired by Abrams manages the project.
The programmer whom Browne hired manages the project.
Abrams works on the project that Browne manages.
Abrams works on the project Browne manages.
Abrams works for a competent manager.
Browne hired a competent programmer who Abrams interviewed.
It appears that Chiang interviewed Browne.
Programmers are hard to interview.
* Programmers are hard to interview engineers.
* Programmers is hard to interview.
Chiang was hard to show an office to.
Offices are hard to show to Chiang.
Devito was hard to show an office.
Offices are hard to show Devito.
He worked.
He was working.
He had worked.
He had been working.
He is working.
He has worked.
He has been working.
He will work.
He will be working.
He will have worked.
He will have been working.
Abrams hired a programmer before Devito interviewed an engineer.
Abrams hired a programmer after Devito interviewed an engineer.

With coordination:

Chiang is a manager and Devito is a programmer.
Chiang and Devito work.
* Chiang and Devito works.
Chiang hired Devito and Devito manages Browne.
Chiang hired Devito and manages Browne.
* Chiang hired Devito and manage Browne.
Abrams was interviewed and Browne was hired.
Abrams interviewed programmers and Browne was hired.
Abrams interviews programmers and is managed by Browne.
Abrams was interviewed and was hired.
Abrams was interviewed by Browne and hired by Devito.
Abrams was interviewed and hired by Browne.
Abrams was interviewed and hired.
Chiang works and manages programmers.
* Chiang work and manages programmers.
* Chiang work and manage programmers.
Chiang interviewed programmers and showed Browne an office.
Chiang hired and manages Devito.

* Chiang hired and manage Devito.
Devito works for Devito and with Browne.
Devito works for and with Chiang.
An old and trustworthy employee manages Devito.
* An old and trustworthy employee manage Devito.
Devito is old and trustworthy.
The managers and programmers are employees.
Devito manages a programmer who Abrams interviewed and Browne hired.
Devito manages a programmer Abrams interviewed and Browne hired.
Devito is the manager who interviewed Abrams and hired Browne.
Did Abrams interview a programmer and hire an engineer ?
Who did Abrams interview and Browne hire ?
Who was interviewed by Abrams and hired by Browne ?
* Who does Devito manage and Chiang work for ?
* Who does Devito manage and Chiang works for ?
* Who does Devito manages and Chiang works for ?
* Who does Devito manage Browne and Chiang work for ?
Who was Abrams interviewed by and hired by ?
* Who was Abrams interviewed by Browne and hired by ?
* Who does Devito manage and Chiang work for Browne ?
Devito is the manager with whom and for whom Chiang works.

For Generation

(the following are generatable from the corresponding parses produced by **horatio**)

they failed
he was eager to back down
do the facts allow the explanation he gave the students
they should back up the teacher
they should back up the good teachers
they should back up the teacher they like
the teacher should have been backed up
she must allow john is a good teacher
she must allow john is a bad teacher
you must allow for the oversimplifications he has made
the teacher allows the boys money for books
he told her he loved mary
she told him what to see
john has alienated the students from the teacher
he allowed the students into the library
the students he had allowed into the library were reading books
they are teachers
he is reluctant to go into the library
the problem is she knows him
we have been in the library
he has become a good teacher
the books belong in the library
the girl went to the library
he brought the books he had liked to the library
he brought the books he liked to the library
he considers the claim she has made an oversimplification

they declared the claim valid
they will decide where to go
they did away with the bad teachers
they want him to kick the bucket
they should pay attention to the problems he saw
great attention was paid to the problems he had seen
the students had been put at risk
they took into account the problems he had seen
they took into account the problems they had seen
he should take them into account
the workshop will take place in the library
they were shooting the breeze
they allowed her to teach linguistics
she was allowed to teach linguistics
they wanted to teach linguistics
he wanted them to put off the workshop
john tried to teach linguistics
they persuaded her to teach linguistics
she was persuaded to teach linguistics
they expected her to teach linguistics
she was expected to teach linguistics
mary is expected to be elected
she promised to teach linguistics
she promised them to teach linguistics
she seems to have taught linguistics
she seems to have taught mathematics
the book seems to have been read by the students
the book was expected to have been read
she is eager to teach
she is easy to please
she is an easy woman to please
the teacher was seen to read a bad book
the students saw john teach mathematics
teachers avoid reading books
she wants to avoid them reading bad books
they believed him to have killed a student
he was believed to have killed a student
the book seems to be read
the book seems to have been read by the student
the book is believed to have been read
the man is believed to have read the book
mary tends to be annoyed by john
john tends to annoy mary
john tries to annoy mary
mary tries to be annoyed by john
john wants to appear to be loved by mary
john appears to want to be loved by mary
when mary saw john she told him she wanted him to meet the teacher
if he saw her he must have seen her before she went into the library
the teacher who teaches linguistics is good
the workshop that he wants to put off will fail
the genius a book about whom he has read teaches mathematics
she likes the town in which she lives
she likes the town which she lives in
she likes the town that she lives in
she likes the town she lives in

she likes the town where she lives
the teacher whose books she likes thinks she is a good student
* i know the university which she tells him she knows he wants she go to
who knew john expected her to break down
what might the man have been looking at
on which table has he put the books
which table has he put the books on
where did he go
have you met mary
do i know him
are they the teachers who taught you linguistics
i knew where he wanted to go
you must decide which books the students should read
i told him where to go
he must have been told where to go
might he have been writing a book
does he believe her to have gone in for linguistics
have you read the letter to the teacher about the library
the problem with you is you know me
do you back up the decision to give him money
they are easy to teach
john is reluctant to teach linguistics
john is black
john has seen a black dog
he is sure to tell them what to read
he is sure i will tell them what to read
mary is an easy woman to please
the man reading a book in the library is a teacher
i want to read a book written by a student
he went to the library with mary
mary was reading a book about linguistics in the library
the woman is reading a book in the library
i am reading a book the students want me to read in the library
she gave the students books
she gave the students good books
she gave the books she wanted them to read the students
she gave the students the books she wanted them to read
the teacher took into account the problems
the teacher took them into account
the teacher took into account the problems the students had seen
do you like books about linguistics
the man reading a book in the library is a good teacher
he considers the claim she made an oversimplification
the students were persuaded to read the books in the library
he had been looked down on
mary has been given a book
mary has been given a good book
mary has been given a book by the student
the students are expected to read books about linguistics
the teacher was seen to read a book about women
books should be read
the student was declared a genius
the problems were paid attention to
great attention was paid to the problems the students had seen
the books they said they liked were put in the library
he had been told where to meet her

he was believed to have killed a bad student
the good books seem to have been read by the students
the teacher whose books i told her i liked knows the university i have persuaded her to go to
the students like the books the teacher wants them to read
what does the teacher think the student is learning
who is the man the woman has been looking for listening to
on which table might the man have put the books
which table might the man have put the books on
i decided what to tell her i believed her to like

Appendix H. Prolog Predicates

The predicates covered in this appendix are the **nonpredefined** Prolog predicates used in **horatio** and **horgen** and described in the body of the text. Each predicate is assigned one or several **class**(es), which reflect(s) the area and/or job it is **mainly** used for. The list of classes is the following:

CHECK: used for checking purposes

DATA BASE: defined by a set of facts (no rules)

EXPANSION: expands a lexical macro-clause

GENERATION: used by the generator

LEXICAL: belongs to the vocabulary files

MACRO: macro-clause for the vocabulary

PARSING: used by the parser

UTIL: utility predicate

The **purpose** section describes what the predicate is used for in **horatio** and/or **horgen**. It is not a description of the purpose of the predicate in general terms. The **argument pattern** section is likewise geared towards the application in hand. The **list of arguments** describes any or all of the typical args used in the predicate argument structure. It should be noted that the parsing predicates implement **difference list parsing** and therefore open up with two arguments concerned with the word list: the first is the word list on entry, and the second the word list on exit. These two arguments are called **InputWordList** and **OutputWordList** in argument patterns. The parsing predicates often also include the following arguments:

Gaplist, which houses the list of gaps to be passed up and down in the treatment of long distance dependencies.

Preference, a number reflecting the preference to be assigned to the parse (a tree with a higher preference is to be preferred over one with a lower precedence)

Weight, a number reflecting the structural complexity of a phrase, to account for end placement of weightier elements

accu

Class: UTIL

Purpose: accumulates the preference indexes of the constituents in order to compute the preference index for the whole S

Argument pattern: accu(Total, ListofValues)
List of arguments: Total: sum of the list of values contained in ListofValues
ListofValues: values to be added up

adverb_sentence

Class: PARSING
Purpose: parses an adverbial clause attached to a main declarative S
Argument pattern: adverb_sentence(InputWordList, OutputWordList, Gaps, Preference, Parsetree, Finiteness, Person, Number, Voice)
List of arguments: Gaps is set to the empty list
Finiteness is set to **finite**

adj

Class: LEXICAL / EXPANSION
Purpose: expands the macro-clause `m_adj` for adjectives
Argument pattern: adj(InputWordList, OutputWordList, LexicalTree, SemRestr, Arglist)
List of arguments: SemRestr is the semantic restriction the adjective places on the noun it modifies
Arglist is the adjective's argument list; it may be empty

agree

Class: PARSING / CHECK
Purpose: checks subject-verb agreement
Argument pattern: agree(Personnp, Numbernp, VpAgr)
List of arguments: Personnp: person of subject NP
Numbernp: number of subject NP
VpAgr: one of /**firstsg, thirdsg, other**/

allopt

Class: CHECK
Purpose: checks that all remaining args in the arglist are optional (`allopt` is called after `satisfylist` has applied to the arglist)
Argument pattern: allopt(Arglist)
List of arguments: Arglist is the list of remaining arguments, after `satisfylist` has worked through it

allsubject

Class: CHECK / GENERATION
Purpose: checks that the function passed as argument is a subject
Argument pattern: allsubject(SubjectFunction)
List of arguments: SubjectFunction: subject, subject_inf, subject_pass, ...

append

Class: UTIL
Purpose: standard list appending
Argument pattern: append(L1, L2, L3)

arglist

Class: PARSING
Purpose: modifies, if necessary, the list of arguments to be satisfied. It does so by calling `reog` to deal with arg-changing transformations such as passive or raising
Argument pattern: arglist(InputWordList, OutputWordList, Gaps, Status, Preference, Precedence, RelStat, RelorInt, Voice, Parsetree, Nature, Arglist, PromotedFunction, SubjectFunctor1, SubjectFunctor2, Class)
List of arguments: Status indicates whether an argument or a modifier has been found; it is

set to 1 if the answer is yes
RelStat shows whether a relative or interrogative pronoun or determiner has been found
RelorInt is used to keep relatives and interrogatives apart
Nature shows whether we have an np or a vp arglist
PromotedFunction is the function promoted to subject
SubjectFunctor1 is a subject functor showing the running subject before satisfaction of the `arglist` predicate
SubjectFunctor2: running subject after satisfaction of the `arglist` predicate
Class is the predicate class of the arg-bearing predicate

assoc

Class:UTIL / DATA BASE
Purpose: associates a rank with a grammatical function (used to give args in canonical order in the parse)
Argument pattern: assoc(Gf, Rank)
List of arguments: Gf: a grammatical function (subject, object, ...)
Order: a number reflecting the position of the arg in the canonical order

aux

Class: LEXICAL
Purpose: lexical predicate for auxiliaries
Argument pattern: aux(InputWordList, OutputWordList, LexemeValue, Type, TypeRequired, Number, AgreementValue, Tense)
List of arguments: Type: inflectional type of the aux
TypeRequired: inflectional type of the next aux to the right

before

Class: UTILITY / CHECK
Purpose: checks that an argument tree precedes another in canonical order
Argument pattern: before[F1,R1], [F2,R2]
List of arguments: the arguments are lists whose first member is a grammatical function (object, pp_arg, ...)

c_sentence

Class: PARSING
Purpose: parses main declarative clauses, with or without on either side or on both adverbial subordinate clauses
Argument pattern: c_sentence(InputWordList, OutputWordList, Preference, ParseTree, Finiteness, Person, Number, Voice)
List of arguments: Finiteness is set to **finite**: we are dealing with main declarative clauses

checkaux3

Class: PARSING
Purpose: records relationships between aspect, voice and auxiliary type
Argument pattern: checkaux3(AspectList, InflectionalSpec, Voice)
List of arguments: Aspectlist is a list, possibly empty; members are **progressive** and **perfect**
InflectionalSpec is an inflectional specification on the next verb to the right
Voice is either passive or uninstantiated (interpreted as active voice)

checksem

Class: SEMANTICS

Purpose: checks that two semantic restrictions are compatible
Argument pattern: checksem(Semrestriction1, Semrestriction2)

control

Class: GENERATION
Purpose: takes care of deleting the controller element in the controlled clause; the controller is always the subject in that clause; its index is retained but the body of the np is ghosted
Argument pattern: control(DeeperTree, MoreSurfacyTree)

controller

Class: GENERATION / CHECK / DATA BASE
Purpose: checks that the grammatical function is one that can control the governed subordinate clause, i.e. act as subject
Argument pattern: controller(Function)
List of arguments: Function: subject, object, subject_inf, indirect_object, ...

corenounphrase

Class: PARSING
Purpose: parses typical, 'core' nps
Argument pattern: corenounphrase(InputWordList, OutputWordList, Gaplist, Index, Preference, Weight, Rel, IntRel, Function, ParseTree, Number, Person, Semantics)
List of arguments: see `nounphrase`

cv

Class: LEXICAL / CHECK / GENERATION
Purpose: checks that a verb belongs to the class of control verbs. Such verbs have an arg which plays the role of subject in a nonfinite complement clause (infinitive or ing clause) which is itself an arg of the control verb
Argument pattern: cv(ControlVerb, Required)
List of arguments: Required: one of /**to**, **bare**, **ing** /, indicating the type of nonfinite complement clause

determiner

Class: LEXICAL
Purpose: provides lexical entries for determiners
Argument pattern: determiner(InputWordList, OutputWordList, LexicalParseTree, Number, Semantics, Index)
List of arguments: Semantics is a semantic restriction used for relative and interrogative determiners
Index is an index functor (**rel(Index)**) used for indexing purposes in relative clauses; Index is set to the string **int** for interrogative determiners

drop

Class:UTILITY
Purpose: removes athematic arguments from the list of arguments to appear in the parse tree
Argument pattern: drop([AthematicFunction, RestofTree])
List of arguments: the arg to drop is the list corresponding to the tree for the athematic argument, whose head is the name of the athematic function (surf_object, subject_inf, ...)

expos

Class: LEXICAL / GENERATION
Purpose: checks that a predicate belongs to the class of extraposition verbs (such as seem as used in *It seems that ...*)
Argument pattern: expos(Extraposverb, to)
List of arguments: **to** is the infinitive particle **to**

extrapos

Class: GENERATION

Purpose: extraposes the subject clause to the right and fills the surface subject position with place-filler IT (*It seems that John likes linguistics*)

Argument pattern: extrapos(DeeperTree, MoreSurfacyTree)

first

Class: UTIL

Purpose: generates the first auxiliary in an auxiliary list

Argument pattern: first(List, FirstElement)

first_header

Class: CHECK / GENERATION

Purpose: checks on the environment in generation

Argument pattern: first_header(String)

List of arguments: String: object, cplt_s, ...

flatten

Class: UTIL

Purpose: flattens a list (code is borrowed from Bratko 1990, p.572)

Argument pattern: flatten(List, FlattenedList)

gen

Class: GENERATION

Purpose: central predicate in generation

Argument pattern: gen(ParseTreeList, GeneratedStringList)

List of arguments: ParseTreeList is a list representing a parse tree
GeneratedStringList is a list of generated words

genasp1

Class: GENERATION

Purpose: generates aspectual auxiliaries (`genasp1` takes care of perfect HAVE)

Argument pattern: genasp1(AspectSlotInParseTree, GeneratedAux, RequiredType1, RequiredType2)

List of arguments: RequiredType1 is the inflectional type of the aux
RequiredType2 is the inflectional type of the next aux to the right

genasp2

Class: GENERATION

Purpose: cf. `genasp1`, but deals with the generation of progressive aux BE

Argument pattern: cf. `genasp1`

generate

Class: GENERATION

Purpose: generates strings from parse trees; it is higher than `gen`, because it first calls `prepgen` to undo the results of passive, raising, etc. and then `gen` to perform the actual generation

Argument pattern: generate(ParseTreeList, GeneratedStringList)

List of arguments: cf. `gen`

genlist

Class: GENERATION

Purpose: help predicate for `gen`; it generates from a list by calling `gen` on the head and then itself on the tail

Argument pattern: genlist(ParseTreeList,GeneratedStringList)

List of arguments: cf. `gen`

genyesno

Class: GENERATION
Purpose: generates auxiliaries in yes-no questions; it inserts DO when necessary
Argument pattern: genyesno(ClausePropertyList, Person, Number, GeneratedAuxList)
List of arguments: ClausePropertyList is the prop functor to be found at clause level

getagr

Class: GENERATION
Purpose: fills in the agreement feature in the first auxiliary
Argument pattern: getagr(Auxiliary, Auxagree, Tense, ToOrNotTo)
List of arguments: Tense: **to**, **bare** and **ing** are also possible values, by the side of proper tense values (**present** / **past**)
ToOrNotTo: either a one-element list (**[to]**) or an empty list (**[]**)

inlist

Class: UTIL / CHECK
Purpose: nondeterministic check that an element belongs to a list
Argument pattern: inlist(Element, List)

insert

Class: UTIL
Purpose: inserts an element into a list (used by insertion sort)
Argument pattern: insert(Element, List, NewList)

insort

Class: UTIL
Purpose: sorts the args in canonical order for outputting in the parse, dropping athematic args from the list (see `drop`)
Argument pattern: insort(List, SortedList)

interrogative

Class: LEXICAL
Purpose: provides lexical entries for interrogatives
Argument pattern: interrogative(InputWordList, OutputWordList, Tree, Semrestric, PPsemantics, PpOrNp)
List of arguments: Semrestric: semantic restriction for interrogative nps
PPsemantics: a list of semantic features for interrogatives playing the part of an adverbial (**why**, **when**, **how**, ...)
PpOrNp: indicates whether the interrogative plays the part of an np or an adverbial (**who** vs **when**)

m_adj

Class: LEXICAL / MACRO
Purpose: provides lexical entries (macro-clauses) for adjectives (expanded by `adj`)
Argument pattern: m_adj(Lexeme, Adjective, Semrestric, Arglist)
List of arguments: Lexeme: includes a reading number, e.g. **black_1**
Adjective: the positive degree of the adjective, e.g. **black**
Semrestric: the semantic restriction placed by the adjective on the noun it modifies
Arglist: the adjective's argument list, possibly empty

m_noun

Class: LEXICAL / MACRO

Purpose: provides lexical entries (macro-clauses) for nouns (expanded by `noun`)

Argument pattern: Two patterns, the first one for countable nouns, the second for uncountable ones:

m_noun(Lexeme, Sing, Plural, Semantics, Arglist)
m_noun(Lexeme, Noun, Semantics, Arglist)

List of arguments: Lexeme: includes a reading number, e.g. **patience_1**
Sing, Plural: inflectional variants
Noun: invariant form (for uncountables)
Semantics: noun semantics is a list of semantic features
Arglist: the noun's argument list, possibly empty

m_verb

Class: LEXICAL / MACRO

Purpose: provides lexical entries (macro-clauses) for verbs (expanded by `verb`)

Argument pattern: m_verb(Verbclass, Particle, Lexeme, Infinitive, Firstp, Secondp, Thirdp, Ing, FirstOrThirdPast, SecondOrPluralPast, En, Transitivity, Semrestric, Arglist)

List of arguments: Verbclass: provides a handle for the lexicographer, and is used by the parser and the generator as a check
Particle: the type and form of the particle attached to the verb, if any, e.g. **part0:down**
Lexeme: includes a reading number, as in **look_down_on_1**
Infinitive: bare infinitive form, e.g. **look**
Firstp: first person singular present tense
Secondp: second person present tense, plural present tense (all persons)
Thirdp: third person singular present tense
Ing: ing-form
FirstOrThirdPast: first or third singular past tense
SecondOrPluralPast: second person past tense; plural past tense (all persons)
En: en-form
Transitivity: transitivity feature
Semrestric: semantic restriction on the deep subject
Arglist: the verb's argument list

modifier

Class: PARSING

Purpose: parses modifiers, strings in the input list that do not match lexical args as provided in the predicate's arglist

Argument pattern: modifier(InputWordList, OutputWordList, Type, Gaplist, Preference, Precedence, Rel, Intrel, ParseTree, SubjectFunctor)

List of arguments: Type: to distinguish vp from np modifiers
Rel, Intrel: to keep track of relatives or interrogatives inside the modifier
SubjectFunctor: to provide a subject to be used for vps within the modifier

modppnp

Class: CHECK / DATA BASE

Purpose: checks that a given preposition can head an np modifier

Argument pattern: modppnp(Preposition)

modppvp

Class: CHECK / DATA BASE

Purpose: checks that a given preposition can head a vp modifier

Argument pattern: modppvp(Preposition)

myappend

Class: UTIL
Purpose: does list appending; is able to deal with the value **none** sometimes used in the property list of the clause
Argument pattern: cf. append

nonfinite
Class: CHECK / GENERATION / DATA BASE
Purpose: checks that the inflectional specification of the wordform is non-finite
Argument pattern: nonfinite(InflectionSpec)
List of arguments: InflectionSpec: ing, toinfinitive, en_active, ...

noun
Class: LEXICAL
Purpose: expands macro-clause for nouns, i.e. `m_noun`
Argument pattern: noun(InputWordList, OutputWordList, ParseTree, Number, Semlist, Arglist)
List of arguments: cf. `m_noun`

nounphrase
Class: PARSING
Purpose: parses non-coordinate nps
Argument pattern: nounphrase(InputWordList, OutputWordList, Gaplist, Index, Preference, Weight, Rel, Intrel, Function, ParseTree, Number, Person, Semantics)
List of arguments: Index: used for coindexing purposes
Rel, Intrel: used in the treatment of relative and interrogative nps
Number, Person: percolated from the head noun
Semantics: a list of semantic features percolated from the np head

nsubject
Class: PARSING / CHECK
Purpose: checks that the np which bears the function passed as arg can be the subject of further vps to the right (the **running** subject, used in control relations)
Argument pattern: nsubject(Function)
List of arguments: Function: a grammatical function, such as subject_inf, object, ...

oraise
Class: LEXICAL / GENERATION
Purpose: checks that a verb belongs to the class of object-raising verbs
Argument pattern: oraise(Verb, Requires)
List of arguments: Requires is set to the particle **to**, indicating that **oraise** verbs are followed by a to-infinitive (*I believe him **to** have taught linguistics*)

oraising
Class: GENERATION
Purpose: carries out subject-to-object raising in generation: the subject in the controlled clause is extracted and made the object of the matrix clause (the `control` predicate will take care of deleting the subject of the controlled clause through ghosting)
Argument pattern: oraising(DeeperTree, MoreSurfacyTree)

parse
Class: PARSING
Purpose: main predicate for the parser. `Parse` parses main declarative clauses, yes-no questions and wh-questions
Argument pattern: parse(InputWordList, OutputWordList, ParseTree)

passive

Class: GENERATION
Purpose: restores the surface subject and produces a by-phrase for the deep subject
Argument pattern: passive(DeeperTree, MoreSurfacyTree)

pick

Class: UTIL
Purpose: non-deterministic selection of an element in a list
Argument pattern: pick(List, Element, RemainderOfList)

poids

Class: UTIL
Purpose: computes the rank of a parse subtree according to a given canonical order. It calls on `assoc`
Argument pattern: poids(ParseTree, Rank)
List of arguments: Rank is numeric; the smaller it is, the nearer to the verb the argument whose structure is reflected in the subtree has to be

pp

Class: LEXICAL
Purpose: provides lexical clauses for personal pronouns
Argument pattern: pp(InputWordList, OutputWordList, AgreementFunctor, Person, Number, Gender, Function, SemFeatureList)
List of arguments: AgreementFunctor is **ppro(Person, Number, Gender)**
SemFeatureList is a list of semantic features (the list format is selected to maintain coding conformity with other nps)

precede

Class: PARSING
Purpose: in a list of instantiated args and modifiers, checks that the weight of an element is not greater than the one of the next element to its right
Argument pattern: precede(PosprecLeft, PosprecRight)
List of arguments: PosprecLeft and PosprecRight are **posprec** functors: **posprec(PositionInLexicalArglist, Weight)**, where PositionInLexicalArglist is the standard position of the element as specified in the lexical entry for the arg-bearer, and Weight is to be computed *in situ* (i.e. while parsing the clause) to account for the end-placement of heavier constituents

prep

Class: LEXICAL
Purpose: provides lexical clauses for prepositions
Argument pattern: prep(InputWordList, OutputWordList, Tree, SemList)
List of arguments: Semlist is a list of possible semantic values to be associated with the pp governed by the prep, e.g. **[direction, location, topic]** for ON

prepgen

Class: GENERATION
Purpose: undoes the subject assignments produced by the parser to account for raising, passive, extraposition, ... and calls on `passive`, `control`, `oraising`, `sraising` and `extrapos`
Argument pattern:prepgen(DeeperTree, MoreSurfacyTree)

prepphrase

Class: PARSING
Purpose: parses prepositional phrases
Argument pattern: prepphrase(InputWordList, OutputWordList, Gaplist, Index, NPIndex, Preference, Weight, Preposition, Rel, Intrel, Function, ParseTree, SemPP, SemNP)
List of arguments: Index: index of the whole pp

NPIndex: index of the np inside the pp
Rel, Intrel: keep track of relatives and interrogatives inside the pp
SemPP: inherited from the prep
SemNP: inherited from the np within the pp

priority

Class: PARSING
Purpose: determines person priority in coordinate structures (*you and I* --> *we*, etc.)
Argument pattern: priority(Person1, Person2, WhoWins)
List of arguments: WhoWins is the lower of Person1 and Person2

psubject

Class: PARSING / CHECK
Purpose: checks that the arg bears a function which enables it to be turned into the subject of the corresponding passive
Argument pattern: psubject(Function)
List of arguments: Function: indirect_object, object, etc.

relative

Class: LEXICAL
Purpose: provides lexical clauses for relative pronouns
Argument pattern: relative(InputWordList, OutputWordList, ParseTree, SemNP, SemPP, NPOrPP)
List of arguments: SemNP: a semantic feature or the string **norestriction** (associated with np relatives such as *who*, *which*, *that*)
SemPP: a list of semantic features (associated with adverbial relatives such as *how*, *why*, *where*, *when*)
NPOrPP: marker to indicate whether the relative is np or pp-like

relclause

Class: PARSING
Purpose: parses relative clauses
Argument pattern: relclause(InputWordList, OutputWordList, Preference, ParseTree, GapList, Person, FunctionInRel, NounSem, NumberAntecedent)
List of arguments: FunctionInRel: function of the relative pronoun within the relative clause
NounSem, NumberAntecedent: these two values are inherited from the antecedent

reog

Class: PARSING
Purpose: keeps track of deep subject under passive and other subject-changing transformations
Argument pattern: reog(Voice, Class, SubjectOnEntry, Arglist, SubjectOnExit, NewArgList, PromotedFunction)
List of arguments: Class: predicate class as specified in the lexical entry
SubjectOnEntry and SubjectOnExit are subject functors of the form (n)subject(ParseTree,Semantics)
PromotedFunction points to a function in the active S that can be promoted to subject of the passive

satisfy

Class: PARSING
Purpose: matches an element in the lexical predicate's arglist with a string in the input word list
Argument pattern: satisfy(InputWordList, OutputWordList, GapList, Preference, Precedence, Rel, IntRel, ParseTree, Arg, SubjectOnEntry, SubjectOnExit)

List of arguments: Rel, IntRel are used to keep track of relatives and interrogatives
Arg: the lexical argument to be matched
SubjectOnEntry: subject functor for the running subject on entering `satisfy`
SubjectOnExit: subject functor for the running subject on exiting `satisfy`

satisfylist

Class: PARSING
Purpose: matches the arguments in the predicate's arglist against the input word list; calls `satisfy`
Argument pattern: satisfylist(InputWordList, OutputWordList, ArgOrModFound, Preference, Weight, Rel, IntRel, Voice, ParseTreeList, NporVp, Arglist, Func, SubjectFunctor)
List of arguments: cf `satisfy` +
ArgOrModFound: flag to indicate whether a match has been found (either an argument or a modifier)
NpOrVp: flag to indicate whether the arglist belongs to an np or a vp
Func: grammatical function in the active that can be promoted to subject of passive
SubjectFunctor: subject functor for the **running** subject

second_header

Class: GENERATION / CHECK / DATA BASE
Purpose: checks the environment in generation
Argument pattern: second-header(Header)
List of arguments: Header: clause, np_modifier, adj_arg, ...

sentence

Class: PARSING
Purpose: parses declarative clauses
Argument pattern: sentence(InputWordList, OutputWordList, GapList, Preference, ParseTree, Type, PersonVp, Number, Voice)
List of arguments: Type: finite / nonfinite
PersonVp: the person is inherited from the vp
Voice: uninstantiated if the voice is active

sfok

Class: SEMANTICS
Purpose: checks that **nps** and **pps** satisfy the semantic requirements of verbs - the feature required by the verb must belong to the list of semantic features associated with the noun, or must be reachable from one of them through feature implications (`up`) materializing a feature hierarchy
Argument pattern: sfok(SemRestric, ListOfFeatures)
List of arguments: SemRestric is a semantic restriction
ListOfFeatures is a list of semantic features

sraise

Class: LEXICAL / GENERATION
Purpose: checks that a verb is a subject-to-subject raising verb
Argument pattern: sraise(SraisingVerb, to)
List of arguments: **to** is the infinitive particle **to** used in generating the nonfinite complement clause (*John seems **to** teach linguistics*)

sraising

Class: GENERATION
Purpose: takes care of subject-to-subject raising
Argument pattern: sraising(DeeperTree, MoreSurfacyTree)

subject_active

Class: GENERATION / CHECK / DATA BASE

Purpose: checks that the arg which bears the function passed as arg to `subject_active` can be the subject of a clause in the active voice

Argument pattern: subject_active(Function)

List of arguments: Function must be either **subject** or **subject_inf** if the predicate is to succeed

up

Class: SEMANTICS

Purpose: explores a semantic hierarchy to exploit semantic inheritance relations (just like the `ancestor` predicate)

Argument pattern: up(SubClass, BiggerClass)

up1

Class: SEMANTICS / DATA BASE

Purpose: builds up an **ako** relation in a semantic hierarchy (just like the `parent` predicate)

Argument pattern: up1(SubClass, BiggerClass)

verb

Class: LEXICAL / EXPANSION

Purpose: expands `m_verb` clauses (macro-clauses for verbs)

Argument pattern: verb(InputWordList, OutputWordList, Class, ParseTree, Finiteness, Tense, Number, Agreement, SemSubj, Args)

List of arguments: Class: provides a handle for the lexicographer and is used for various checks in the parser and generator

Agreement: the agreement value of the verb

SemSubj: the semantic restriction placed by the verb on the (deep) subject

Args: the verb's argument list

verbphrase

Class: PARSING

Purpose: parses verb phrases

Argument pattern: verbphrase(InputWordList, OutputWordList, SubjectOnEntry, Gap, Preference, ParseTree, Rel, Intrel, Type, Tense, Aspect, Modality, Number, Person, Voice, SubjectOnExit)

List of arguments: SubjectOnEntry is a subject functor (**subject(Tree, Sem)**) giving the running subject on entering the procedure

Rel, Intrel: used in the treatment of relatives and interrogatives inside the vp

Type: finite / nonfinite

SubjectOnExit is a **nsubject** functor (**nsubject(Tree,Sem)**) giving the **running** subject on exiting the procedure

whquestion

Class: PARSING

Purpose: parses wh-questions as main clauses

Argument pattern: whquestion(InputWordList, OutputWordList, Preference, ParseTree)

The predicates whose names begin with an x deal with coordinate structures; they have the same argument pattern and argument list as their simpler equivalents. They are:

xadjphrase
xaux
xnoun
xnounphrase

xprepphrase
xrelclause
xsentence
xverbphrase

yesnoquestion

Class: PARSING
Purpose: parses yes-no questions
Argument pattern: yesnoquestion(InputWordList, OutputWordList, Preference, ParseTree)

Index to Prolog Predicates

PA points to a predefined predicate specific to Arity Prolog, or at least not standardly available in the family of Edinburgh Prologs. For the non-PA predicates the last reference generally points to the appendix on Prolog predicates.

References

1. Prolog

a) Bratko, I., *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 1990, second edition

b) Clocksin, W. F. and Mellish, C. S., *Programming in Prolog*, Third Edition, Springer-Verlag, 1987

c) Marcus, C., *Prolog Programming*, Addison-Wesley, 1986

d) Ross, P., *Advanced Prolog*, Addison-Wesley, 1989

e) Sterling, L. and Shapiro, E., *The Art of Prolog*, Advanced Programming Techniques, The MIT Press, 1986

2. Prolog and NLP

a) Covington, M. A., *Natural Language Processing for Prolog Programmers*, Englewood Cliffs, Prentice Hall, 1993

b) Gal, A, et al., *Prolog for Natural Language Processing*, Wiley, Chichester, 1991

c) Gazdar, G. and Mellish, Ch., *Natural Language Processing in PROLOG*, Addison-Wesley, Reading, Mass., 1989.

d) Pereira, F.C.N. and Shieber, S.M., *Prolog and Natural-Language Analysis*, CSLI Lecture Notes, Nr 10, 1987

e) Walker, A. et al., *Knowledge Systems and Prolog*, Addison-Wesley, Reading, Mass, 1987.

3. Linguistics

Shieber's book is a succinct, but very informative introduction to the concept of unification in grammar, which plays a crucial role in contemporary linguistic theories.

Shieber, S., *An Introduction to Unification-Based Approaches to Grammar*, Chicago University Press, Chicago, 1986

On linguistic theories belonging to the generative framework, the following two introductions stand out:

Sells, P, *Lectures on Contemporary Syntactic Theories*, CSLI Lecture Notes Nr 3, Chicago University Press, Chicago, 1985

Horrocks, G., *Generative Grammar*, Longman, London, 1987

The four main models are described in the following books, which do not aim at pedagogical presentation:

1) Lexical Functional Grammar (LFG)

Bresnan, J. W., Kaplan, R. M., *Lexical Functional Grammar*, in Bresnan, J.W. (ed.), *The Mental Representation of Grammatical Relations*, Cambridge, Mass., MIT Press, 1982

2) Generalized Phrase Structure Grammar (GPSG)

Gazdar, G., Klein, E., Pullum, G., Sag, I., *Generalized Phrase Structure Grammar*, Basil Blackwell, Oxford, 1985

3) Head-Driven Phrase Structure Grammar (HPSG)

Pollard, C. J, Sag, I. A., *Information-based Syntax and Semantics*, Vol. 1: Fundamentals, CSLI Lecture Notes nr 13, Chicago University Press, Chicago, 1987; Vol.2: Agreement, Binding, and Control, 1991.

4) Government and Binding (GB)

Chomsky, N., *Lectures on Government and Binding*, Foris, Dordrecht, 1982

Eclectic and fairly comprehensive:

Quirk, R, Geenbaum, S, Leech, G., Svartvik, J., *A Comprehensive Grammar of the English Language*, Longman, 1985

Dictionary:

LDOCE = P.Procter (ed.), *Longman Dictionary of Contemporary English*, Longman, London, 1978

Other publications referred to:

Aho et al. 1988 = Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, Mass., 1988

Alshawi et al. 1992 = Alshawi, H. et al., *The Core Language Engine*, The MIT Press, Cambridge, Mass. and London, 1992

Boguraev and Briscoe 1989 = Boguraev, B. and Briscoe, T. (eds) ,*Computational Lexico-graphy for Natural Language Processing*, Longman, London and New York, 1989

Bresnan 1981 = Bresnan, J., **A Realistic Transformational Grammar**, in Halle, M., Bresnan, J. and Miller, G.A. (eds), *Linguistic Theory and Psychological Reality*, The MIT Press, Cambridge, Mass., 1981

Dahl and Abramson 1984 = Dahl, V. and Abramson, H., **On gapping grammars**, *Proceedings Second Logic Programming Conference*, Uppsala, 1984

Dymetman and Isabelle 1990 = Dymetman, M. and Isabelle, P., **Grammar bidirectionality through**

controlled backward deduction, in Saint-Dizier and Szpakowicz 1990, pp. 275-293

Flickinger et al. 1987 = Flickinger, D., Nerbonne, J., Sag, I., and Wasow, T., **Toward evaluation of NLP systems**, *Workshop paper at the 25th Int. Mtg. of the Assoc. for Computational Linguistics*, Stanford University, Cal., USA

Hirschman and Dowding 1990 = Hirschman, L. and Dowding, J., **Restriction Grammar: a logic grammar**, in Saint-Dizier and Szpakowicz 1990, pp. 141-167

Isabelle et al. 1988 = Isabelle, P., Dymetman, M. and Mackiovitch, E., **CRITTER: a translation system for agricultural market reports,** *Proceedings of the 12th International Conference on Computational Linguistics*, Budapest, 1988

Matsumoto 1991 = Matsumoto, Y., **Handling Coordination in a Logic-based Concurrent Parser**, in Brown, C. and Koch, G. (eds), *Natural Language Understanding and Logic Programming III*, Elsevier Science Publishers, 1991, pp. 1-12

Matsumoto et al. 1983 = Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H. and Yasukawa, H., **BUP: a bottom-up parser embedded in Prolog**, *New Generation Computing*, 1(2), 1983

Michiels 1982 = Michiels, A., **Exploiting a Large Dictionary Data Base**, Unpublished PhD Thesis, University of Liège, Liège, 1982

McCord 1982 = McCord, M. C., **Using slots and modifiers in logic grammars for natural language**, *Artificial Intelligence*, Vol. 18, pp. 327-367

McCord 1987 = McCord, M. C., Chapter 5 of **Walker et al. 1987**

McCord 1989a = McCord, M. C., **A New Version of the Machine Translation System LMT**, *Journal of Literary and Linguistic Computing*, 4, pp. 218-229

McCord 1989b = McCord, M. C., **LMT and Slot Grammar**, paper read at the IBM Europe Institute, August 1989, Garmisch-Partenkirchen

McCord 1990 = McCord, M. C., **SLOT GRAMMAR: A System for Simpler Construction of Practical Natural Language Grammars**, in Studer, R. (ed), *International Symposium on Natural Language and Logic*, Lecture Notes in Computer Science, Springer-Verlag

Pereira 1981 = Pereira, F.C.N., **Extraposition Grammars**, *American Journal of Computational Linguistics*, Vol.9, Nr 4, 1981

Pereira and Warren 1980 = Pereira, F.C.N, and Warren, D.H.D., **Definite clause grammars for language analysis: a survey of the formalism and comparison with augmented transition networks**, *Artificial Intelligence*, 13 (3), 1980

Saint-Dizier and Szpakowicz 1990 = Saint-Dizier, P. and Szpakowicz, S., (eds), *Logic and Logic Grammars for Language Processing*, Ellis Horwood, Chichester, 1990

Saint-Dizier et al. 1990 = Saint-Dizier, P., Toussaint, Y., Delaunay, C., Sebillot, P., **A natural language processing system based on the government and binding theory**, in Saint-Dizier and Szpakowicz 1990, pp. 108-140

Tomita 1991 = Tomita, M., **Why Parsing Technologies**, in Tomita, M. (ed), *Current Issues in Parsing Technology*, Kluwer Academic Publishers, Boston, 1991

[1] We follow Alshawi et al. 1992 (p.1) in distinguishing between **analysis** (using only linguistic knowledge) and **interpretation** (applying contextual knowledge)

[2] The companion disk includes the following :

- the grammar files for the analysis and generation components of **horatio**

- the vocabulary files

- the CMD(OS/2) and BAT(DOS) files for compiling and linking the program

- the AWK program used for the selective downloading of the lexicon

- the SED program used for manipulating the Prolog terms to make them suitable for the generator

- the test suites used to demonstrate the capabilities of the system for parsing and generation

- some sample parses highlighting some of the capabilities of the system

All files are documented and ready to be compiled or run, on both DOS and OS/2.

[3] Arity Prolog is produced by Arity Corporation, Damonmill Square, Concord, Massachusetts, USA.

[4] McCord 1987 = Chapter 5 (*Natural Language Processing in Prolog*) of Adrian Walker (ed), **Knowledge Systems and Prolog**, Addison-Wesley, Reading, Mass., 1987

[5] It should be emphasized here that **horatio** has never been part of 'standard' Eurotra, but has always remained a sideline. More information on Eurotra can be found in the various volumes in the *Studies in Machine Translation and Natural Language Processing* series published by the Commission of the European Communities in Luxemburg, especially in Volumes I and II, 1991 (I : *The Eurotra Linguistic Specifications* ; II : *The Eurotra Formal Specifications*). Both volumes are edited by C. Copeland, J. Durand, S. Krauwer and B. Maegaard.

[6] In agreement with McCord (see **McCord 1987**, p. 338), I do not include the subject in the predicate's argument list. As McCord writes, "Since every verb has a subject (in a finite clause), we will not actually list the subject slot by name, but will just put the subject marker in a separate argument of the lexical entry" (in McCord's system a **marker** is a semantic restriction).

[7] `satisfy` is similar to `fill` in McCord's **Modular Logic Grammar** (see **McCord 1987**, p. 344 and foll.)

[8] `Satisfylist` is similar to `postmods` in McCord's **MLG** (see **McCord 1987**, p. 344 and foll.)

[9] `Pick` is defined by the following code :

```
 pick([Head|Tail],Head,Tail).
  pick([Head|Tail],Elem,[Head|Ntail]) :-
                pick(Tail,Elem,Ntail).
```

[10] Cf the **theme** in McCord's **MLG** (see **McCord 1987**, p. 346)

[11] Note that this is the same **SEEM** as the **sraising** one.

[12] *An easy question for you to answer* is dealt with below.

[13] Covington 1992 (p. 300) defines a predicate **once** in standard Prolog whose semantics is the same as that of the snips.

[14] McCord (see **McCord 1987**, p. 348 and following) has designed a similar system for precedence relations. His predicate `precede` inspired the treatment in **horatio**.

[15] Matsumoto 1991 has a similar check: "The rules for coordination handling are activated only when a coordinate conjunction appears in the input" (p. 10).

[16] Cf. the section on **left extraposition** in **McCord 1987** (p. 351 and foll.).

[17] At least in the subset of English that **horatio** attempts to cover, which does not include *Our Father, which art in Heaven* ...

[18] FAR code can be housed in other memory segments than the 64K segment automatically allocated to Arity Prolog code. In the segmented memory model used in this version of Arity Prolog, such declarations are necessary in middle-sized and large grammars.

[19] **MKS** (Mortice Kern Systems, Ontario, Canada) produce implementations of the Unix utilities for the DOS and OS/2 operating systems, including **awk** (for which the OS/2 implementation provides a large version - **awkl** - which allows longer records to be treated).

[20] The reader is assumed to know this UNIX utility; see **Aho et al. 1988**.

[21] RLRN is the conjunction of two field (i.e. subject matter) codes:

RL : Religion (not Bible) (Christian and/or Jewish)

RN : Religion (other than Christian and Jewish)

[22] H in byte 5 : deep subject has feature [+ HUMAN]

[23] X in byte 10 : deep object has feature [+ ABSTRACT] or [+ HUMAN]

[24] T in byte 10 : deep object has feature [+ ABSTRACT]