HPSG Grammars in ALE

Colin Matheson

HPSG Grammars in ALE

Colin Matheson

Foreword

This is a course in the development of Head-driven Phrase Structure Grammars in ALE, the Attribute-Logic Engine. The work was largely funded under the Joint Information Systems Committee's New Technology Initiative (NTI/39).

The HTML files were produced using latex2html, so a number of compromises were necessary. Notably, the graphics are a bit messy, and the `examples' style is clunky too. In particular, the paper discussing ALE in Emacs hasn't survived the translation too well -- many readers will be able to skip this, although some of the introductory comments may be useful.

Please note that there are a number of (fairly small) graphics -- some screen dumps, HPSG-style Attribute-Value Matrices, and so on -- which are necessary. So, if you normally switch off graphics downloading, don't in this case.

The course was designed for a 10-week term of postgraduate teaching on the Linguistics MSc at the University of Edinburgh. Each section, apart from the first, represents a lecture, and the accompanying exercises were partly supervised in practical sessions.

The material covered includes all the HPSG schemas and most of the central types and features. There's a section on semantics, and some reasonably advanced Linguistic topics, such as the analysis of raising and control structures and binding theory, are discussed. All of the analyses are provided with ALE implementations apart from the last part which looks at *tough* constructions and binding. Apart from the future inclusion of a short section on prepositional phrases, and some exercises for parts 8 and 9, no changes are envisaged. The latter may be added at the beginning of 1997.

The course assumes a knowledge of basic syntactic issues -- in Edinburgh it follows a 10-week introductory course which is more or less theoretically neutral. An introductory knowledge of Prolog is probably also an advantage, but I don't think it's crucial.

Source files can be found in the /sources directory at this URL. All the grammars are in /grammars, and the LaTeX source files are in /latex, which includes an archived version. There are compressed and uncompressed PostScript versions in /postscript.

I am greatly indebted to Claire Grover and Suresh Manandhar, whose advanced course in computational grammars provides much of the background material and most of the actual analyses included here. The typos and brainos are all mine, though. I must

also thank the students who completed the course on whom a number of options were tested. Comments on any of the material, or the Web presentation, are very welcome.

HPSG Grammars in ALE

1 Interfaces for ALE

Introduction

This is the introductory part of a course in developing HPSG grammars in ALE, the Attribute-Logic Engine. Background reading to the course can be found in the recent HPSG book (see the HPSG home pages at `http://ling.ohio-state.edu/HPSG/Hpsg.html` for more information) and in Carpenter & Penn (1994) (further details on the ALE home pages at `http://macduff.andrew.cmu.edu/ale`).

It is assumed here that all the background issues have been been taken care of -- the Prolog and Emacs set-ups, and so on. All the necessary source files and information can be found at the above URLs. This paper describes two example sessions, one with the system being run in an Emacs buffer, and another using Olivier Laurens' ALE-Emacs interface, which is a much better option if it is available. Just about everything which is true about running the system in a buffer will be true of running it outside Emacs altogether -- life's a lot simpler, though, when the Emacs facilities (especially with Laurens' extensions) are available.

We now cover running ALE, loading a simple phrase structure grammar, inspecting the contents of the grammar, parsing input, and exiting. This is followed by a look at the example grammar used in the session.

Starting ALE

Assuming everything is set up properly, running the system is just a matter of typing `ale` at the Unix prompt:

```
> ale

ALE Version 2.0.1; 9 Jan 1995
Copyright (C) 1995, Bob Carpenter and Gerald Penn
All rights reserved.

interpreter is inactive

yes
| ?-
```

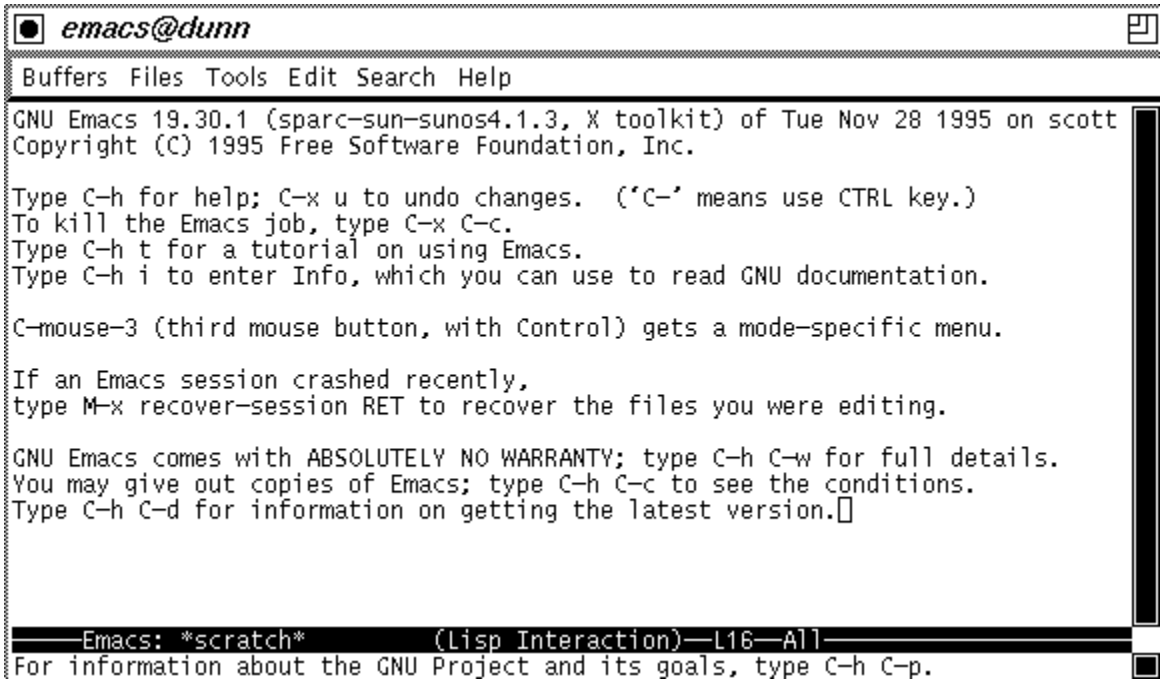
However, one way to make life easier is to run everything from inside Emacs, as the following section shows.

ALE in Emacs

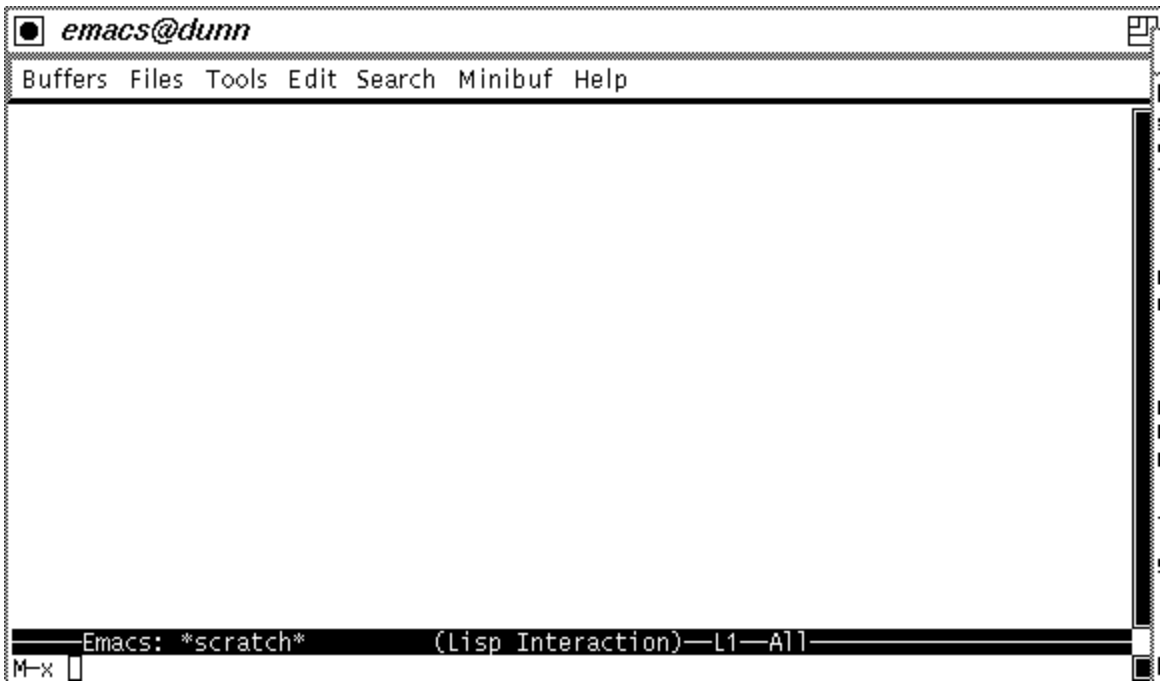
All you have to do is start Emacs:

```
> Emacs
```

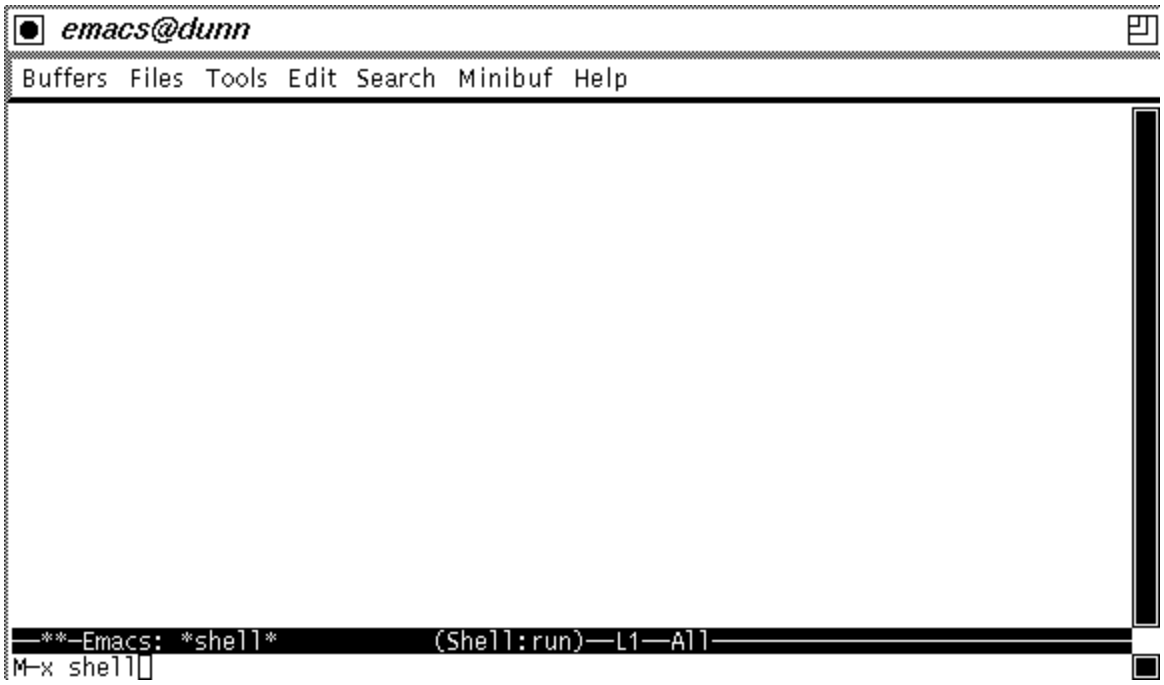
This will put you in the normal Emacs window:



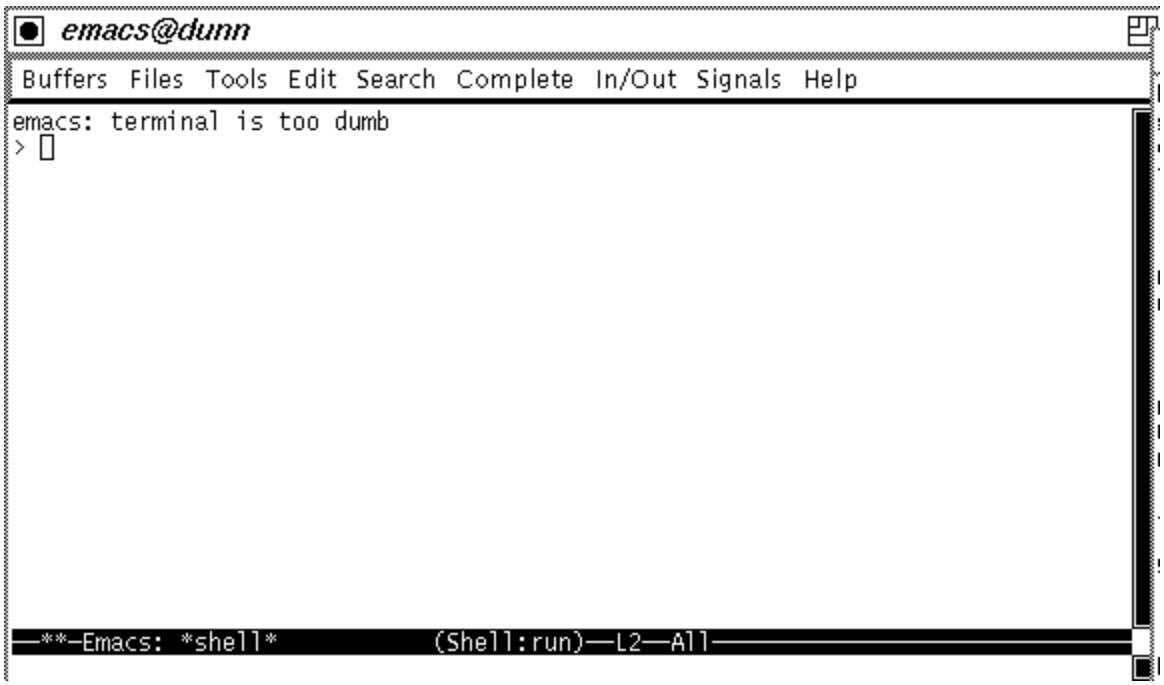
If you type ESCAPE-X (The escape key followed by X), you will get the command line:



Start a shell here by typing `shell` followed by `RETURN`:



You'll now be back in the Emacs window, but with a shell prompt:



You can now start ALE as before, but you have all the facilities of Emacs available. Try typing `ls` to confirm the contents of your directory, and then recall the previous command using `ESCAPE-P` -- this is a very useful option when working in ALE. You can move through previous commands using `ESCAPE-P` and `ESCAPE-N`, which makes life a lot simpler. Similarly, the cut-and-paste commands can be very helpful -- the history of your interaction is retained in the Emacs buffer.

The following sections look at some of the commands for loading and inspecting grammars. The examples given work perfectly well outside Emacs, of course.

Loading Grammars

Having started ALE, we can load a grammar using the `compile_gram` command -- here we are using the grammar in `grammar0.pl` to illustrate the basic facilities:

```
| ?- compile_gram(grammar0).
{consulting /import/usersA/colin/Teach/ALE/grammar0.pl...}
{/import/usersA/colin/Teach/ALE/grammar0.pl  consulted,  40  msec  1248
bytes}
 .
 .
 .
{compiling /import/usersA/colin/Teach/ALE/.rule.pl...}
{/import/usersA/colin/Teach/ALE/.rule.pl compiled, 100 msec 448 bytes}

yes
| ?-
```

The `.pl` filename extension can be omitted, as shown. Various warnings may appear, indicating for example that the grammar contains no lexical rules. The system will normally work perfectly well -- the messages are simply for information. However, more serious problems often occur which are flagged as errors and which usually mean that compilation has not been completed successfully. In the latter case the grammar will need to be edited and recompiled.

Inspecting Grammars

Having compiled a grammar, various aspects of the loaded representations can be checked. For example, we can look at the lexical entries using the `lex` command. I know there's a word *sandy* in the lexicon, so its entry can be inspected as follows:

```
| ?- lex(sandy).

WORD: sandy
ENTRY:
np

ANOTHER?
```

Typing `y.` (the fullstop is necessary with all input) will cause the program to look for another entry for *sandy*. If a Prolog variable had been used as an argument in place of the constant *sandy*, all the lexical entries would have been listed. The information following `ENTRY:` represents the feature structure associated with the lexical item -- in this case *sandy* is just entered as an NP.

The `rule` command can be used in a similar way to inspect the grammar rules. As with `lex`, a Prolog variable (a symbol starting with an upper-case letter) can be used in the argument position to inspect all the entries, so `rule(X)` produces the following:

```
RULE: np_vp

MOTHER:

  s

DAUGHTERS/GOALS:

CATEGORY  np

CATEGORY  vp

ANOTHER?
```

The grammar only contains one rule, as asking for more would show. We shall see how to enter and edit the rules and lexical entries soon.

Parsing

Input to the parser must be a Prolog list. So `rec[sandy,snored]` results in:

```
STRING:
0 sandy 1 snored 2

CATEGORY:

s

ANOTHER?
```

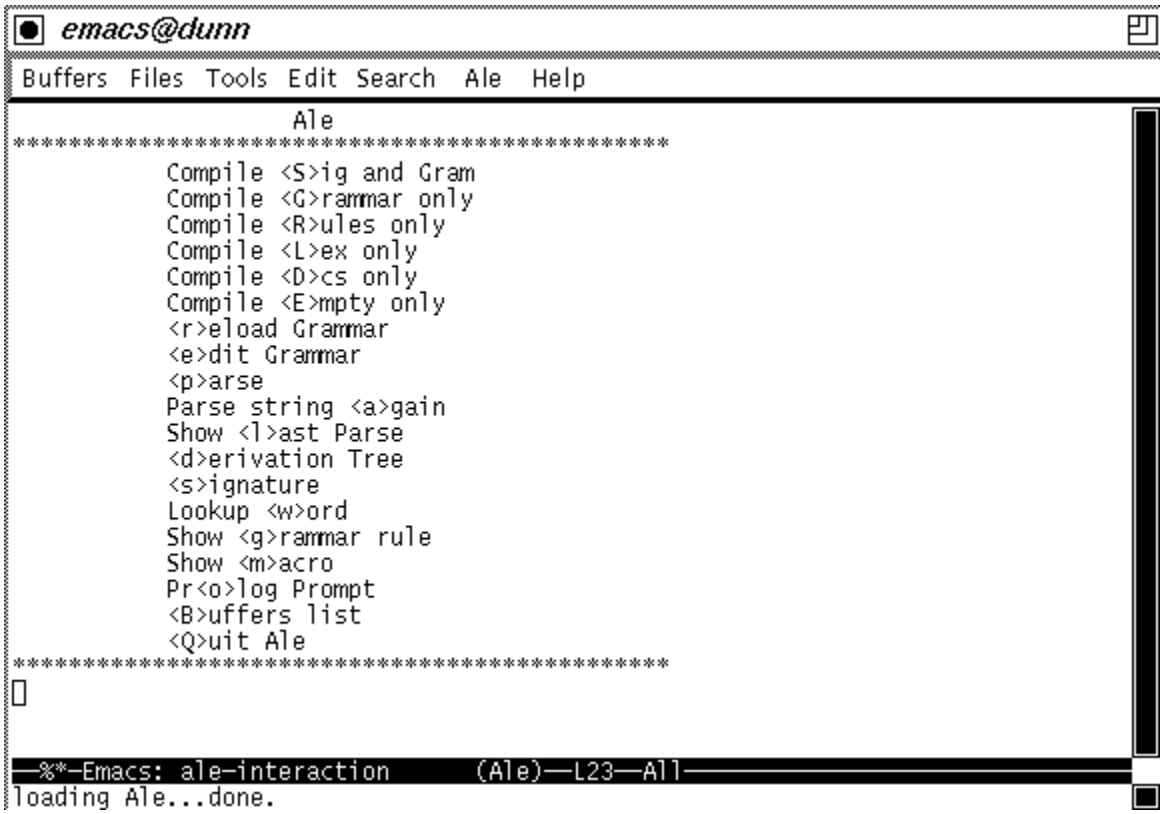
The system can obviously recognise the string *sandy snored* as a sentence. There may be other parses, of course, hence the `ANOTHER` prompt. Typing `y.` will result in a negative answer and the return of the system prompt.

Exiting

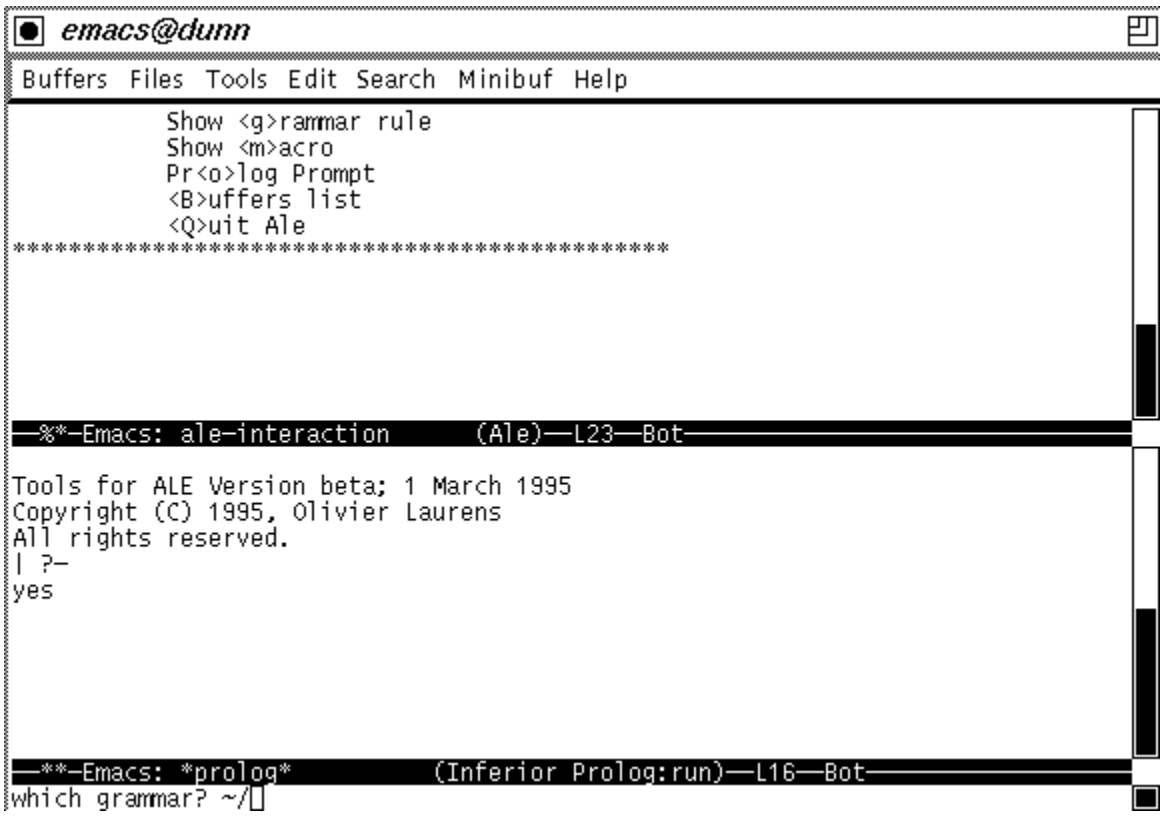
To exit, just type `CTRL-X-CTRL-C`, which will take you to the command line again where you will be asked if you want to kill the process and exit. Just type `yes` and that's it.

2 Laurens' ALE Interface

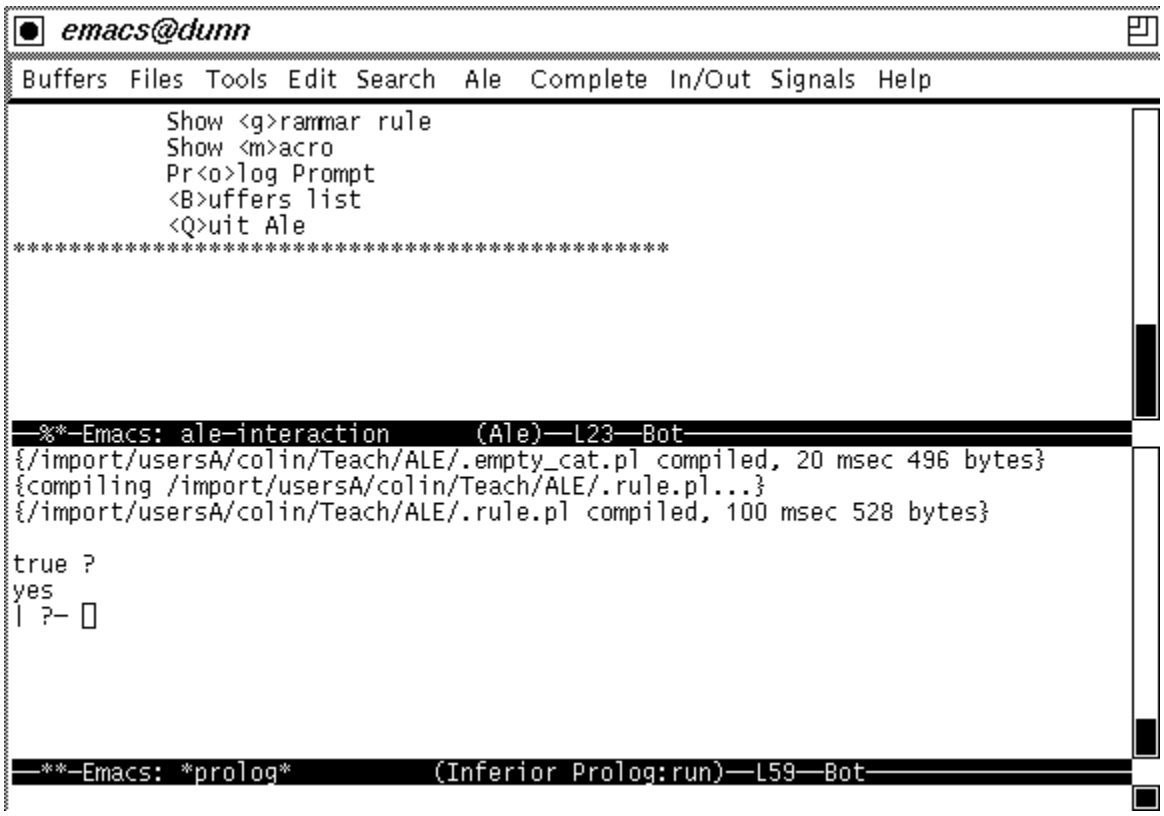
Full information on this interface can be found on the ALE home pages at CMU, as mentioned in the introduction. Having completed the necessary set-up and started the interface, the user is presented with the following screen (inside Emacs):



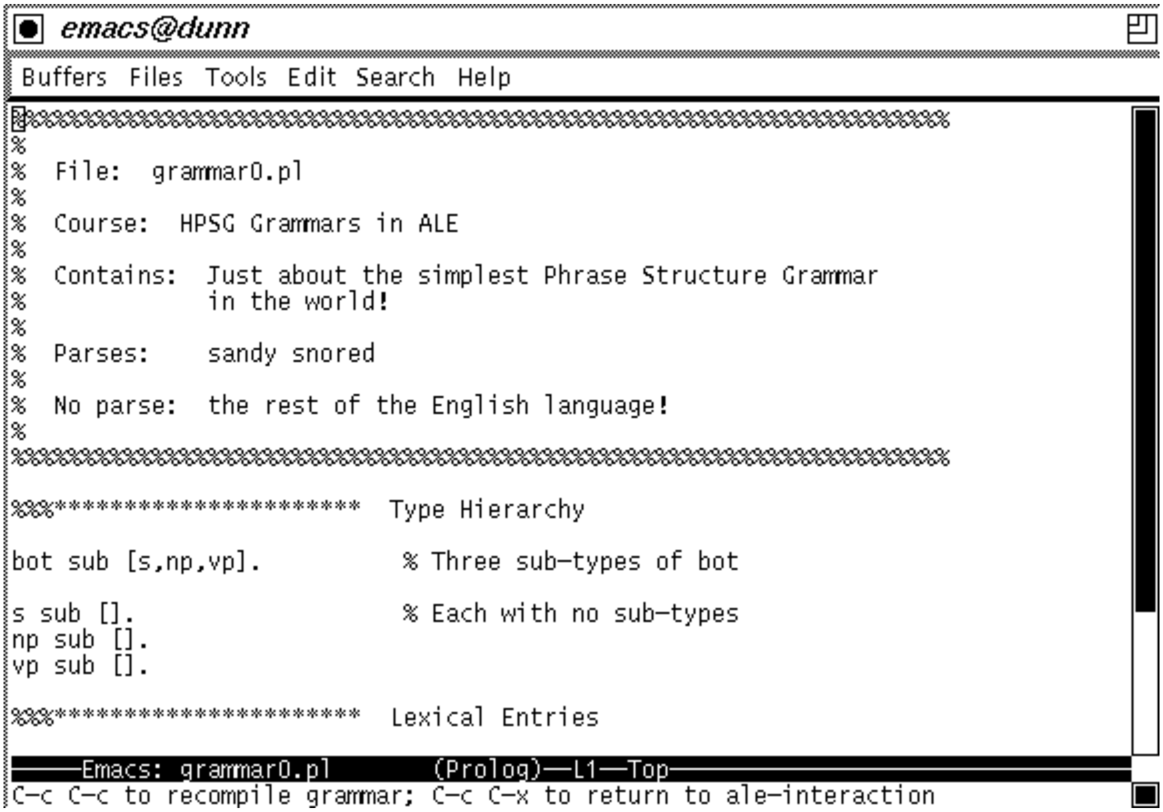
This is the main interaction buffer. Many of the options are fairly self-explanatory -- whole grammars, and parts of grammars (the lexicon, for instance), can be loaded and compiled. Loaded grammars can be edited, and strings can be tested to see whether or not they parse. We shall see some of the other options soon -- for the moment, we can load the grammar `grammar0.pl` using the `S` (Compile Signature and Grammar) command:



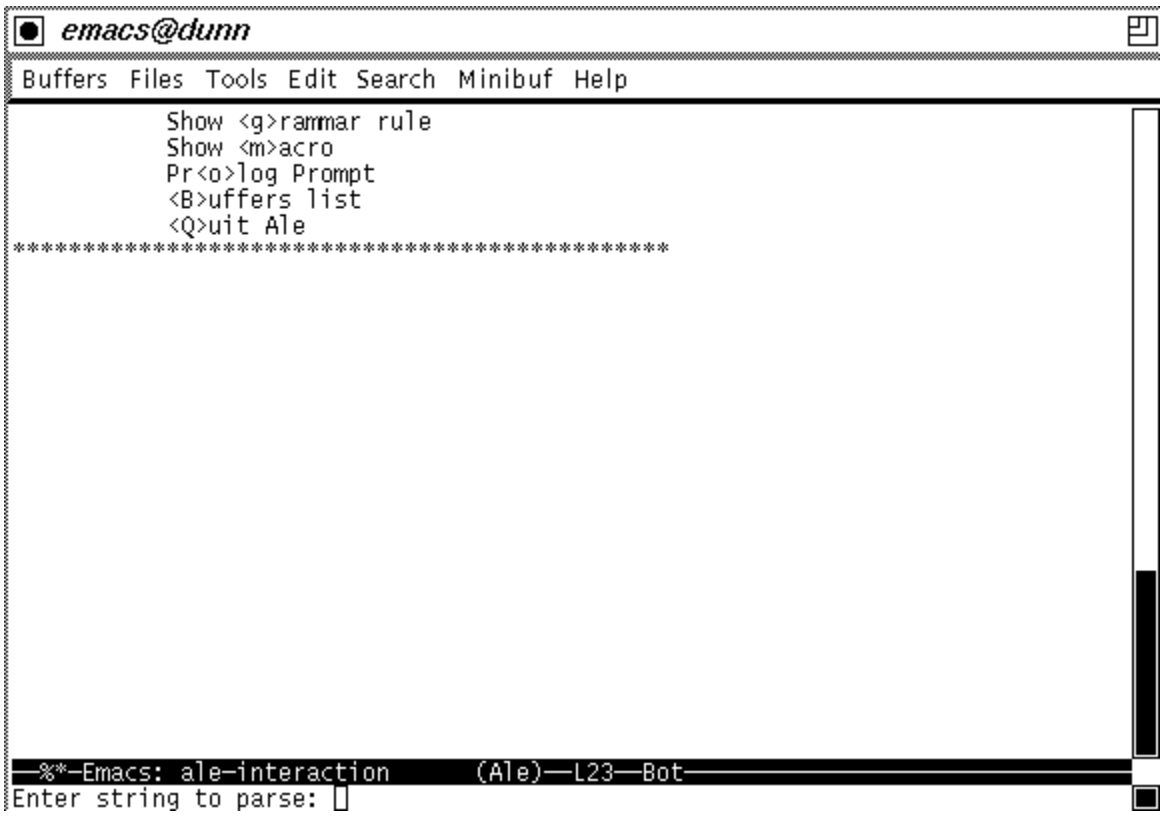
At this point the path to the appropriate file should be typed into the minibuffer. The result will be a number of messages indicating that various files are being loaded, followed by something like the following screen:



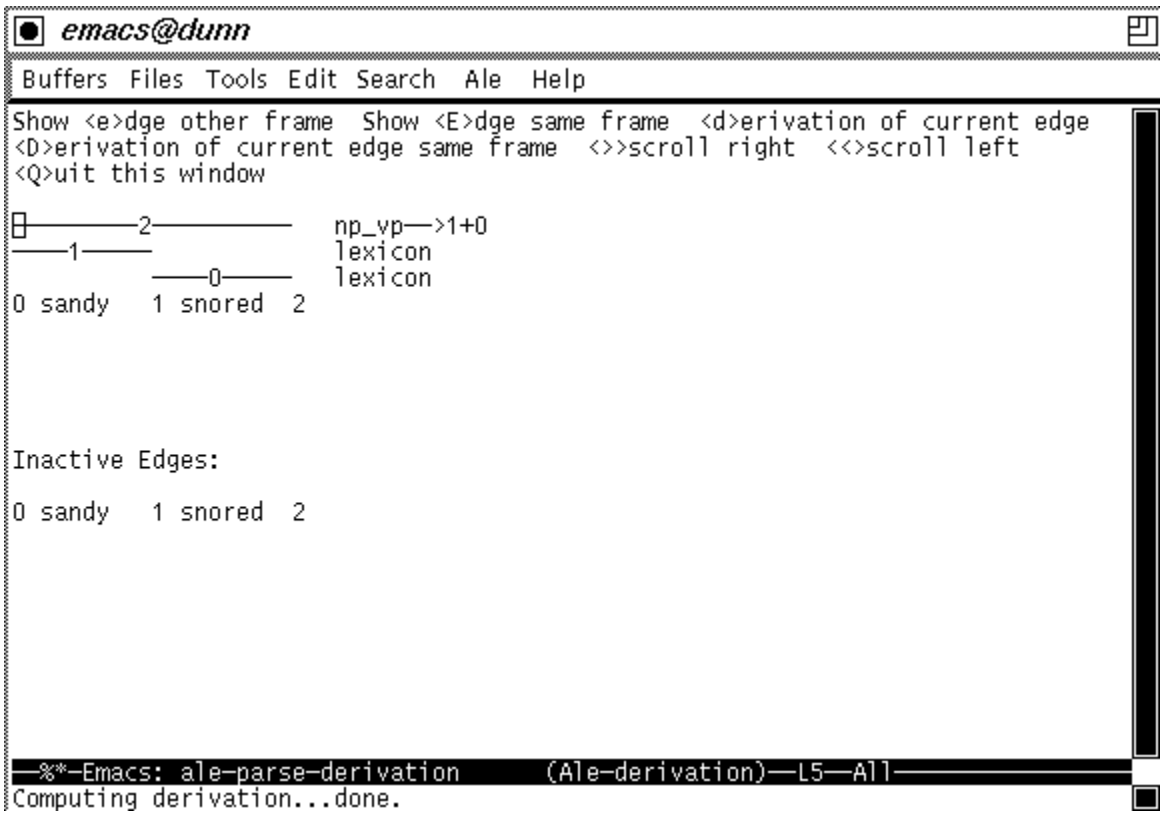
We can now switch back into the interaction buffer (`CTR-X-b` is one option, or we can use the buffer list in the menubar -- here I've also used `CTRL-X-1` to make it the only buffer on view) and have a look at the grammar using the `e`(dit) command, which asks for the name of the file to be edited. The default is the current grammar, so `RETURN` will result in the following:



The contents of this file are described below -- the main points are that it can be edited and saved from this buffer, and then reloaded from the interaction buffer. After returning to the interaction buffer (using `CTRL-X-b` or the buffer menu as before), the grammar can be used to parse a string using the `p`(arse) command:



Typing *sandy snored* at the prompt, followed by RETURN, should result in a message that this string has 1 parse. We can inspect this parse using the d(erivation) command, which displays information on the rules used to parse the string:



Various options are available for looking further at the details of the parse, but the main point to note is that the grammar has combined *sandy* and *snored* using the `np_vp` rule. We can inspect the parts of the parse by placing the cursor on the object we wish to see and typing `e`(dge), which will open another window to display the analysis of the object in question. In the current case the word *sandy* is simply entered in the lexicon as an NP, so looking at it will simply result in the following:

When the grammar is complicated, this option is very useful. This window can be removed using the `q`(uit) command, and typing `q` again in the derivation window will return control to the main interaction window.

You can exit using `CTRL-X-c` or via the file option in the menubar. Other aspects of the interface are mentioned in passing in the course itself, for now we shall look briefly at how to escape from a Prolog error and then at the basic contents of an ALE grammar, using the file `grammar0.pl` to illustrate the main points.

Errors

It's more or less inevitable that you will experience Prolog errors on occasion. The simplest method of escaping is to type `CTRL-C` (or `CTRL-C-CTRL-C` from inside Emacs) which will produce the interruption line:

```
Prolog interruption (h for help)?
```

Type `a` here to abort, and you'll get back to the normal Prolog/ALE prompt:

```
Prolog interruption (h for help)? a
{Execution aborted}
| ?-
```

ALE Grammars

This section looks at some basic aspects of representing grammatical information in ALE -- much more machinery will be introduced as the course progresses. The example grammar `grammar0.pl` accompanies this discussion and contains the very simple grammar used in the previous session.

The only particularly unusual aspect of `grammar0` is probably the type hierarchy, and this is partly the subject of the first part of the HPSG course. For the moment, it is enough to know that all the descriptions used in the grammar must be given a type, and in this case that means the objects `s`, `np`, and `vp`.

The Lexicon

Lexical information in ALE is represented as shown below:

```
sandy ---> np.
snored ---> vp.
```

The main thing to note is the character sequence `--->` which is used as a rewrite symbol. The lexicon above is as basic as possible -- typically the description following the rewrite symbol is a complicated feature structure in HPSG.

In order to have a working grammar, we now only require a method of combining words into larger structures, and this is done by the grammar rules.

Grammar Rules

Rewrite rules can also be fairly straightforward. The NP-VP rule that we saw in the interface discussion above is entered as shown below:

```
np_vp rule
s
===>
cat> np,
cat> vp.
```

First of all the rule is named (`np_vp`), and this is followed by the atom `rule` which specifies the type of information for the ALE compiler. We shall see other types of specification later. The subsequent lines contain the body of the rule; firstly the category of the mother, then the rule rewrite symbol, and then two lines -- each starting with the `cat>` symbol -- which indicate the categories of the daughters. There can be any number of daughters, and the information in the mother and daughter categories can be complex. One important point is that the order of the `cat>` statements is fixed -- the objects can only be parsed in the order NP-VP.

The `cat>` symbols are there because rules can contain other statements (`goals' -- much in the same manner as Prolog DCGs) which are flagged differently. The punctuation, here as always, is extremely important -- note that the mother is not followed by any punctuation, but that all daughters are followed by commas except the last, which is completed by a fullstop.

Exercises

These are some simple exercises on running the system and editing grammars. The aim is to extend the basic grammar in `grammar0.pl` to include transitive and ditransitive verbs as well as slightly less basic NPs -- please go ahead with this now before going on to the first part of the HPSG course proper.

Please *always* add comments to your files explaining what your grammar does, and doesn't, do. Lists of example sentences are ideal in assessing how the grammar works -- so include them in the comments.

1. Start ALE, load the grammar file `grammar0.pl`, and check that you can parse *sandy snored*. Edit the grammar to add new lexical entries as follows:

```
kim ---> np.
fell ---> vp.
```

Check that you can parse *kim snored*, *sandy fell*, and so on.

2. Extend the grammar to handle transitive verbs by adding a new grammar rule of the general form *VP →Verb NP*, a new type `verb`, and a new lexical entry for *likes*:

```
likes ---> verb.
```

Verify that you can parse *sandy likes kim*.

3. Add a rule of the form *NP →Det N*, new types `det` and `noun`, and suitable lexical entries for *the*, *bone* and *dog*, for example:

```
the ---> det.
```

Check that *the dog snored* parses ok.

4. Extend the grammar with a rule for ditransitive verbs, and a suitable lexical entry for *gave*, so that sentences like *kim gave the dog the bone* can be parsed. Does your grammar also parse * *kim likes the dog the bone* and * *sandy gave kim*? If so, how could these be blocked?

3 Type Hierarchies and the Subject-Head Schema

Introduction

This is the first part proper of a course in developing HPSG grammars in ALE. You should already have read one of the preliminary papers on running ALE, `emacs-ale` or `emacs-ale-ol`, and completed the enclosed exercises.

From now on, we'll be concentrating on building an HPSG grammar which is more or less compatible with the assumptions in Chapter 9 of the Pollard & Sag book. We shall start by looking at type hierarchies and at two basic schemas, the subject-head schema and the head-complement schema. The exercises at the end of this paper introduce the notion of lists to the grammar, and must be undertaken if these are to be fully understood.

Before looking at the basic HPSG machinery, next section introduces some general points about the theory.

HPSG

As mentioned, this course will develop grammars which are broadly in line with HPSG theory as outlined in chapter 9 of Pollard & Sag. This means that a number of the feature structures used are incompatible with those in the earlier chapters -- notably, the SUBCAT feature is replaced by the SUBJ and COMPS features -- however, most of the earlier discussion is relevant.

We shall develop the grammar as incrementally as possible, building feature structures to cover new data where necessary, and we shall start with very simple representations. Unfortunately, these desiderata are occasionally in conflict -- it is sometimes necessary to work with intermediate representations which are incompatible with the assumptions in the book. For example, the category information in the theory is actually only a part of the total syntactic and semantic representation of a sign, and we shall need to complicate the type hierarchy to take account of this. However, to keep things simple, we shall start by assuming that signs introduce the category feature directly and introduce the `correct' analysis later.

The general structure of HPSG theory is that a grammar is a collection of different types of declaration. These will include a declaration of all the `types' to be used in descriptions, a list of grammar rules (`schemas'), a lexicon, some general grammatical principles, and so on. ALE allows most aspects of the theory to be captured fairly straightforwardly, and in fact one of the main aims of the course is to develop skills in translating HPSG representations directly into ALE.

Perhaps the most important aspect of the theory in comparison with earlier phrase structure approaches -- GPSG, for instance -- is that lexical items contain information on what kind of subject they take, what kind of complements they take, whether or not they take `specifiers', and so on. Thus most of the work which is done by grammar rules in a typical PSG is handled by the lexical entries and by a few generalised rules in HPSG.

The following section introduces one of the most important kinds of declaration in HPSG theory - the type hierarchy.

Type/Inheritance Hierarchies

The standard method of representing grammatical information in modern computational grammar theories is the Attribute-Value Matrix (AVM). All of the descriptions used in ALE can be represented as AVMs, with the added restriction that each representation must have a TYPE. For instance, this is a typical AVM representing part of the description of the verb *likes* :

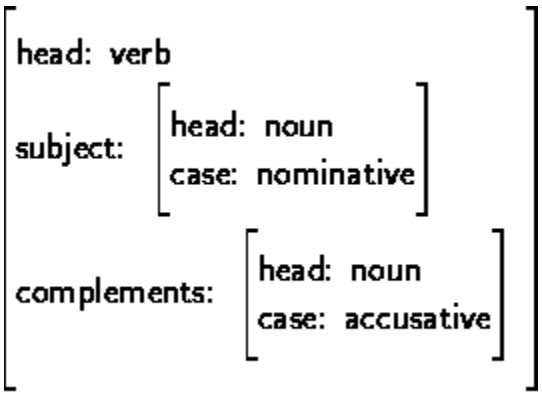
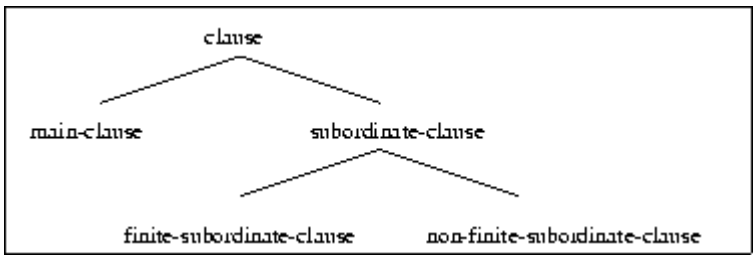


Fig1: AVM Representing Information about a Verb

This object *must* have a type assigned to it -- the information shown relates to the category of the word, and so typically this package of features and values would be given the type `category'.

HPSG assumes a very important role for types -- for example, we can imagine a hierarchy of clauses as follows:



Here finite subordinate clauses are a sub-type of subordinate clauses, and so on. Any properties of the general type, 'clause' are assumed to belong to the sub-types, while the subtypes themselves may differ in the kinds of features which are appropriate, and so forth. This allows precise and concise linguistic generalisations to be stated, and it also allows an efficient computational implementation -- a feature exploited by ALE .

We shall begin our HPSG grammar by working through the type hierarchy in the grammar file (grammar1.pl) which accompanies this part of the course. The type declaration starts with the following line:

```
bot sub [sign,cat,head].
```

The first point to note is that there is a special type `bot` which is the unique most general type -- in other words, everything is a sub-type of `bot`. The grammar defines three subtypes, each of which plays a central role in HPSG theory. Firstly, a Sign is the top-level representation of words and phrases, representing their phonology, syntax, and semantics -- for precise details, consult the Pollard & Sag book and/or the ALE manual, but it is enough here to know that this type is just a general term covering all the information representing an object in the theory.

Category information is the general syntactic description of a grammar object -- what complements it takes, and so on. Finally, the Head type contains the basic categorial information -- whether the object is a noun, verb, adjective, or whatever.

Types can introduce features -- in Fig 1 above, the category information includes the `head`, `subj`, and `comps` features. SUBJECT and COMPLEMENTS are two of the three `valency' features assumed in the theory -- the third, SPECIFIER, we shall see soon. For the moment we shall only use the SUBJECT and HEAD features to characterise signs:

```
cat sub []
   intro [head:head,
          subj:head].
```

Note that there is no full-stop at end of the first line -- it indicates the end of the whole declaration. Notice also the syntax of the feature-name -- feature-value statements, all of which are enclosed in square brackets and separated by commas. The feature-value pairs themselves are separated by colons. Finally, the formatting into multiple indented lines is simply to make the whole thing easier to read.

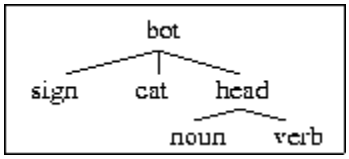
Each feature must indicate what type its value is -- in this case, both `head` and `subj` take values of type `head`, and this also must be declared:

```
head sub [noun,verb].
```

All types must be declared, and we can assume for the moment that `noun` and `verb` are `atomic' types with no subtypes or features:

```
noun sub [].
verb sub [].
```

Note that we do not have to include `noun` and `verb` in the list of subtypes of `bot` -- the ALE compiler understands the transitive nature of the subtyping relationship. Thus, `head` is a subtype of `bot` and the new types are subtypes of `head`, so they are automatically subtypes of `bot`:



One other feature is introduced in the type hierarchy in `grammar1.pl`:

```
sign sub []
   intro [cat:cat].
```

This states that signs introduce the feature `cat` with value type `cat` -- clearly there is room for confusion here, and it would be possible to use different symbols where appropriate. However, HPSG often employs this kind of representation, so one may as well get used to it right from the start.

The most important general points to note are:

- we are working with features and types
- features and types are entirely different notions
- features take types as values
- we can specify `paths' to get to particular values

To illustrate the last point, note that we can use the following path to specify that some sign represents a noun:

```
cat:head:noun
```

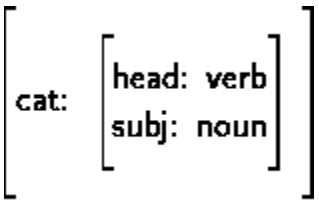
To see this kind of thing in action, let us now turn to the lexical entries contained in the file `grammar1.pl`.

Lexical Entries

The lexicon in grammar1.pl contains the following:

```
sandy ---> cat:head:noun.

snored ---> cat:(head:verb,
                 subj:noun).
```

Looking at the entry for *snored* first, we can represent the same information using the following AVM:



The SUBJECT feature in HPSG contains information on what kind of thing is appropriate as subject of the category in question -- so here we are saying that the subject of *snored* must be a noun. The HEAD feature contains the core description of the object, which in the above examples is simply `noun` or `verb` at the moment.

The formatting in the ALE version is once again intended as an aid to the reader, reflecting as closely as possible the AVM. The syntax of the ALE statements should be noted carefully. Multiple feature-name -- feature-value statements are contained in round brackets, and it must be emphasised that the commas between the statements represent conjunction. Thus the entry for *snored* says that its head is `verb` *and* that its subject is `noun`. This must not be confused with the comma which appears in Prolog lists, where it represents the list separator -- we shall see examples of this soon. As the comma inside round brackets represents conjunction, the order of the statements does not matter. So we could have had:

```
snored ---> cat:(subj:noun,
                 head:verb).
```

The entry for *Sandy* apparently states only that it has the head feature `noun` -- the defining characteristic of a noun. However, this is deceptively simple. Using the `lex` command, or the `w` command in the Emacs interface, we can see what the full representation of *Sandy* is:

```
WORD: sandy
ENTRY:
sign
CAT cat
```

```
    HEAD noun
    SUBJ head
```

The SUBJECT feature has clearly been added with the value `head`. This is because the type declaration states that the type `category' has two features:

```
cat sub []
   intro [head:head,
          subj:head].
```

This says that all structures of type `cat` have the features `head` and `subj`. If these features are not given specific values in the entries, then ALE gives them the most appropriate value as declared in the type hierarchy

We shall see shortly that this kind of underspecification -- although it is often useful -- can cause problems. For the moment, in order to use the information that *snored* takes a noun as a subject to allow the sentence *Sandy snored* to be parsed, all we now need is a grammar rule which combines two categories as long as the SUBJECT feature on the second one matches the category specification on the first. This is the precisely what the subject-head schema does, as the following section shows.

Schemas

It was noted above that schemas are an important kind of declaration in the theory. These are effectively the grammar rules -- specifying the output of combining one or more categories in a particular way. There are currently 6 schemas in all in the theory, representing generalisations over all the grammar rules which are typically found in phrase structure grammars. For example, in the preliminary exercise, the following PS rules are likely to have been used:

S →NP VP
VP →V
VP →V NP
VP →V NP NP

HPSG assumes that these can be subsumed by two schemas -- the subject-head schema in place of the first rule and the head-complement schema for the rest. In the latter case, the lexical entries for verbs will specify what their complements are, and a single rule is enough to combine the lexical items with suitable objects.

In grammar `grammar1.pl`, the following version of the subject-head schema is implemented:

```
subject_head rule
(cat:head:verb)
===>
cat> (cat:head:Head),
cat> (cat:subj:Head).
```

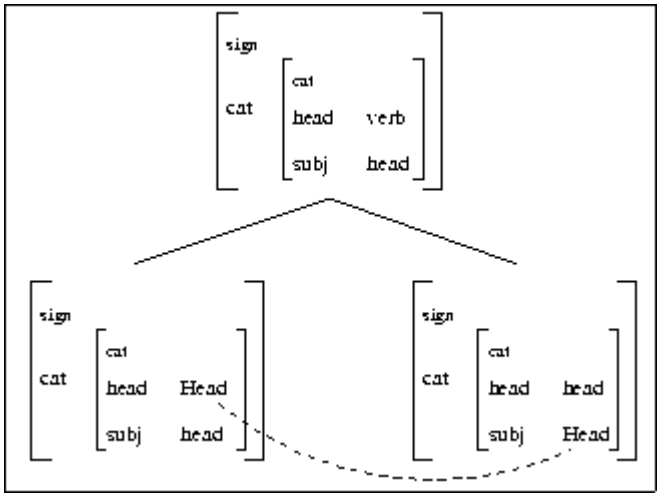
Mostly, the syntax should be familiar from the preliminary exercise, although one important new aspect is the use of Prolog variables. Symbols which start with upper-case letters are variables, and variables with the same name must unify -- so the `Head` variable on the daughters must have the same value.

The rule, like the lexical entry for *Sandy*, is again deceptively simple -- because of the type declarations, the rule which ALE is actually using includes other features. Using the `rule(subject_head)` command, or the `g` command in the Emacs interface (followed by the rule name), we can check what the internal ALE version of the subject-head rule is:

```
RULE: subject_head

MOTHER:

  sign
  CAT cat
      HEAD verb
      SUBJ head

DAUGHTERS/GOALS:

CAT  sign
     CAT cat
         HEAD [0] head
         SUBJ head

CAT  sign
     CAT cat
         HEAD head
         SUBJ [0]
```

Here it is clear that the mother and the first daughter have `subj` specifications not mentioned in the input rule, and also that the second daughter has the `head` feature in addition to `subj`. As with the lexical entry for *Sandy*, this is because of the type declaration for `cat`. In this case, all the features have the value `head`, which of course means either of the basic sub-types `noun` or `verb` in the current grammar.

So, the full AVM representation of the subject-head schema is:



Note that the mother feature specification (which in some sense represents the `output' of the rule -- the result of combining the subject and head verb) is specified to be a verb. This is too specific -- we may want other kinds of category to be combined using this schema -- but for the moment we shall accept this analysis.

Finally, note that in order to check if a lexical entry fits with a category in a rule, it is occasionally necessary to have a close look at the underlying ALE structures -- for example, check that the full representation for *Sandy* which we saw in section 1.4 above will unify with the first category in the subject-head rule as printed by the *rule* command.

Exercises

The grammar in `grammar1.pl` will parse **snored Sandy*, **Sandy Sandy*, **Sandy Sandy Sandy*, and so on, the problem being that *snored* and *Sandy* are taken to be possible subjects of *Sandy*. This exercise aims to get rid of these parses by introducing a new type `list` and correctly specifying that nouns take empty lists as subjects, in contrast to verbs.

1. Start ALE, load the grammar file `grammar1.pl`, and check that you can parse *Sandy snored*. Verify that you can also parse **snored Sandy*, **Sandy Sandy*, and so on, and make sure you understand what the problem is.

2. Part of the solution to the problem involves the introduction of a new type to the grammar -- the `list` type. HPSG makes use of lists in many ways, and one of the basic assumptions is that features such as SUBJECT and COMPLEMENTS take list of categories as their values -- so COMPLEMENTS, for instance, might be the list <NP, NP > in the case of the verb *give* (as in *give John the book*), or the list <NP, PP > (as in *sold the book to John*). HPSG is arguably over-general in assuming that lists of subjects are possible -- but no harm is caused by going for the most general statement.

To get lists working in ALE, we must include suitable statements in the type hierarchy. Firstly, add `list` to `bot`:

```
bot sub [sign,list,cat,head].
```

Now change the value of the `subj` feature so that it takes lists instead of `head` types.

Lists will have two subtypes -- empty lists and non-empty lists, so add this statement too:

```
list sub [e_list,ne_list].
```

Non-empty lists are objects which have a head and a tail, both of which we shall represent using features. We shall assume for the moment that the head of a list can be any sign, while the tail must be a list. The following should therefore be added to your grammar:

```
ne_list sub []
  intro [hd:sign,
         tl:list].
```

Finally, all we need is the statement that an empty list is a basic type:

```
e_list sub [].
```

Reload the grammar with these changes. You should get a complaint that the lexical entry for *snored* is now faulty -- can you see why? We shall fix this problem shortly. For the moment, check the new hierarchy using the `show_type` command:

```
| ?- show_type(list).

TYPE: list
SUBTYPES: [e_list,ne_list]
SUPERTYPES: [bot]
IMMEDIATE CONSTRAINT: none
MOST GENERAL SATISFIER:
     list

yes
| ?- show_type(ne_list).

TYPE: ne_list
SUBTYPES: []
SUPERTYPES: [list]
IMMEDIATE CONSTRAINT: none
MOST GENERAL SATISFIER:
     ne_list
     HD sign
        CAT cat
            HEAD head
            SUBJ list
     TL list
```

Have a close look at these to make sure you understand the results of the queries -- the `most general satisfier' can be thought of as the type which most precisely describes the

object you have asked about. This is obviously straightforward in the current situation, but the questions can be more complicated, as we shall see.

3. We can now use lists in the grammar. Firstly, we want to say that nouns don't take subjects, and so we shall simply add the specification `subj:[]` to the lexical entry for *Sandy*. Do this now and check the result using `lex` or `w`:

```
WORD: sandy
ENTRY:
sign
CAT cat
    HEAD noun
    SUBJ e_list
```

4. Now we want to correct the lexical entry for *snored*. The problem is that the entry says that the value of `subj` is `noun` -- something of type `head` -- while the type declaration says that the value of `subj` should be a list. Hence the complaint that the entry is `unsatisfiable'.

To fix the entry, we shall effectively say that the subject of *snored* (and of all verbs, at least in English) is a list containing one element, which is a noun. To do this, we shall use the Prolog list syntax:

```
[a,b,c, ...]
```

As mentioned earlier, the commas here separate items in the list. This unfortunately contrasts with their use as conjunction markers inside round brackets in ALE representations and some care is required. In order to pick out the first thing in the SUBJECT list, we simply refer to it directly:

```
cat:head:subj:[FirstItem]
```

This should be a noun in the case of *snored*, and so the actual lexical entry will be:

```
snored ---> cat:(head:verb,
                 subj:[cat:head:noun]).
```

Check that everything is ok using `lex` or `w`:

```
WORD: snored
ENTRY:
sign
CAT cat
    HEAD verb
    SUBJ ne_list
         HD sign
            CAT cat
                HEAD noun
                SUBJ list
         TL e_list
```

4. All we need to do to get the grammar working is alter the subject-head schema to take account of the fact that the SUBJECT feature is now list-valued. The statement we want is that the head feature on the subject must match the head feature on the first element of the head category's SUBJECT list:

```
subject_head rule
(cat:head:verb)
===>
cat> (cat:head:Head),
cat> (cat:subj:[cat:head:Head]).
```

Check that the rule is ok using the `rule` or `g` command, which should produce the following for the daughter categories:

```
DAUGHTERS/GOALS:

CAT  sign
     CAT cat
         HEAD [0] head
         SUBJ list

CAT  sign
     CAT cat
         HEAD head
         SUBJ ne_list
              HD sign
                 CAT cat
                     HEAD [0]
                     SUBJ list
              TL e_list
```

Check that the previously problematic **snored Sandy*, **Sandy Sandy*, and so on, are now blocked, and that you are able to parse *Sandy snored* with the following feature structure:

```
sign
CAT cat
    HEAD verb
    SUBJ list
```

Note the value of SUBJECT here -- can you see where this might cause problems? (Try parsing **snored Sandy snored* -- are there any other over-generations?)

Solutions

The grammar in `grammar2.pl` contains solutions to most of the above problems.

4 The Head-Complement Schema

Introduction

This is the second part of a course on HPSG and ALE. Here we shall look initially at the use of macros to simplify rules and lexical entries. We then investigate a new schema -- the head-complement rule -- which we shall use in the analysis of sentences such as *Sandy likes Kim*.

Macros

So far, we have been using the following lexical entries:

```
sandy ---> cat:(head:noun,
                subj:[]).

snored ---> cat:(head:verb,
                 subj:[cat:head:noun]).
```

With a large lexicon, having to specify full paths everywhere will clearly become time-consuming, and the objects will be difficult to read. It would be useful to have a shorthand method of picking out and referring to particular descriptions, and this is exactly what macros allow. For instance, we can define an NP macro as follows:

```
np macro
  cat:(head:noun,
       subj:[]).
```

Here we name the statement (`np`), and follow this with the `macro` keyword which identifies the statement for the compiler. The body of the statement is just the feature structure which is associated with the name. In this case we are saying that an NP is a nominal object which has found a subject -- or, at least, which is not looking for a subject. We can now use the macro to simplify the lexical entries:

```
sandy ---> @ np.

snored ---> cat:(head:verb,
                 subj:[@ np]).
```

The use of a macro is obviously signalled by the `@` sign. Notice that in the case of the subject of *snored*, the new entry has more information than the old one, which simply specified that the subject should be nominal.

The Head-Complement Schema

So far we can only handle intransitive verbs such as *snored*. In order to extend the grammar to include transitive and ditransitive verbs, we need to implement a new schema which will allow heads to combine with complements. This will allow us to say that a verb such as *likes* takes one NP complement while *gives* can take two (*gives Sandy books*, and so on).

HPSG assumes the this information is stored in a second valency feature -- COMPLEMENTS. Like SUBJECT, it is list-valued, and so the ALE type hierarchy will be extended as follows:

```
cat sub []
   intro [head:head,
          subj:list,
          comps:list].
```

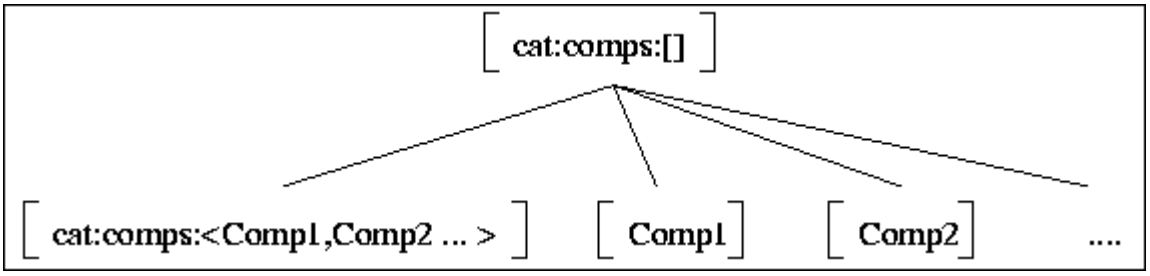
We will need to specify values for this feature on everything in the lexicon. In the case of nouns, we will assume for the moment that all NPs take no complements and include the specification in the NP macro:

```
np macro
  cat:(head:noun,
       comps:[],
       subj:[]).
```

For intransitive verbs such as *snored*, COMPLEMENTS should be an empty list. We shall now extend our lexicon with entries for *likes* and *gives*, and these have non-empty `comps` lists:

```
snored --->(cat:(head:verb,
                 subj:[(@ np)],
                 comps:[])).

likes ---> (cat:(head:verb,
                 subj:[(@ np)],
                 comps:[(@ np)])).

gives ---> (cat:(head:verb,
                subj:[(@ np)],
                comps:[(@ np),(@ np)])).
```

The head-complement rule should match a list of categories with the value of the COMPLEMENTS feature on the head daughter, as the following schematic AVM shows:



The `comps` feature on the head here has the list `<Comp1,Comp2, ... >` as value -- a list containing any number of items. The rule shows this list being matched against a suitable sequence of categories to produce an object which is fully satisfied as far as its complements go (captured by the `comps:[]` specification on the mother). ALE provides the `cats>` symbol to allow this kind of list matching, and the general form of the rule will be:

```
head_complement rule
(cat:comps:[])
===>
cat> (cat:comps:Comps),
cats> Comps.
```

The `cats>` specification automatically attempts to match a list of categories with the value of `comps` on the head. There is a problem with this kind of statement as it stands, however -- the parser will fall into an infinite loop because the `Comps` variable could represent an empty list. In order to avoid this, we can simply add the requirement that the list of complements should be non-empty:

```
head_complement rule
(cat:comps:[])
===>
cat> (cat:comps:Comps),
cats> (Comps, ne_list).
```

Some care should be taken with the descriptions used here and in the subject-head schema:

```
subject_head rule
(cat:head:verb)
===>
cat> (cat:head:Head),
cat> (cat:subj:[cat:head:Head]).
```

Both `subj` and `comps` are list-valued, of course. In the subject-head rule, though, we pick out the first element in the list and match it against the single category introduced by the `cat>` symbol. In the head-complement rule, we match the whole list which is the value of `comps` with the list of categories introduced by the `cats>` symbol. Time must be taken to ensure that what is matching what is fully understood and that the format of the Prolog list representations are clear. To emphasise this, the following two rules, in contrast to the head-complement and subject-head rules above, are wrong:

```
head_complement rule
(cat:comps:[])
===>
cat> (cat:comps:[Comps]),
cats> (Comps, ne_list).

subject_head rule
(cat:head:verb)
===>
cat> [cat:head:Head],
cat> (cat:subj:[cat:head:Head]).
```

In the head-complement rule, an attempt is being made to match the first element in the `comps` list on the head with a list of categories, which will not work. In the subject-head rule the first element in the `subj` list on the head is being unified with the first element in the subject category, which is another mis-match.

Finally, notice that the following version of the head-complement rule will work, but only one complement will be matched:

```
head_complement rule
(cat:comps:[])
===>
cat> (cat:comps:[Comps]),
cats> [Comps].
```

We could parse *Kim likes Sandy* with this rule, but not *Kim gives Sandy books*, as only the first element in the COMPLEMENTS list is being matched against the first element in the `cats>` specification.

Exercises

These exercises introduce macros and the head-complement rule. Sentences such as *Kim likes Sandy* and *Kim gives Sandy books* are parsed. The grammar file `grammar2.pl` provides a starting point.

1. Load the file `grammar2.pl` and make sure you understand the contents. The most important aspect is the analysis of lists, as described in part 2 of the course. Now substitute the NP macro discussed in the text for the path descriptions which were previously used in the lexical entries for *Sandy* and *snored*. Check that you can still parse *Sandy snored*.

2. The grammar as it stands will parse **snored Sandy snored* and **Sandy Sandy snored* with *snored* and *Sandy* as the subject of *Sandy snored*. This is because the result of parsing *Sandy snored* is the following feature structure:

```
sign
CAT cat
    HEAD verb
```

```
    SUBJ list
```

So far, we have said nothing about the SUBJECT feature on the mother, and so the grammar thinks that *Sandy snored* can effectively have anything as a subject. The answer to this is to ensure that the result of combining a subject with a head is a category which is satisfied as far as this feature is concerned -- so the mother should have the specification `subj:[]`. Add this to the subject-head rule and verify that the two ill-formed sentences mentioned above no longer parse.

3. Add the feature `comps` to the `cat` type in the hierarchy as described in the text. Reload the grammar and use the `show_type` predicate to check that the `cat` type has the appropriate specification:

```
TYPE: cat
SUBTYPES: []
SUPERTYPES: [bot]
IMMEDIATE CONSTRAINT: none
MOST GENERAL SATISFIER:
     cat
     COMPS list
     HEAD head
     SUBJ list
```

4. Add the `comps` feature with appropriate values to your existing lexical entries and macro. Now add new entries for the transitive verb *likes*, the ditransitive verb *gives*, and the nouns *kim* and *books*. Check the entries are ok -- for example (using `lex` or `w`):

```
WORD: likes
ENTRY:
sign
CAT cat
    COMPS ne_list
          HD sign
             CAT cat
                 COMPS e_list
                 HEAD noun
                 SUBJ e_list
          TL e_list
    HEAD verb
    SUBJ ne_list
         HD sign
            CAT cat
                COMPS e_list
                HEAD noun
                SUBJ e_list
         TL e_list
```

Check that you can parse *Kim snored*.

5. Implement the head-complement schema as described in the text and check that you can parse the VPs *likes Sandy* and *gives Sandy books*. Notice, however, that if you try to parse a full sentence, the system wil produce an error of the form:

```
error: cats> value with sort, list is not a valid list argument1
Parse(s)
```

(If you get stuck in an temporary error buffer in Emacs trying this, just type `CTRL-X-K` and `RETURN` to kill it.) The problem is actually that underspecification in the specifier-head rule is confusing the parser and causing the head-complement rule to be called with incompatible feature structures. At the moment, we are only specifying the `subj` feature on the head daughter in the subject-head schema, and we should add a `comps` value too. HPSG assumes that subjects are added after all the complements have been found, and so the rule requires a `comps:[]` specification on the head:

```
cat> (cat:(subj:[cat:head:Head],
           comps:[])).
```

Now sentences such as *Kim likes Sandy*, and *Kim gives Sandy books*, should parse -- verify this.

6. Check if your grammar parses **likes snored*. If not, does it parse **snored likes Kim* and **likes gives Kim books*? It's possible to have two analyses for the last sentence, in fact. If any of these are accepted by your grammar, identify the problem (or problems) and think about how you might you fix it.

Solutions

Solutions to most of the above questions can be found in `grammar3.pl`.

5 The Specifier-Head Schema

Introduction

In this third part of the course on HPSG and ALE we shall look firstly at one of the general principles of HPSG and at its implementation in ALE. The third valency feature -- SPECIFIER -- is then introduced, and the specifier-head schema is discussed in detail. The discussion includes a look at the distinction assumed in the theory between substantive and functional categories.

Principles and Goals

HPSG assumes that general principles apply to the schemas. One of the most important of these is the Head Feature Principle (HFP), which is formulated in the HPSG book as follows (p.34):

Head Feature Principle

The HEAD value of any headed phrases is structure-shared with the HEAD value of the head daughter.

Pollard & Sag suggest that the effect of this is to guarantee that headed phrases really are `projections' of their head daughters. The arguments for this are more or less identical to the reasons put forward in support of X-bar syntax, the basic idea being that there is a head category in a phrase which effectively characterises the whole object -- hence VPs are verbal because they have a verbal head, NPs are nominal because of the head noun, and so on.

We have encountered some very practical reasons for having a principle of this kind -- the schemas which we have implemented allow strings such as **likes gives Kim books* in which *gives Kim books* is a complement of *likes*. This is because the head feature on *gives* is not being passed to the VP, and the resulting underspecification allows *gives Kim books* to be analysed as a nominal.

So, we clearly want to ensure that VPs are specified as being verbal, NPs as nominal, and so on. This could be done explicitly in the ALE rules, for instance:

```
head_complement rule
(cat:(head:HeadHead,
      comps:[]))
===>
cat> (cat:head:SubjectHead),
cat> (cat:(head:HeadHead,
           subj:[cat:head:SubjectHead])).
```

Here we have added a `head:HeadHead` specification to the categories of the mother and head daughter, thus ensuring that the lexical head's features are identical with the mother category's, as suggested by the HFP. However, as the quote from Pollard & Sag suggests, HPSG assumes that the correspondence should be a universal principle, applying to every schema that has a head. In order to implement this in ALE we shall use `goal>` statements in the schemas in much the same way as Prolog DCGs enforce restrictions on categories in rules. Thus, in the head-complement schema, we would like to say simply that the mother category inherits the same head features as the head daughter. To get this to work, we firstly need some method of identifying the mother and head categories, and this is done in the same way as the `Comps` list was introduced in the `cats>` specification:

```
head_complement rule
(Mother, cat:comps:[])
===>
cat> (HeadDtr, cat:comps:Comps),
cats> (Comps, ne_list).
```

Remember that the commas inside round brackets are conjunctions, which means that ALE assumes that the initial variables `Mother` and `HeadDtr` characterise the whole of the relevant categories. The descriptions following the comma work as before. Using the `goal` feature, we can now impose the HFP on the above schema:

```
head_complement rule
(Mother, cat:comps:[])
===>
cat> (HeadDtr, cat:comps:Comps),
cats> (Comps, ne_list),
goal> head_feature_principle(Mother,HeadDtr).
```

Note that the full-stop which was at the end of the `cats>` line is now a comma, of course. All that's needed now is a statement of the HFP itself, and this is just the requirement that the `head` features match:

```
head_feature_principle(cat:head:Head, cat:head:Head) if true.
```

We want this to apply to all schemas, of course, so similar changes must be made to the subject-head rule. Note also that with the subject-head rule we can identify the subject daughter using a variable and match this with the whole category which is the value of the SUBJECT feature on the head:

```
subject_head rule
(cat:subj:[])
===>
cat> SubjDtr,
cat> (cat:(subj:[SubjDtr],
           comps:[])).
```

With the HFP added, this version is obviously more concise than the previous one -- and more accurate too, in that we are matching all of the value of `subj` with the subject daughter rather than just the head feature. Note also that we can remove the stipulation that the mother is verbal -- the HFP will take care of that.

The Specifier-Head Schema

So far we have looked at the combination of subjects with heads and at the analysis of heads and complements. HPSG assumes a third `valency' feature in categories (SPECIFIER) in order to handle some of the facts concerning specifier-head relationships, and a new schema -- the specifier-head schema -- to implement the combination.

The specifier-head schema requires a two-way restriction -- specifiers restrict their heads, and heads restrict their specifiers. Thus, for instance, a specifier like *this* selects singular nouns, while at the same time a common noun selects a determiner as a specifier. HPSG also assumes that complementisers like *that* and *for* have properties in common with determiners, and so the entry for *that* will specify that it selects sentences, while at the same time sentences restrict the kind of marking they can take.

So, in addition to the `subj` and `comps` features, categories will include the `spr` (SPECIFIER) feature to indicate exactly what kind of specifiers they take. Common nouns, as just mentioned, will take determiners as specifiers, and so the fully specified lexical entry for *book* will look like this:

```
book ---> cat:(head:noun,
               subj:[],
               comps:[],
               spr:[cat:head:det]).
```

We shall actually implement a new macro to simplify such entries. However, note that we will need to amend the type definition for `head` -- it should include `det` as a sub-type. However, in HPSG the situation here is slightly more complicated than just having `det` as another sub-type of `head` along with `noun` and `verb`. The assumption is that heads fall into two general types -- substantive and functional. Substantives are things like nouns and verbs, while functional categories are determiners (*this, every*), complementisers (*for, whether*), and suchlike. Furthermore, the functional types have an extra feature -- `spec` (not to be confused with `spr`) -- which is used to restrict the kinds of things they specify.

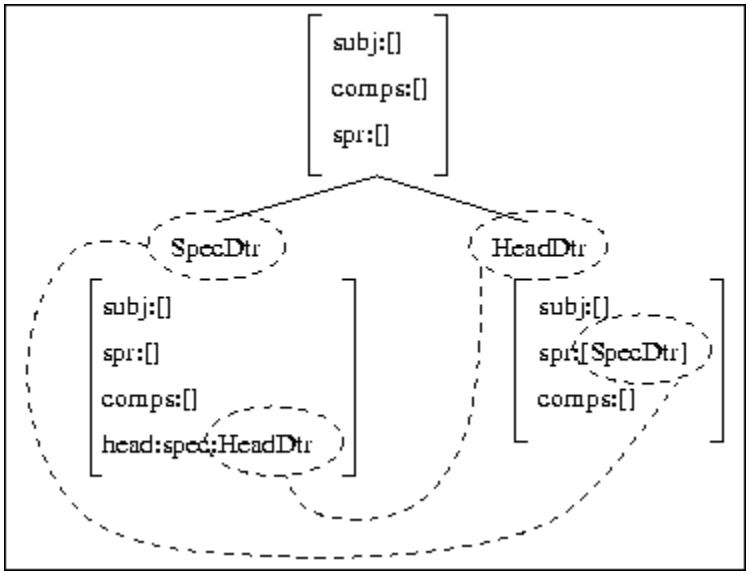
The appropriate part of the type hierarchy will therefore look like this:

```
head sub [subst,func].

subst sub [noun,verb].

func sub [mark,det]
```

```
  intro [spec:sign].

noun sub [].
verb sub [].
det sub [].
mark sub [].
```

The lexical entry for a determiner such as *the* will thus include the information that its `spec` value is an NBar category -- an NP which has still to find a specifier.

The main thing we need now is a version of the specifier-head schema to ensure that the appropriate categories match, and this can be represented by the following AVM:



Summary

In this part of the course we saw how HPSG principles such as the HFP can be implemented in ALE. This involved a look at how variables can be used to identify whole categories in rules. We then introduced the substantive and functional types and saw how the specifier-head schema might be implemented. The following exercises cover this material, and should be attempted now.

Exercises

These exercise fix some problems with the existing grammar and then alter the type hierarchy to include the new features `spr` and `spec` and the substantive and functional types, with their subtypes. Entries for *the* and *book* are described and the specifier-head schema is implemented.

1. The file `grammar3.pl` provides a basis for these exercises -- have a look to make sure you understand everything it contains. Verify that it produces two parses for *likes likes

Kim, one where *likes* is the subject of *likes Kim*, and one where *likes kim* is the complement of *likes*. Implement the Head Feature Principle as described in the text to rule out the second of these analyses.

2. The text suggests that we should identify the subject daughter in the subject-head rule and unify the whole of the head's SUBJECT feature with this:

```
cat> (SubjDtr),
cat> (HeadDtr, cat:(subj:[SubjDtr] ....
```

Make these changes, and verify that there's still a problem in that **likes likes Sandy* will still parse because the VP *likes Sandy* is not specifying what its subject should be. Add variables to the mother and head daughter to pass the `subj` feature up directly and verify that **likes likes Sandy* no longer parses.

3. The above solution is part of a general problem -- from now on, we should ensure that all the valency features are fully specified in the rules. Note that this applies to COMPLEMENTS in the subject-head rule also -- at the moment, the parser will fail if a string such as **Sandy snored snored* is attempted because the mother has no `comps` specification. Add a suitable value and verify that the latter string can now be handled -- with no parses, of course.

4. Add the feature `spr` (SPECIFIER) with the value `list` to the type definition of categories. Use `show_type` to check the result:

```
TYPE: cat
SUBTYPES: []
SUPERTYPES: [bot]
IMMEDIATE CONSTRAINT: none
MOST GENERAL SATISFIER:
     cat
     COMPS list
     HEAD head
     SPR list
     SUBJ list
```

5. Change the definition of the NP macro to ensure that NPs are fully satisfied for all the valence features and verify that lexical entries for nouns are correct. For example:

```
WORD: kim
ENTRY:
sign
CAT cat
    COMPS e_list
    HEAD noun
    SPR e_list
    SUBJ e_list
```

6. The text describes the HPSG account of substantive and functional types. Implement the type hierarchy with the new feature `spec` as discussed and verify the result using `show_type(func)`, which should produce:

```
TYPE: func
SUBTYPES: [mark,det]
SUPERTYPES: [head]
IMMEDIATE CONSTRAINT: none
MOST GENERAL SATISFIER:
     func
     SPEC sign
          CAT cat
              COMPS list
              HEAD head
              SPR list
              SUBJ list
```

7. In the HPSG version of X-Bar theory, X1 categories are those which have an unsatisfied specifier feature -- N1, for example, will be an NP which is looking for a determiner as a specifier. Implement a new macro `nbar` which uses the `anonymous' variable (underscore) to capture the fact that N1 categories have a non-empty but non-specific specifier:

```
nbar macro
  cat:(head:noun,
       subj:[],
       comps:[],
       spr:[_]).
```

Add a lexical entry for *book* which uses this macro and `fills in' the specifier information:

```
book ---> (@ nbar,
           cat:spr:[cat:head:det]).
```

As always, check the result of adding the new information before proceeding.

8. Add the following lexical entry for *the*:

```
the ---> cat:(head:spec:(@ nbar)
              spr:[],
              subj:[],
              comps:[]).
```

Note that, as discussed in the text, the value of the `spec` feature is an N-Bar.

Now implement the specifier-head schema using the AVM in the text as a guide and verify that you can parse *the book* and sentences such as *Kim gave Sandy the book.*

Solutions Solutions to all the above can be found in `grammar4.pl`.

6 The Head-Subject-Complement Schema

Introduction

In this fourth part of the course on HPSG and ALE we look at verbal features and the analysis of inverted sentences in English (*does Kim like books*, for instance) using the head-subject-complement schema.

Verbal Features

HPSG assumes that verbs have a number of features, including `vform`, `inv`, and `aux`. The use of these is discussed in the next section, and we then look at how the type system should be amended to take account of the new possibilities.

The VForm, Inv, and Aux Features

The AUXILIARY feature is simply used to distinguish auxiliary and main verbs, hence we shall state that *does* is `+aux` while *likes* is `-aux`. As we shall see, ALE does not allow exactly this formulation, but the practice is identical.

The VFORM feature is used to identify various forms of the verb -- the infinitive, the past participle, the passive participle, and so on. We shall focus here on just two, these being the base (`bse`), and finite (`fin`) forms. This will allow us to distinguish the main verb forms in the sentences below:

(4.1) Kim likes Sandy

(4.2) Kim does like Sandy

In 4.1, *likes* is the finite form, while in 4.2 *like* is the base form. The HPSG analysis of verbs assumes that auxiliary verbs such as *does* take base VPs as their complements, and this is exactly how we shall use the features `fin` and `bse` below.

The `inv` (INVERTED) feature is used in the analysis of English interrogatives such as *does Kim like Sandy* in which it is assumed that the auxiliary verb and the subject NP are `inverted' -- in order to make this work, we require a new schema, the head-subject-complement schema. Before looking at this, however, we need to sort out what the type system should be for the new features.

Verb Feature Types

First of all, we must add suitable features to the type `verb` -- until now there were none, of course. The ALE type hierarchy should therefore include the following:

```
verb sub []
  intro [vform:vform,
         aux:boolean,
         inv:boolean].
```

So we have three new features which take the values `vform` and `boolean`. As noted above, we shall only identify two values for `vform` -- `fin` and `bse` -- and so this new type will have the following definition:

```
vform sub [fin,bse].
```

The `fin` and `bse` types also need to be defined, and these will be basic elements with no features or subtypes.

The `boolean` value captures the `[+/-]` distinction often used in grammars -- as discussed above, we want to say that a verb like *does* is `+aux`, for instance, but ALE does not allow the use of plus and minus signs in grammars. We can get the required results, however, by saying that we have the feature `boolean` with values `yes` and `no` -- this will allow us to say that main verbs have the features `aux:no` and `inv:no`, for example. To get this to work, then, we simply need to state that `boolean` has the following type definition:

```
boolean sub [yes,no].
```

Again, we need to add the basic types `yes` and `no`, and note that we also need to define the types `vform` and `boolean` as sub-types of `bot`.

HPSG Auxiliary Verbs

The analysis of auxiliary verbs in HPSG has a long history, beginning with the account provided in early GPSG. The general idea is that auxiliaries are subcategorised for certain kinds of complement VP -- so something like *does* takes base form complements while *ought* takes infinitives, for example. This will allow us to capture the following data:

(4.3) Kim does like Sandy

(4.4) *Kim does to like Sandy

(4.5) Kim ought to like Sandy

(4.6) *Kim ought like Sandy

Another interesting aspect of the analysis of auxiliaries is the question of how their subjects are handled. HPSG assumes that in something like *Kim does like Sandy* both *does* and *like* have the same subject, which is implemented by structure sharing. In order to get this to work in ALE, we simply need to state that the subject of an auxiliary like *does* is the same as the subject of its VP complement, so the lexical entry will be:

```
does ---> (cat:(head:(verb,
                      vform:fin,
                      aux:yes),
                comps:[(@ vp,
                        cat:(subj:Subj,
                             head:vform:bse))],
                subj:Subj)).
```

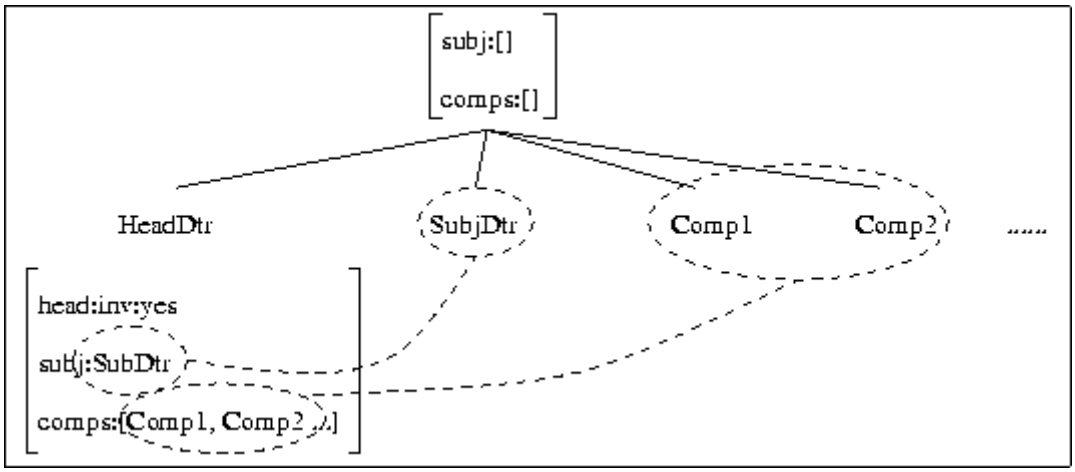
The required structure sharing is captured by the variable `Subj` which represents the value of both the subject of the auxiliary and the subject of the auxiliary's complement VP (which is specified in the latter example using a new macro).

Note that values for `vform` and `aux` should now be given for all verbs. Note also that no value is provided for `inv` on the auxiliary -- so, of course, the type definition will add the feature with the value `boolean`, which allows either `yes` or `no`. As this feature is used to determine whether or not a verb can be inverted, this will mean that such verbs will appear in both inverted and uninverted structures. All main verbs, however, must get the specification `inv:no` in order to block sentences such as **likes Kim Sandy* being accepted by the head-subject-complement schema, which we shall now investigate.

The Head-Subject-Complement Schema

As noted above, this schema is used in the analysis of English inverted sentences, and it can also be used to handle languages which have verb-subject-object order -- Welsh being a good example. A direct translation from Welsh of the sentence *Kim likes Sandy*, for instance, would be *likes Kim Sandy*, in which *Kim* is the subject and *Sandy* the object. HPSG assumes that a new schema, with at least three daughters, is required to handle such data.

Effectively, as the name suggests, the head-subject-complement schema is a combination of the subject-head schema and the head-complement schema, but in English the `inv` feature adds a new element. A simple AVM for this schema is:



The `inv` feature will be used to distinguish the auxiliary verbs which can be inverted from verbs which can't -- so we will allow *does Kim like Sandy* but rule out **likes Kim Sandy*.

Summary

The verbal features VFORM, AUXILIARY, and INVERTED in HPSG theory were discussed and we looked at the distinction between auxiliary and main verbs. We then saw how the head-subject-complement schema is designed to handle inverted sentences in English as well as SVO languages.

Exercises

Here we begin by implementing new `detp` and `s` macros. The latter is used in the lexical entry for *believes* in the analysis of embedded sentences. We then work through the various changes that are required to provide an analysis of inverted English sentences such *does Kim like Sandy*. These include the definition of the `vform`, `aux`, and `inv` features and their inclusion in the existing verb entries, the addition of base form verb entries and auxiliary verbs, and the implementation of the head-subject-complement schema.

1. Look at the grammar file `grammar4.pl` and make sure you understand everything it contains. Write a new `detp` macro which mirrors the NP version and use it in the lexical entry for single nouns, which should now have the specification `spr:[@ detp]`. Verify that you can still parse *Sandy likes the book*.

2. Implement an `s` (sentence) macro which we can use in the lexical entry for a verb such as *believes* as in *Kim believes Sandy likes the book*. Now add a suitable lexical entry for *believes* and check that the latter sentence parses.

3. Add the features `vform`, `aux`, and `inv` to the type hierarchy as discussed in the text and check the results using `show_type`, which should, for instance, give you:

```
| ?- show_type(vform).

TYPE: vform
SUBTYPES: [fin,bse]
SUPERTYPES: [bot]
IMMEDIATE CONSTRAINT: none
MOST GENERAL SATISFIER:
     vform

| ?- show_type(boolean).

TYPE: boolean
SUBTYPES: [yes,no]
SUPERTYPES: [bot]
IMMEDIATE CONSTRAINT: none
MOST GENERAL SATISFIER:
     boolean
```

Remember you need to add the new types to `bot` and also to define the new basic types used.

4. We want auxiliary verbs to take VPs as complements, so it will be useful to have a VP macro. A VP in HPSG can be defined as something which is verbal, which has a satisfied complements list, and which is still looking for a subject. Add a suitable macro which captures this description.

5. Add lexical entries for the auxiliaries *does* and *can* as discussed in the text and use the `lex` or `w` command to check the results. For example:

```
WORD: can
ENTRY:
sign
CAT cat
    COMPS ne_list
          HD sign
             CAT cat
                 COMPS e_list
                 HEAD verb
                      AUX boolean
                      INV boolean
                      VFORM bse
                 SPR list
                 SUBJ [0] ne_list
                      HD bot
                      TL e_list
          TL e_list
    HEAD verb
         AUX yes
         INV boolean
         VFORM fin
    SPR list
    SUBJ [0]
```

6. The auxiliaries *does* and *can* take base (`bse`) forms of main verbs as complements, as the lexical entry for *does* in the text suggests. The next step is therefore to add entries for the main verbs (*snore*, *like*, *give*, and *believe*) with appropriate values for the `vform`, `aux`, and `inv` features. Remember to add the features to the existing finite forms too. Verify that you can parse *Kim does like Sandy* and *Kim can give Sandy the book*.

7. Implement the head-subject-complement schema as suggested by the AVM in the text and check that you can parse *does Kim like Sandy* while blocking **likes Kim Sandy*. The general form of the schema will be:

```
head_subject_complement rule
Mother
===>
cat> HeadDtr .....
cat> SubjDtr .....
cats> (Comps, ne_list)
```

The HFP should apply here too, of course.

Solutions

The grammar in grammar5.pl includes solutions to all the above problems. Note that it also includes some Prolog code which is necessary for the next set of exercises and which produces an error message when the file is loaded due to some of the features not being defined. You can ignore the error for now.

7 The Adjunct-Head Schema

Introduction

This fifth part of the course on HPSG and ALE is mainly concerned with the formulation of the adjunct-head schema. It is now necessary, however, to revise the information content of signs, and so we begin with some discussion of the general form of HPSG feature structures.

Modifiers

HPSG assumes that another schema is required to handle the combination of modifiers (adjuncts) with heads -- this will account for a wide range of data, including the following:

(5.1) Red books (adjective-noun)

(5.2) Flies in the soup (noun-PP)

(5.3) Boldly go (adverb-VP)

The idea is that modifiers are subcategorised for certain types of object -- so adjectives state that they modify nouns, and so on. In order to implement this, the theory employs two more head features -- MODIFIER (`mod`) and PREDICATIVE (`prd`) -- the values of `mod` are the kinds of thing that the adjunct modifies, while `prd` is another boolean feature which determines whether or not an adjunct is appropriate in so-called predicative contexts (such as *the book is red*, where the adjunct follows the copula verb *be*).

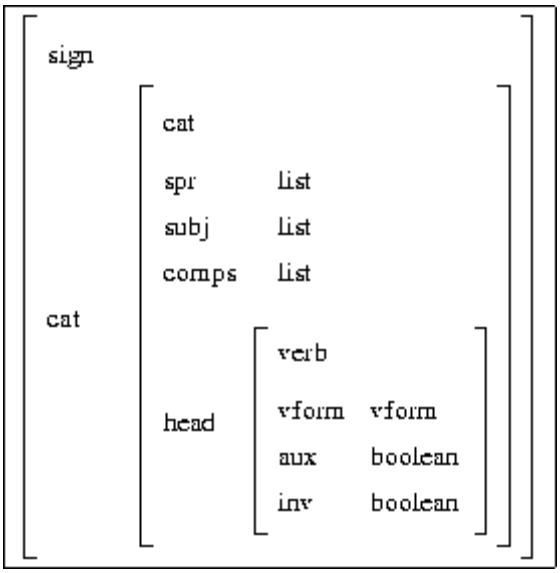
The new features are appropriate for the substantive categories -- nouns, verbs, and so on, and so ALE includes them in the type system as follows:

```
subst sub [noun,verb,adj]
         intro [prd:boolean,
                mod:mod_synsem].
```

Note that we have added the substantive subtype `adj`, and that the feature `prd` -- like `aux` and `inv` -- takes the boolean type as its value. The use of `mod_synsem` as the value of `mod` requires explanation, though, and as this means ultimately that we must revise the general architecture of signs, the next section looks at this question before we go on to discuss adjuncts.

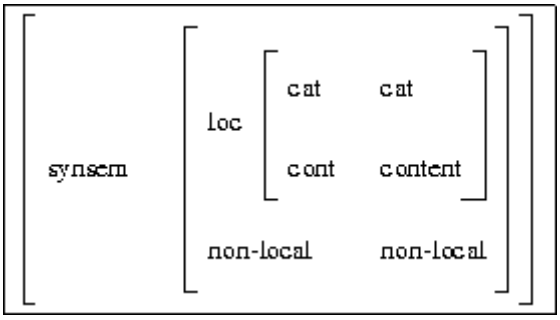
HPSG Signs

At the moment, all the information in the signs we have been using is contained in the `cat` structure, and as we shall shortly need to look beyond this, it will be useful to make some changes. Taking something of the type `verb` for an example, the current representation is as below:



HPSG actually assumes that all of this information is just part of the whole syntax and semantics of a sign, and furthermore, it is part of the `local' information associated with the object. For the moment, it is enough to know that the `non-local' part of signs -- which we shall introduce in the next part of the course -- deals with long-distance dependency information, while the local information is concerned with the category specification, and so on -- the feature structures we have been dealing with, in fact.

The general architecture of signs is thus a little more complicated than we have been assuming, as shown below:



All of the objects we have been using are in the CATEGORY feature's value -- we have just added some more structure around it. The CONTENT feature will hold the semantic information, the LOCAL and NON-LOCAL features will deal with the basic category

information and the long-distance dependency material, and everything is contained in the SYNSEM feature.

Adjuncts in ALE

Based on the above, we should now alter the HPSG grammar we are developing to make it compatible with the theory and to take account of adjuncts. The following sections therefore look at the type hierarchy, the lexical entries, and finally the adjunct-head schema.

Adjunct Type Definitions

Looking again at the question of adjuncts like *red*, it was noted above that the MODIFIER feature will be used to restrict the kinds of thing that a sign can modify. Most verb forms, for instance, cannot be used as modifiers (**the like book* cf. *the red book*, and so forth), while adjectives modify nouns, adverbs verbs, and suchlike, and we therefore want to capture the fact that the values of `mod` are anything -- including nothing. HPSG therefore invents a type -- `mod_synsem` -- which effectively covers `anything and nothing' in this sense.

Following the above reasoning, in the HPSG book the type system is described as follows:

```
bot [ ... mod_synsem ...].

mod_synsem sub [synsem,none].

synsem sub []
  intro [loc:loc,
         nonloc:nonloc].
```

The `synsem` type will introduces the local and non-local features, as suggested above. However, there is an added complication here in the ALE implementation. You will remember that the categories in the schemas must appear in the order in which they are entered -- so in our subject-head schema, for instance, the subject must precede the head. The trouble is that in English, and many other languages, adjuncts appear both before and after the objects they modify -- compare *the red book*, with the adjective before the noun, with *the man in the corner*, in which the prepositional phrase follows the noun. We will therefore have to use two rules to handle the adjunct-head schema -- one for adjunct-head and one for head-adjunct.

This leaves the problem of how to restrict modifiers to one or other schema -- we don't want to allow **the book red* -- and we shall assume the solution to this proposed by Suresh Manandhar. His answer is to have two subtypes of `synsem` -- `pre_mod_synsem` and `post_mod_synsem` -- which are subsequently used in the schemas and lexical entries to ensure the necessary restrictions.

The type system which we shall implement will therefore be identical to the theory except that `synsem` will have the two new subtypes, both themselves defined as basic types:

```
synsem sub [pre_mod_synsem,post_mod_synsem].
intro [loc:loc,
       nonloc:nonloc].
```

If we assume for present purposes that `nonloc` and `cont` are defined as basic types, and change the definition of `sign` to ensure that it introduces the feature SYNSEM instead of CATEGORY, then this covers everything necessary to get the higher-level types and features working. We do also need to alter the definition of the substantive type, of course, as described above.

Notice that we will also have to change all the lexical entries we have been using, and it makes sense to alter the macros too. As an example, here is the existing form of the NP macro:

```
np macro
  cat:(head:noun,
       comps:[],
       subj:[],
       spr:[]).
```

This will become:

```
np macro
  loc:cat:(head:noun,
           comps:[],
           subj:[],
           spr:[]).
```

The assumption in the macros, then, is that they describe the LOCAL information. As we shall see shortly, this will involve an important alteration to the rules, but for the moment it should be noted that the lexical entries for NPs must include the SYNSEM feature in order to specify the full path properly. So, the existing entry for *Kim*:

```
kim ---> @ np.
```

Becomes:

```
kim ---> (synsem:(@ np,
                  loc:cat:head:mod:none)).
```

Note that we have not included a `mod:none` element in the definition of an NP -- it is arguable that we do want NPs to be used as modifiers on occasion (*wine gums*, *bottle opener*), and hence the feature is assumed here to be specified directly on each nominal entry.

Lexical Entries for Adjuncts

The lexical entries are fairly straightforward. We simple want to identify the kinds of things an adjunct can modify in much the same way as we used the `spec` feature to determine the kinds of things that a specifier specifies. An adjective like *red*, then, will look like this:

```
red ---> (synsem:loc:cat:(head:(adj,
                                mod:(pre_mod_synsem,
                                     @ nbar)),
                          spr:[],
                          subj:[],
                          comps:[])).
```

The main thing to notice is that we have stated, not only that the adjective modifies NBars, but also that it is of the pre-modifier type. This will ensure that we can parse *red book* but not **book red*.

It will be necessary, of course, to add the specification `mod:none` to every lexical entry that we do not want to consider as a modifier -- everything else in the lexicon, in fact.

Adjunct-Head Rules in ALE

We are now almost ready to implement the schema(s). However, we are going to have a problem with matching the new structure of signs with the current values of the valency features. To make this clear, look at the lexical entry for *snored*, which should now have the following general form:

```
snored --->
  synsem:loc:cat:(head:(verb,
                        vform:fin,
                        aux:no,
                        inv:no),
                  subj:[@ np],
                  comps:[])).
```

The subject-head schema, for example, will try to match the `subj` specification against the category of the lexical entry for *Kim* in order to parse *Kim snored*, and it will fail. This is because the subject specification is given as an NP, which we have now defined as shown in the macro above to be an object of type `synsem`. However, the lexical entry for nouns -- as with all words, in fact -- ensures that they are of type `sign`, and so the current rule will not be able to parse anything due to the type mismatches.

This is actually part of the general theory of HPSG -- objects are subcategorised for syntax and semantics information, not for whole signs. To get this to work, it would of course be possible in most circumstances to include the necessary path specifications in the rules. However, there would still be a problem in matching lists of objects (the

value of `comps`, for example), and so it is normally assumed that this should be done by a DEFINITE CONSTRAINT which effectively mediates between signs and synsem values. The grammar file grammar5.pl, which contains answers to the last set of exercises, includes the necessary piece of Prolog code:

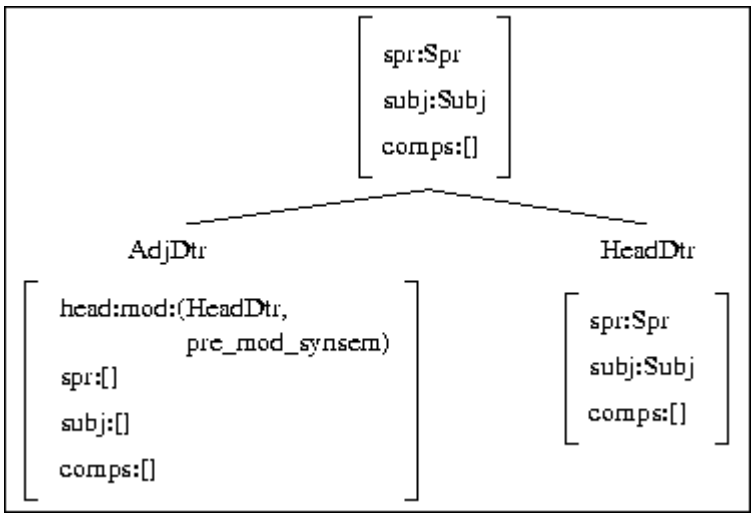
```
sign_to_synsem(synsem:Synsem,Synsem) if true.

list_sign_to_synsem([],[]) if true.

list_sign_to_synsem([H|T],[S|R]) if
     sign_to_synsem(H,S),
     list_sign_to_synsem(T,R).
```

It is not necessary to understand this -- although it's good practice to make an attempt at it. The basic `sign_to_synsem` predicate is very simple, and the other just applies this to everything in a list. The main thing is that we can now use these predicates as goals in the ALE rules in order to ensure that the necessary matches are effected. Thus, for example, we will need to add goals to all the rules in the way that they are included in the new head-subject-complement schema below:

```
head_subject_complement rule
(Mother, synsem:loc:cat:(spr:[],
                         subj:[],
                         comps:[]))
===>
cat> (HeadDtr, synsem:loc:cat:(head:inv:yes,
                               subj:[SubjSynsem],
                               comps:CompSynsems)),
goal> (list_sign_to_synsem(CompDtrs,CompSynsems)),
cat> (SubjDtr),
goal> (sign_to_synsem(SubjDtr,SubjSynsem)),
cats> (CompDtrs,ne_list),
goal> (head_feature_principle(Mother,HeadDtr)).
```

This is just as before, except that the value of `subj` on the head, for instance, is converted using `sign_to_synsem` and then matched with the subject daughter. One important point is that the order in which the grammar compiler encounters the predicates and variables is important -- the variables should not appear before they are used in a goal, so moving the `sign_to_synsem` statement above the category specification for the subject daughter will not work. On the other hand, it occasionally causes problems if goals which are more complicated than simple path specifications follow all appearances of the variables too, and so the safest assumption is often that such goals should appear `between' variables, as with the `list_sign_to_synsem` in the latter rule. In general, one of the solutions to try if problems are encountered with goals is to re-order the statements.

Finally, then, we can look at the adjunct-head and head-adjunct schemas. The following is an AVM for the adjunct-head schema:



The other schema will be identical, of course, except for the order of the categories and the use of `post_mod_synsem` in the value of `mod`. Some points to note are that it's assumed that the head daughter has `consumed' all its complements before it's modified, and that the other valency features are just passed on. Thus, for instance, we are assuming that sentences such as *Kim definitely likes Sandy* and *Kim likes Sandy definitely* are ok, while **Kim likes definitely Sandy* is not -- in the latter case the adverb would need to be modifying the verb before it had found its complement.

Summary

This part of the course looked at the analysis of modifiers in current HPSG. We saw that this involves a complication in the type system that we have been using, and we also saw that we will actually need two ALE rules to cover the single HPSG schema. As a result modifiers often have to be restricted to one or other schema, and we saw how one solution to this is via the type hierarchy. Finally, the form of the head-adjunct schema itself was discussed.

Exercises

These exercises introduce all the amendments to the type hierarchy suggested in the text. This is quite a difficult thing to get right, and you may find that a look ahead to the grammar in `grammar6.pl` is useful on occasion. Once the type system is working, though, you should still try to get the schemas working as described in the text without looking at the answer!

1. The file `grammar5.pl` should be used as a basis for these exercises -- make sure you understand everything it contains. Note that it includes the `sign_to_synsem` definitions mentioned in the text, and that these will produce an error message until all the features mentioned in the text are defined. The error can be ignored for the moment.

Alter the type hierarchy as described in the text to include the new types `mod_synsem`, `pre_mod_synsem`, `post_mod_synsem`, `synsem`, `loc`, `nonloc`, and `cont`. Load the grammar (you will probably get complaints about some of the lexical entries -- ignore them for now) and check that `show_type(sign)` produces the following result:

```
TYPE: sign
SUBTYPES: []
SUPERTYPES: [bot]
IMMEDIATE CONSTRAINT: none
MOST GENERAL SATISFIER:
     sign
     SYNSEM synsem
            LOC loc
                CAT cat
                    COMPS list
                    HEAD head
                    SPR list
                    SUBJ list
                CONT cont
            NONLOC nonloc
```

2. Alter the definition of the `subst` type as described in the handout, adding the `adj` type and the features `prd` and `mod` with appropriate values. Check the results using `show_type(subst)`:

```
TYPE: subst
SUBTYPES: [noun,verb,adj]
SUPERTYPES: [head]
IMMEDIATE CONSTRAINT: none
MOST GENERAL SATISFIER:
     subst
     MOD mod_synsem
     PRD boolean
```

You will also have to alter the definition of the `spec` feature in the type hierarchy -- it should now take the `synsem` type as value. Note in general that, while working with the type system, it is often necessary to exit the system completely and reload in order to keep things straight. You may experience some inexplicable problems if you don't do this.

3. Change the lexical entries and macro definitions as suggested in the text to take account of the new features and type hierarchy. Verify the entry for *snored* using `lex` or `w`:

```
WORD: snored
ENTRY:
sign
SYNSEM synsem
       LOC loc
           CAT cat
               COMPS e_list
               HEAD verb
```

```
                    AUX no
                    INV no
                    MOD none
                    PRD boolean
                    VFORM fin
               SPR list
               SUBJ ne_list
                    HD synsem
                       LOC loc
                           CAT cat
                               COMPS e_list
                               HEAD noun
                                    MOD mod_synsem
                                    PRD boolean
                               SPR e_list
                               SUBJ e_list
                           CONT cont
                       NONLOC nonloc
                    TL e_list
           CONT cont
       NONLOC nonloc
```

4. Add the lexical entry for *red* as described in the handout and check the result:

```
WORD: red
ENTRY:
sign
SYNSEM synsem
       LOC loc
           CAT cat
               COMPS e_list
               HEAD adj
                    MOD pre_mod_synsem
                        LOC loc
                            CAT cat
                                COMPS e_list
                                HEAD noun
                                     MOD mod_synsem
                                     PRD boolean
                                SPR ne_list
                                    HD bot
                                    TL e_list
                                SUBJ e_list
                            CONT cont
                        NONLOC nonloc
                    PRD boolean
               SPR e_list
               SUBJ e_list
           CONT cont
       NONLOC nonloc
```

5. Alter all the existing schemas to take account of the new structure of signs, adding the goals `sign_to_synsem` and `list_sign_to_synsem` as described in the handout. Note that the definition of the required definite constraint is already in the file `grammar5.pl`.

Note further that you'll need to change the definition of the HFP also as it will require the new path specifications too:

```
head_feature_principle(synsem:loc:cat:head:Head,
                       synsem:loc:cat:head:Head) if true.
```

It's a very good idea to do all of this in small stages -- change the head-subject-complement schema, for instance, and if the lexical entries for *does*, *Kim*, and *snore* are all correct, then *does Kim snore* should work. And so on. And remember that it's often necessary to exit the system and recompile everything when basic changes such as those necessary are being made.

6. Add the adjunct-head and head-adjunct schemas as described in handout 5 and check that you can parse *red book* and *Kim likes the red book* but not *book red*.

7. Assuming that adverbs are of type `adj` but with different `mod` values, add a lexical entry for *definitely* and check that you can parse *Kim definitely likes Sandy* but not **the definitely book*. Note that *Kim likes Sandy definitely* should be ok too, while **Kim likes definitely Sandy* should not parse.

Solutions

Solutions to all the above questions can be found in `grammar6.pl`. The file also contains some Prolog code which will be used in the next exercise and which currently causes error messages when the file is loaded.

8 The Filler-head Schema and the Marking Principle

Introduction

This sixth part of the course on HPSG and ALE introduces the HPSG analysis of long-distance dependencies, looking here at topicalised sentences such as *the book Sandy believes Kim likes*. The account presented here is a combination of GPSG and early HPSG -- the next part of the course will look at the current HPSG treatment. We also look at an implementation of the HPSG Marking Principle and (briefly) at the Specifier Principle.

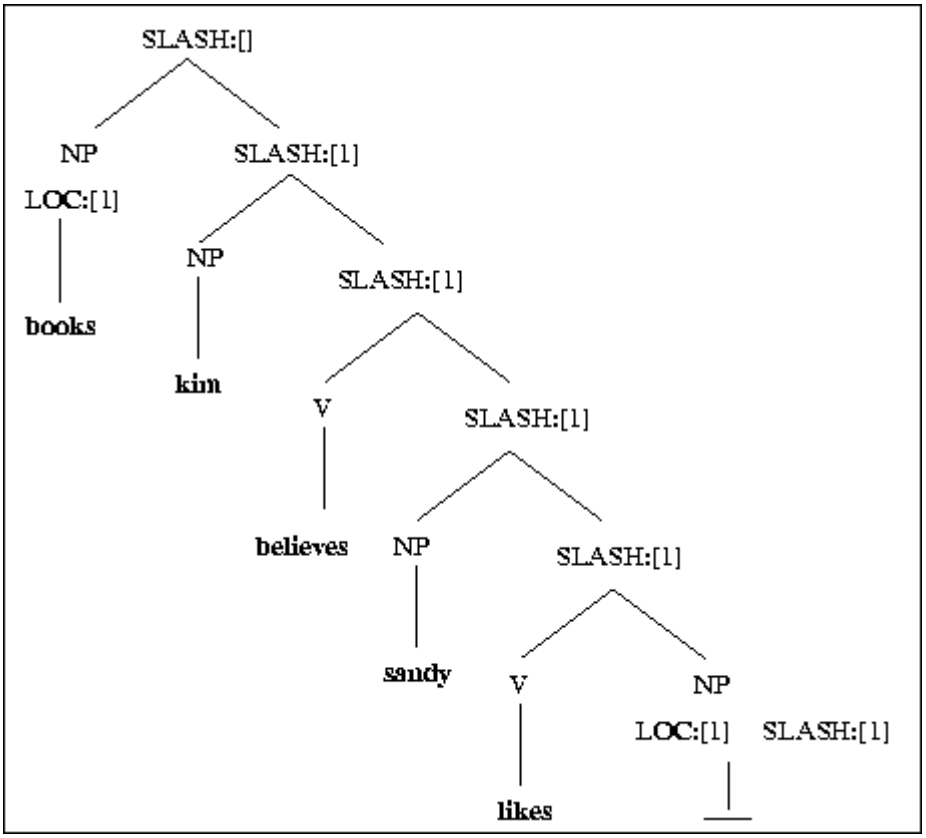
The Slash Feature

One of the most important aspects of GPSG initially was the suggestion that long-distance dependencies could be handled without transformations, the argument being that structure-sharing could account for most of the facts. In order to implement this, the feature SLASH was introduced, its values being, for example, the `missing' object in the sentence *beans I like*. SLASH values are passed around structures until they can be `bound' by suitable objects -- in *beans I like*, the missing complement of *like* in the VP is represented by a SLASH value which is then matched against the topicalised NP *beans*, as we shall see.

In the ALE grammar that we have been developing, there is a `placeholder' feature `nonlocal` which we shall now use to handle the long-distance dependencies. For the moment, we shall assume simply that the value of this feature is just the feature SLASH itself. We shall further assume that SLASH takes list of synsem objects as its values, and so we shall alter the type hierarchy to include the following:

```
nonloc sub []
   intro [slash:list].
```

In the HPSG book, a tree similar to the following is used to illustrate the use of SLASH (p.160) -- note that the path specification required in the ALE grammar to get to the SLASH value will just be `synsem:nonloc:slash`:

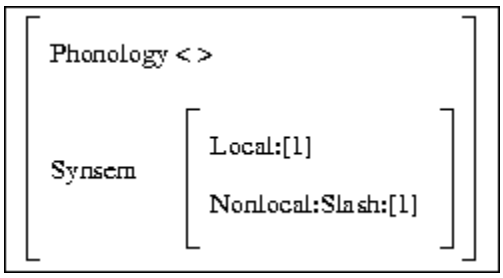


There are three immediately important aspects to this tree, dealing with the top, the middle, and the bottom. At the bottom of the tree, something must be done to handle the missing object itself -- in the above example, the object of *likes* is effectively an empty NP. Secondly, the information about the missing object must be passed up the tree, and finally, at the top, we shall need a new schema to combine a `filler' (*Kim* in this example) with the head -- which in this case is a sentence with a missing object NP.

We shall deal with these phenomena here by introducing `trace' (the empty NP), by adding the NON-LOCAL FEATURE PRINCIPLE, and by implementing the filler-head schema.

Empty NPs

There has been a long debate about the best way to represent `missing' elements in syntactic structure, and we now need something to represent `an NP which is missing an NP' -- in current terms something represented by the feature structure `NP[Slash NP]`. In the early chapters of the HPSG book, it is assumed that there is an empty NP (`trace') in the lexicon which accounts for this. A slightly simplified version of trace can be represented as follows:



This is something which has no phonology (of course), and which simply says that its LOCAL and SLASH features are identical. So, when combined with a verb such as *likes*, which will instantiate its required object NP features, the result will be a SLASH value which represents the object of the verb.

Now, while it is possible to imagine an entry in an ALE lexicon which has no phonology, it is not possible to include something like the lexical entries we have been using which has no orthography. In order to include trace in our lexicon, therefore, we shall assume that it is represented as `e' (for `empty'), and hence we will have:

```
e ---> synsem ....
```

In addition to the features in the AVM representation above, all that is required are values (all empty lists, of course) for the valency features, and also a `mod:none` specification. With everything else in place, we should now be able to parse sentences such as *Kim likes e*.

It should be noted that ALE actually provides a method of representing empty categories without having any orthographic representation. However, the latest versions of HPSG assume that there are no such things as empty categories, as we shall see in the next part of the course, so the simplest approach in the present circumstances is to use the special lexical entry *e* as a stop-gap.

The Non-Local Feature Principle (NLFP)

As noted above, we now need some method of passing SLASH values around the trees. It might seem at first that SLASH should be a head feature -- the head feature principle will automatically pass values from head daughter to mother, of course. However, closer inspection of the tree above shows that the SLASH value does not always come from the head daughter -- in fact, right at foot of the tree, the value must originate as a complement of *likes*.

Also, it is now generally accepted that more than one daughter can have a SLASH value, both of which must be passed up to the mother category. It is for these reasons that HPSG assumes that the value of SLASH is actually a *set* of objects. The initial version of the NLFP in Pollard & Sag therefore states that:

NonLocal Feature Principle (p.162)

The value of each nonlocal feature on a phrasal sign is the union of the values on the daughters.

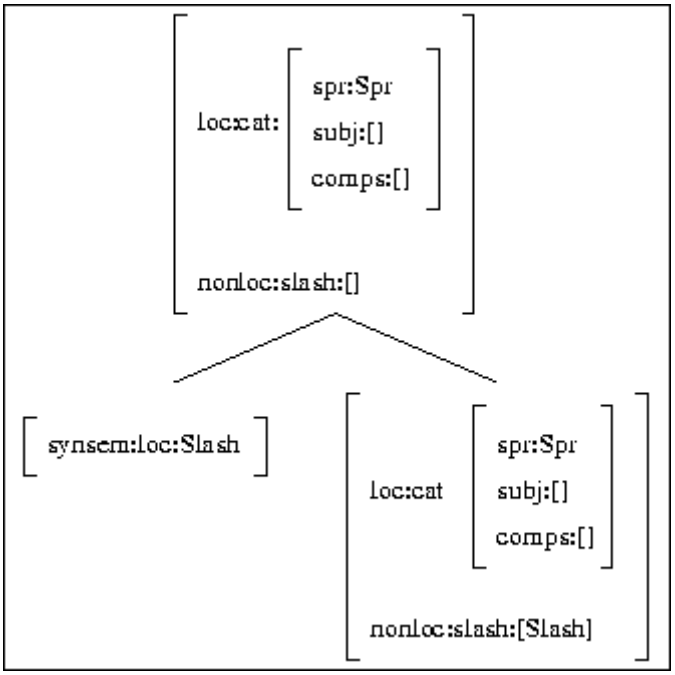
To simplify matters a little here, we shall assume that we are dealing with lists rather than sets -- the next part of the course introduces sets to the grammar. So, what we need is something which will gather all the values for SLASH on daughters and pass the result up to the mother category -- the grammar in grammar6.pl includes the necessary code. We can then add another goal to the schemas which enforces the principle -- however, this is a little more awkward than the situation with, for instance, the head feature principle, and so grammar6.pl also includes all the necessary goals statements. For instance, the subject-head rule is now:

```
subject_head rule
(Mother, synsem:loc:cat:(spr:Spr,
                         subj:[],
                         comps:[]))
===>
cat> (SubjDtr),
goal> (sign_to_synsem(SubjDtr,SubjSynsem)),
cat> (HeadDtr, synsem:loc:cat:(spr:Spr,
                               subj:[SubjSynsem],
                               comps:[])),
goal> (head_feature_principle(Mother,HeadDtr)),
goal> (nonlocal_feature_principle(HeadDtr,[SubjDtr],Mother)).
```

It is not necessary to understand exactly how the principle is implemented -- however, it is important to realise that any SLASH values on the subject and head daughters will be combined and passed on to the mother.

The Filler-Head Schema

All we need now is a schema which will combine a topicalised NP like *Kim* with a sentence such as *Sandy likes e*. This is not particularly difficult:



One point to note is that there is an explicit `slash:[]` specification on the mother -- this is at odds with the suggestion made above that more than one gap is possible. In the following part of the course we shall implement the NLFP more faithfully; for the moment we shall just allow a single gap. Assuming that all lexical entries apart from trace have the specification `slash:[]`, everything should be in place to get sentences such as *the book Kim believes Sandy likes e* to parse.

The Marking Principle

One area of the theory which we have been ignoring is the treatment of `marking'. The grammar in `grammar6.pl`, which represents the answers to all the exercises set so far, will parse strings such as:

(6.1) *The Kim snored

(6.2) *The book Kim likes Sandy

In 6.1 the determiner is accepted as a specifier for the sentence *Kim snored*, and in 6.2 the sentence *Kim likes Sandy* is taking an NP as a specifier. The reason for this is that we have said nothing so far about what the specifier requirement is in sentences, and we shall now investigate the HPSG analysis of this.

The theory of HPSG assumes that, in addition to parts-of-speech such as noun, verb, and adjective, there is a class of `markers'. These are typically the complementisers, such as *that, for, whether*, and so on, and we have already included the type `mark` as a subtype of `func`:

```
func sub [mark,det]
  intro [spec:synsem].
```

Complementisers, and things like conjunctions, are entered in the lexicon with this type. Like determiners, that they have the feature `spec`, which means they will be handled by the specifier-head rule, and also that they should identify the kinds of things they specify in the same way as determiners select nouns. At the same time, verbs (for instance) will identify the kinds of things they take as specifiers, and together these statements will rule out sentences such as 6.1 and 6.2 above.

The theory therefore implements another feature, MARKING, which determines how a particular category is `marked' -- a sentence with a complementiser, such as *that Kim likes Sandy*, will have the specification `marking:that`, and this will allow verbs which are sensitive to particular complementisers to select complements of the right form. We shall therefore be able to deal with contrasts such as:

(6.3) Kim believes that Sandy is dumb

(6.4) Kim wondered whether Sandy is dumb

(6.5) *Kim believes whether Sandy is dumb

(6.6) *Kim wondered that Sandy is dumb

The default value of the feature will be that categories are unmarked unless the specifier-head rule has operated to add a particular marking value. To get this to work in ALE, we need to add the marking feature, and a new type, to the type declarations:

```
bot sub [ ... marking ... ].

cat sub []
   intro [head:head,
          subj:list,
          comps:list,
          spr:list,
          marking:marking].

marking sub [marked, unmarked].

marked sub [comp,conj].

comp sub [that,for].

that sub [].
for  sub [].
conj sub [].
unmarked sub [].
```

Note the inclusion of `conj` for completeness. The lexical entry for `that` will look like this:

```
that --->
  (synsem:(loc:(cat:(head:(mark,
                           spec:(@ s,
                                 loc:cat:(head:vform:(fin;bse),
                                          marking:unmarked))),
                     spr:[],
                     subj:[],
                     comps:[],
                     marking:that)),
           nonloc:slash:[])).
```

Points to note are the specification that the sentence which this complementiser specifies should be unmarked (we don't want to allow **that that Kim snored*), the fact that `mark` in the head value distinguishes *that* from determiners, and the value `that` on the marking feature. Notice also that this means that we will have to include the specification `marking:unmarked` on everything in the lexicon which is not a marker, and remember also that verbs should include a value for the SPECIFIER feature which identifies a particular complementiser type. The entry for *likes*, for instance, should now be:

```
likes ---> (synsem:(loc:cat:(head:(verb,
                                   mod:none,
                                   vform:fin,
                                   aux:no,
                                   inv:no),
                             subj:[@ np],
                             comps:[@ np],
                             spr:[@ comp],
                             marking:unmarked),
                    nonloc:slash:[])).
```

Note that this includes a reference to the macro `comp`, which will be the description of a complementiser. This will be something which is a marker and whose marking is `comp`:

```
comp macro
  loc:cat:(head:mark,
           marking:comp).
```

Note that the entry for *that* included the specification `(fin;bse)` -- the semi-colon denotes disjunction, so this will actually produce two lexical entries for *that*.

The specifier-head rule must be altered to take account of the marking value, and we shall implement this as yet another goal:

```
specifier_head rule
Mother
===>
cat> SpecDtr,
cat> HeadDtr,
goal> (head_feature_principle(Mother,HeadDtr)),
goal> (nonlocal_feature_principle(HeadDtr,[SpecDtr],Mother)),
```

```
goal> (marking_principle(Mother,SpecDtr)).
```

We now need a version of the marking principle, and this is just the following statement:

```
marking_principle(synsem:loc:cat:marking:Mark,
                  synsem:loc:cat:marking:Mark)
   if true.
```

Together, these statements will ensure that the required marking feature is added to categories by the specifier-head rule -- notice, however, that in addition to this we must add the principle to all the other rules to ensure that the value for marking gets passed to the mother. In every other case, the value comes from the head daughter, and so we need the following goal on the rest of the rules:

```
goal> (marking_principle(Mother,HeadDtr)).
```

With these changes in place, the grammar should allow *Kim believes that Sandy likes the book* and rule out **the Kim snored*, and so on.

The Spec Principle

When we introduced the specifier-head schema, we explicitly stated in the rule that the value of `spec` on the specifier daughter should match the head daughter:

```
cat> (SpecDtr,cat:(subj:[],
                   spr:[],
                   comps:[],
                   head:spec:HeadDtr)),
cat> (HeadDtr, ....
```

HPSG actually assumes that the `Spec Principle' should handle this:

Spec Principle (p.51)

If a nonhead daughter in a headed structure bears a SPEC value, it is token-identical to the SYNSEM value of the head daughter.

The grammar in grammar6.pl contains a suitable version of the principle -- all that is necessary is that a goal should be added to the specifier-head schema as follows:

```
goal> (spec_principle(SpecDtr,HeadDtr))
```

The `head:spec:HeadDtr` specification should now be removed from the specifier-head rule.

Arguments in Macros

It is often useful to have macros which include variables -- for instance, rather than specify the full path to a marking value, it would be useful on occasion to be able to say `marking:that`, or whatever, without having to include `synsem:loc:cat:marking:X`. ALE allows this quite simply:

```
marking(Marking) macro
   loc:cat:marking:Marking
```

As an example of the use of this kind of macro, we want to restrict adverbs such as *definitely* so they only modify unmarked VPs, and so the lexical entry would be:

```
definitely --->
   synsem:(loc:cat:(head:(adj,
                          prd:no,
                          mod:(@ vp,
                               @ marking(unmarked))),
                    spr:[],
                    subj:[],
                    comps:[],
                    marking:unmarked),
           @ noslash).
```

The argument of the macro just appears inside round brackets, as shown.

Summary

Here we looked at a simple account of non-local dependencies using the feature SLASH in much the same manner as GPSG. We also introduced the MARKING feature which is used to characterise constructions precisely in a variety of situations. Finally, we looked briefly at the Spec Principle.

Exercises

These exercises introduce the HPSG analysis of long-distance dependencies using SLASH. The Marking Principle and the Spec Principle are then implemented.

1. The file `grammar6.pl` contains answers to the last set of exercises as well as some Prolog code which implements the NLFP and the Spec Principle as discussed in the text. Have a look to check that the contents makes sense -- you don't have to understand exactly what the Prolog is doing, but you should understand what the principles are. You should get a complaint about `slash` not being declared if the grammar is loaded as it stands -- this first exercise fixes that.

Add the `slash` feature to `nonloc` and construct a lexical entry for trace as suggested in the handout. Now ensure that you can parse *likes e*. Note that the value of SLASH on the result is just `list`.

2. All the necessary non-local feature principle statements and definitions are already in `grammar6.pl`. The goals are currently commented out, so remove the percent signs from the rules. Note that you will need to add `AdjDtr` variables to the relevant daughters in the adjunct rules to identify the categories in question.

We now need to say that all lexical entries -- apart from the empty category -- have the specification `nonloc:slash:[]`. As we shall alter this to handle sets in the next part of the course, the best idea is to use another macro here:

```
noslash macro
   nonloc:slash:[].
```

Implement this and add the specification `@ noslash` to all the lexical entries (apart from trace, of course). Check that you can parse *likes e* again -- this time with a value for slash as below:

```
NONLOC nonloc
       SLASH ne_list
             HD loc
                CAT cat
                    COMPS e_list
                    HEAD noun
                         MOD none
                         PRD boolean
                    SPR e_list
                    SUBJ e_list
                CONT cont
             TL e_list
```

3. Now add the filler-head schema as described in the text, remembering that the NLFP should not be applied to this schema yet. You should now be able to parse *Kim Sandy likes e* and *books Sandy believes Kim likes e*.

4. Add the feature `marking` and the type `marking` to the type declaration as described in the handout. Check that everything is ok using `show_type(marking)` and so on:

```
| ?- show_type(marking).

TYPE: marking
SUBTYPES: [marked,unmarked]
SUPERTYPES: [bot]
IMMEDIATE CONSTRAINT: none
MOST GENERAL SATISFIER:
     marking
```

5. Add a lexical entry for *that* as described in the handout and check the result -- you should get two structures, one of which is:

```
sign
SYNSEM synsem
       LOC loc
           CAT cat
               COMPS e_list
               HEAD mark
                    SPEC synsem
                         LOC loc
                             CAT cat
                                 COMPS e_list
                                 HEAD verb
                                      AUX boolean
                                      INV boolean
                                      MOD mod_synsem
                                      PRD boolean
                                      VFORM bse
                                 MARKING unmarked
                                 SPR list
                                 SUBJ e_list
                             CONT cont
                        NONLOC nonloc
                               SLASH list
               MARKING that
               SPR e_list
               SUBJ e_list
           CONT cont
       NONLOC nonloc
              SLASH e_list
```

You should also now be able to parse *that Kim likes Sandy*, but there will be multiple results -- we need to specify for verbs what the marking is on its specifier.

6. Add a `comp` macro as described in the handout and include it in all the verb entries along with the `marking:unmarked` specification as shown in the lexical entry for *likes*. Check that your entry is valid using `lex` again -- `MARKING comp` should appear in the `SPR` structure, and `MARKING unmarked` in the local category information.

7. Now we need to add the Spec Principle, so add the goal to the specifier-head rule as discussed in the text. Now implement the marking principle as described in the handout and add the goal to all the rules -- remember that the specifier-head schema is different from the others. Check that you can parse *Kim believes that Sandy snored*, and so on, with the right result. You should also get cases where a sentence has been topicalised -- so *that Kim snored Sandy believes e* should be ok too.

9 Inherited Slash, To-Bind Slash, and Lexical Rules

Introduction

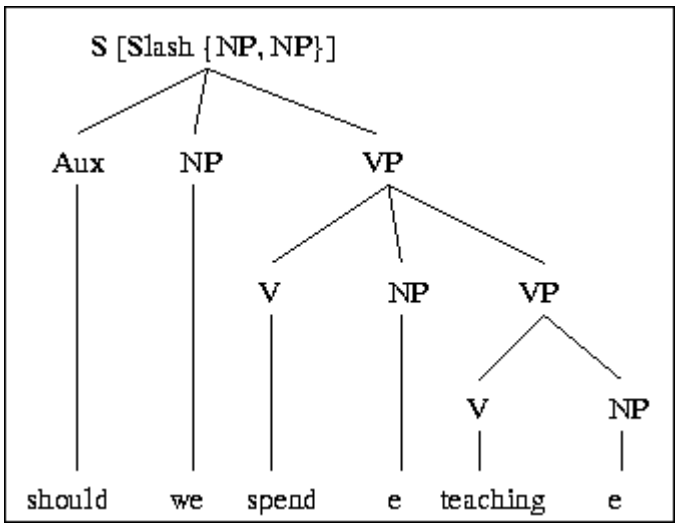
This is the seventh part of the course on HPSG and ALE. Here we look at the HPSG analysis of long-distance dependencies as laid out in Chapter 9 of Pollard & Sag. This involves two new features, INHERITED and TOBIND -- both of which take SLASH as values -- and also involves a discussion of the use of lexical rules in the current theory.

Set-Valued Slash

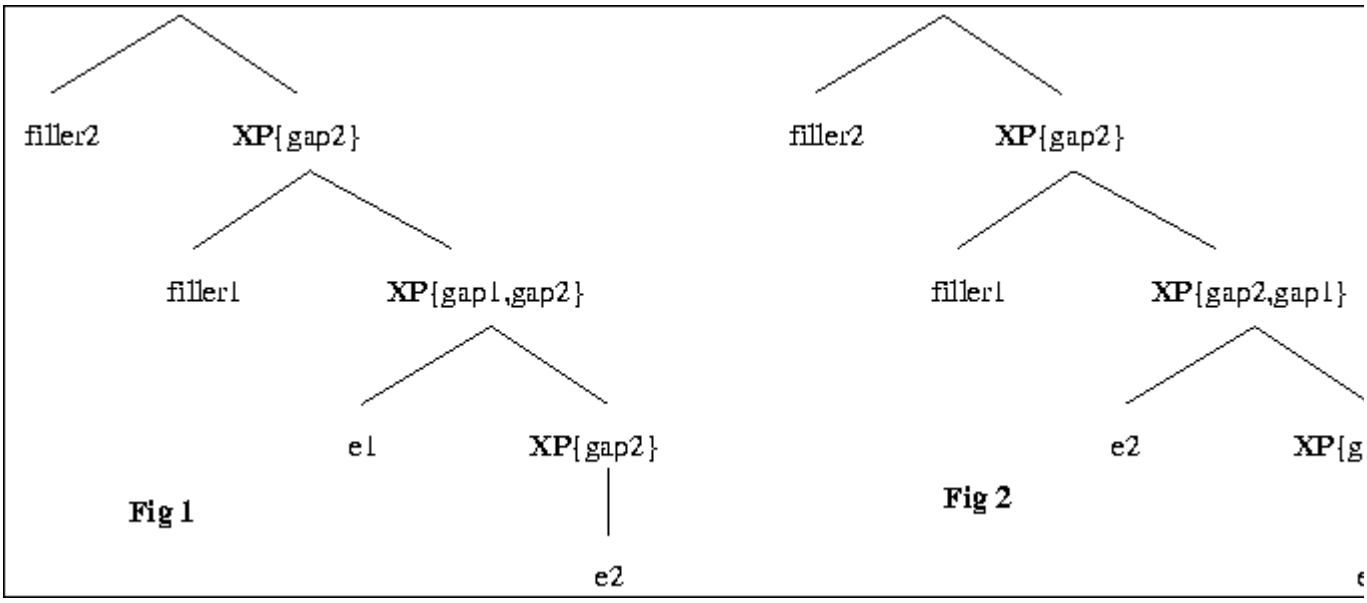
The simple implementation of Slash that we have been using so far does not generalise easily over all the schemas -- in the grammar file `grammar6.pl`, the NonLocal Feature Principle applies to all the schemas except the filler-head rule, and the argument is that it should apply universally. At the moment we stipulate that the mother in the filler-head schema has the specification `slash:[]`, which of course rules out multiple non-local dependencies of this kind. It has been argued that this is too restrictive, and that in fact many languages require SLASH to represent more than one category. It may even be that sentences such as the following would require multiple-valued SLASH in English:

(7.1) ? Someone that stupid, how much time should we waste teaching?

Here we seem to have `gaps' after *waste* and *teaching*, and the analysis of, for instance, the structure containing the inverted auxiliary verb *should* and its complements would require the representation of two missing NPs. On the assumption that the verb *spend* takes an NP and a non-finite VP as complements (*spend time watching the news*), we could assign the following structure to the sentence in 7.1:



Not only is it argued that SLASH must be able to represent multiple categories, but it is also suggested that the value should be a set, and not a list -- as shown by the curly brackets around the NPs in the above structure. The reason for this is simply that the order in which the gaps meet their fillers must be allowed to vary. Sets, being unordered lists, provide the necessary flexibility. Thus, for instance, we shall be able to cope with examples such as:



If the value of SLASH on the XP node which represents the two gaps were a list, then the example in Fig 2 could not be handled -- we could only ever match the filler against the first item in the list. With sets, as shown, there is no problem.

In order to allow this, we shall therefore add sets to the grammar. To do this in ALE, in which of course the basic data structures are Prolog lists, it's necessary to employ

Prolog code which treats lists as sets -- you don't need to understand exactly how this is done, but it is a good idea to follow the reasoning.

In the type hierarchy, we shall represent sets as types which have two features -- an element, which is a syntactic object, and a set of elements. This is almost exactly the same representation as lists:

```
set sub [e_set,ne_set].

ne_set sub []
  intro [elt:loc,
         elts:set].
```

Note that we are assuming that the only sets we shall be interested in are sets of local values. SLASH will now take sets as values, and in order to refer to a particular item in one of these sets, all we need to do is pick out an element using the `slash:elt:Element` path.

Inherited and To-Bind Features

Here we shall approach the implementation of the NonLocal Feature Principle as suggested in the book and quoted in the last handout:

NonLocal Feature Principle (p.162)

The value of each nonlocal feature on a phrasal sign is the union of the values on the daughters.

We can now handle the `union' part of this as we can have sets as values. However, to get the principle to work generally across all the schemas, HPSG assumes that the NONLOCAL feature actually introduces two intermediate features, both of which take SLASH as values. These are called the `feat` and TOBIND features, and the idea is that the INHERITED value on the mother is the accumulation of all the INHERITED values on the daughters, minus the `tobind` values on the daughters. This implementation will allow a single statement to apply to all the schemas -- in practice, in every case but the filler-head schema, we shall simply accumulate the `inherited:slash` values, but the filler-head schema will introduce a `tobind:slash` value which will effectively be deleted from the set to be passed on. The relevant parts of new rule, then, look like this:

```
filler_head rule
(Mother, synsem ....
===>
cat> (FillerDtr,synsem:loc:Slash),
cat> (HeadDtr,synsem:(loc ....
                      nonloc:(inherited:slash:elt:Slash,
                              tobind:slash:elt:Slash))),
goal> (head_feature_principle(Mother,HeadDtr)),
```

```
goal> (nonlocal_feature_principle(Mother,HeadDtr,[FillerDtr])),
goal> (marking_principle(Mother,HeadDtr)).
```

The filler's local values must match the `inherited` and `tobind` value on the head, and the NLFP will ensure that the mother's value for `inherited:slash` represents the head daughter's value minus the filler.

This is clearly rather involved, and as mentioned above it's not necessary to understand the code which implements the NLFP. However, the analysis in `grammar7.pl` is generally faithful to the account in Chapter 9 of Pollard & Sag, and it is worthwhile making sure that the theoretical side of the exposition is clear.

One final point to make here is that the HPSG book assumes another two features in the value of INHERITED and TOBIND -- QUE and REL. The first is used in the analysis of interrogatives, although the book does not actually discuss these, and the second appears in relative clauses, which we shall ignore here.

Lexical Rules

Current HPSG uses lexical rules a great deal. At their simplest, `lexical redundancy rules' are used to reduce the number of lexical entries. Thus, for instance, it is normal practice to assume that verbs should be represented just by the base form in the lexicon, and that the other forms should be generated using lexical rules. Applying this to our grammar, the following ALE lexical rule will generate the 3rd person singular forms of verbs:

```
pres_3s lex_rule
   (synsem:(loc:cat:(head:(verb,
                           vform:bse,
                           aux:no,
                           mod:Mod,
                           inv:Inv),
                     spr:Spr,
                     subj:Subj,
                     comps:Comps,
                     marking:Marking),
             nonloc:(inherited:slash:e_set,
                     tobind:slash:e_set)))
  **>
   (synsem:(loc:cat:(head:(verb,
                           vform:fin,
                           aux:no,
                           mod:Mod,
                           inv:Inv),
                     spr:Spr,
                     subj:Subj,
                     comps:Comps,
                     marking:Marking),
             nonloc:(inherited:slash:e_set,
                     tobind:slash:e_set)))
  morphs
```

```
(X,y) becomes (X,i,e,s),
 X becomes (X,s).
```

The input feature structure represents the base form of the verb (*like*, for instance), and the output is a direct copy except that the VForm value is now `fin`. Furthermore, the `morphs` statement is used to ensure that the 3rd person singular will be *likes*, and that the same form of a verb such as *worry* will be *worries*. Similar rules will be used to generate all the other parts of the verb paradigm -- the present and past participles, the passive participle, and so on.

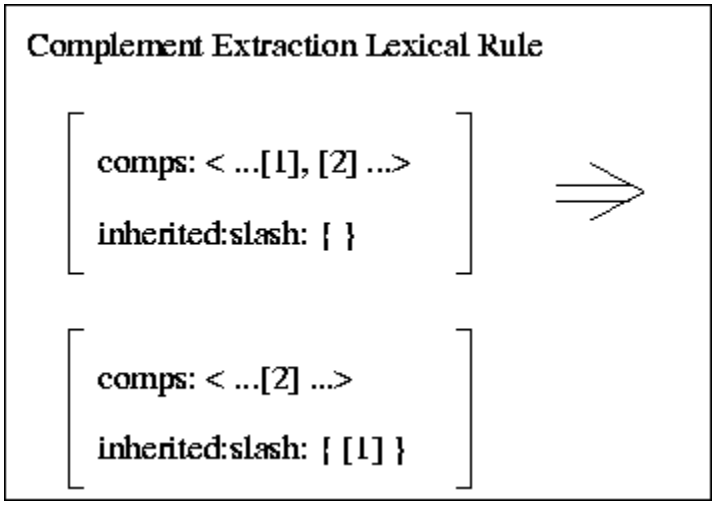
HPSG goes further with the use of lexical rules, using them to handle a wide range of grammatical phenomena. English passive structures, for instance (*Sandy was believed*, and so on), can be analysed in this way, and as we shall now see, lexical rules can also be used to eliminate traces from the grammar. Two rules are employed to do this in the theory, the Complement Extraction Lexical Rule and the Subject Extraction Lexical rule, as discussed below.

Complement Extraction lexical Rule (CELR)

In Chapter 9 of Pollard & Sag, the CELR is introduced as follows (p.378):

The basic idea is that SLASH originates not on traces, but rather from the head that licenses the `missing' element. ... We assume there is a lexical rule ... that takes as input a lexical entry with a certain COMPS list and returns as output a lexical entry that is just the same except that one element has been removed from the COMPS list and placed within the INHER:SLASH value.

The basis of the rule can therefore be represented by the following input and output structures:



Given a verb such as *likes*, this rule should apply to produce something with has an empty comps list and an NP in the `inherited:slash` value -- exactly the same structure, in fact, as we would get from combining the verb with trace.

To get this working in ALE, we need to introduce operations on the `comps` list, and we can do this using an `if` statement in the lexical rule, which (simplified) will look like this:

```
celr lex_rule
   (synsem:(loc:cat:(comps:Compsin),
            nonloc:(inherited:slash:e_set,
                    tobind:slash:e_set)))
  **>
   (synsem:(loc:cat:(comps:Compsout),
            nonloc:(inherited:slash:(elt:Loc,
                                     elts:e_set),
                    tobind:slash:e_set)))
  if
    select((loc:Loc),Compsin,Compsout)
  morphs
    X becomes X.

select(X,(hd:X,tl:Z),Z) if true.
select(X,[Hd|Y],[Hd|Z]) if select(X,Y,Z).
```

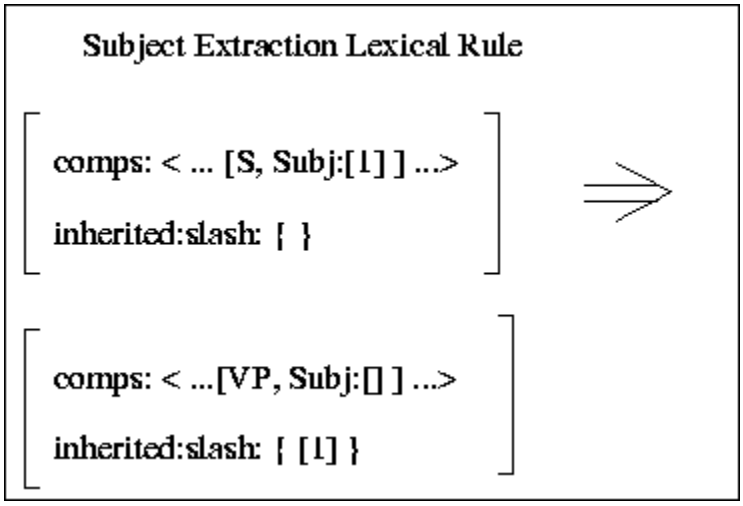
Again, it's not necessary to understand the mechanics of the Prolog here, but you should note that the effect is to remove something (represented by the variable `Loc` above) from the `Compsin` list and place it in the `inherited:slash` set, leaving the rest of the complements in the `Compsout` list. Note also that this will result in three lexical entries for a verb such as *give* which has two complements -- one with the first complement in the SLASH set, one with the second, and one with neither. This will ultimately mean that sentences like *the book Sandy gave Kim* will be ambiguous depending on whether *the book* is the object given or the `receiver' (as in *Kim Sandy gave the book*) -- clearly the receiver option is wrong semantically, but it's assumed that that's a different problem.

In order for the above rule to work, all that's required is that the other category features (`head`, `spr`, `subj`, and `marking`) should be added and copied from the input structure to the output, as in the 3rd person singular rule we saw earlier. Note that we don't need to copy all the `head` features separately as we actually want an exact copy of the input feature structure for everything except the `comps` and `inherited:slash` values. Finally, note that the lexical rules apply recursively, applying to their own output and to the output of other lexical rules where the input feature specifications match, so the 3rd person singular rule and the CELR will combine to produce two versions of *like*, one with and one without the complement extracted, and two versions of *likes* also.

Subject Extraction Lexical Rule (SELR)

Using trace, sentences such as *Kim Sandy believes likes books* can be analysed with no extra effort -- NP trace can be the subject of *likes*, bound to *Kim* in the normal manner. The CELR will not handle these alone, of course, as only complements of the verb are affected, and so we must also implement the SELR.

The SELR is a little more complicated than the CELR, for various reasons, an important one being that subject extractions are much more limited than complement extractions -- for further discussion, see the relevant parts of Pollard & Sag (pp171ff, for example). Generally, the theory assumes that it is not possible to affect the `subj` values directly in the same way as the CELR operates on the `comps` list, and the SELR therefore works only on verbs which have sentential complements. The idea is thus that the rule `reaches inside' the `comps` list of a verb like *believes* and extracts the subject of the embedded sentence. Schematically, the SELR looks like this:



As shown, the `subj` value on the embedded sentence is placed on the `inherited:slash` value on the main verb. In ALE, we require a bit of Prolog code to get this working, and so the relevant parts of the SELR are:

```
selr lex_rule
   (synsem:(loc:cat:(comps:Compsin),
            nonloc:(inherited:slash:e_set,
                    tobind:slash:e_set)))
  **>
   (synsem:(loc:cat:(comps:Compsout),
            nonloc:(inherited:slash:(elt:Loc,
                                     elts:e_set),
                    tobind:slash:e_set)))
  if
    subst((loc:(cat:(head:vform:fin,
                     subj:e_list,
                     comps:e_list))),
          (loc:(cat:(head:vform:fin,
```

```
                    subj:[(@ np,loc:Loc)],
                    comps:e_list))),
          Compsin,
          Compsout)
  morphs
    X becomes X.

subst(X,Y,(hd:X,tl:Z),(hd:Y,tl:Z)) if true.
subst(X,Y,[Hd|Tl1],[Hd|Tl2]) if subst(X,Y,Tl1,Tl2).
```

Once again, the other category features must be added to get the full working version of the rule. With these in place, the grammar should once again handle sentences such as:

(7.2) The red book Kim likes

(7.3) The red book Sandy believes Kim likes

(7.4) Kim Sandy believes likes the red book

Summary

This part of the course introduced the current analysis of long-distance dependencies in HPSG, which mainly involved a look at the INHERITED and TOBIND features. We then looked at lexical rules and at how empty categories can be eliminated by the the use of lexical rules which operate on the valency features.

Exercises

Here we implement a fairly standard HPSG account of long distance dependencies using the INHERITED and TOBIND features and lexical rules. We also implement an analysis of passive sentences using lexical rules.

Solutions to these exercises can be found in `grammar8.pl`.

1. The file `grammar7.pl` provides the starting point for these exercises, so and make sure you understand the contents -- as the text suggests, the actual Prolog code is not so important, but an understanding of what it's doing is.

The grammar includes the new type `set` and a more standard version of the Non-Local Feature Principle as discussed in the text. Note that the `noslash` macro now has the following definition:

```
noslash macro
   nonloc:(inherited:slash:e_set,
           tobind:slash:e_set).
```

The lexical entry for trace is the same as the previous grammar, however, and so it produces an error message. Trace should now unify its `loc` value with an element in the set represented by `inherited:slash`, and the `tobind:slash` value should be `e_set`. Alter the entry for trace to take account of the new features and verify that the grammar now parses sentences such as **the book Sandy Kim gave e e*. If it's not clear how to pick out a set element, look at the filler-head rule and the discussion in the text for clues.

2. We now want to start using lexical rules to generate verb entries, so, ignoring the auxiliaries, remove all main verb entries from the lexicon except the base forms -- replacing the `(fin;bse)` specifications with `bse` will be appropriate in at least some cases. Now implement the `pres_3s` (3rd person singular present tense) lexical rule as described in the handout and check that *likes* is being generated. Note that *Kim snored* is not now a possible test sentence, unless you provide another lexical rule to generate past tense forms -- *Kim snores* should be ok, though. Add a (transitive) lexical entry for *worry* and check that you can parse *the book worries Kim*.

3. The handout provides the basis for ALE versions of the CELR and the SELR, and these partial representations are already present in grammar7.pl. Remove trace from the grammar and fill out the category specifications in the lexical rules (use the `pres_3s` rule as a guide). Note that, as the handout suggests, the head features can be copied using a singular variable where it was necessary to spell them out in the `pres_3s` rule, so all we need in this case is:

```
 ... head:Head, .. etc
**>
 ... head:Head, .. etc
```

With the rest of the valency features and the marking feature in place, check that you can parse *Sandy Kim likes* (no `e' now, of course) and *the book Sandy gives Kim* -- the latter with two analyses as discussed in the handout. Note that it may be necessary to exit from ALE and start again from scratch at this stage -- the compiler sometimes gets confused when lexical rules are being edited.

4. It was mentioned in the handout that lexical rules are used to generate passive sentences in English, such as *Kim was liked*. As a first stage to getting this to work, we need to add the type `pas` (passive participle) as a subtype of VFORM. Check the result using `show_type(vform)`:

```
TYPE: vform
SUBTYPES: [fin,pas,bse]
SUPERTYPES: [bot]
IMMEDIATE CONSTRAINT: none
MOST GENERAL SATISFIER:
     vform
```

You can use the `s' command to check the whole type signature if you're using the ALE Emacs interface.

5. Now we want a lexical rule which will generate suitable participial forms such as *liked* and *given*, and which will also alter the feature structure to capture certain facts about passives. For the moment, we shall ignore the *by*-phrases which often appear (*Kim was given the book by Sandy*) and concentrate on getting *Kim was given the book*. One obvious factor is that the object NP in a VP like *give Sandy the book* has become the subject in *Sandy was given the book*, and this is what we want the lexical rule to achieve.

We therefore need a rule which will move an NP from the `comps` list of a verb entry and place it in the `subj` value of the output structure. The relevant part of an ALE version of this will be:

```
   head:(verb,
         vform:bse, .... etc ...
   comps:([loc:(cat:(head:(noun,
                           prd:ObjPrd,
                           mod:ObjMod),
                     subj:[],
                     comps:[],
                     marking:ObjMarking))|ObjRest]),
**>
 head:(verb,
       vform:pas, .... etc ...
 subj:[(loc:(cat:(head:(noun,
                        prd:ObjPrd,
                        mod:ObjMod),
                  subj:[],
                  comps:[],
                  marking:ObjMarking)))],
 comps:ObjRest,
     .
     .
morphs
 give becomes given,  .... etc ...
```

Here we have split the list on the input category into its head and tail using the usual Prolog `[Head|Tail]` syntax, the head of the list being the whole structure representing the object NP. (It is assumed that the first complement is the object). This feature structure is then copied into the `subj` value on the output, and the `comps` value on the output is the tail of the original `comps` list. All of the other parts of the input must also be copied, of course.

So, the next stage is to implement such a lexical rule and check that you get a lexical entry for *liked* as a result -- note that you should get two entries for *given*, one in which the CELR has operated in order to allow *the book Sandy was given*.

6. Finally, all we need is a lexical entry for the passive auxiliary *was*, which will be exactly like the other auxiliaries except that it takes a passive VP (VForm:pas) as its complement. With this in place, check that you can parse *Sandy was liked*, *Sandy was given the book*, *the book Sandy was given*, *Kim Sandy believes was given the book*, and any other suitable sentences you can think of!

10 Semantics

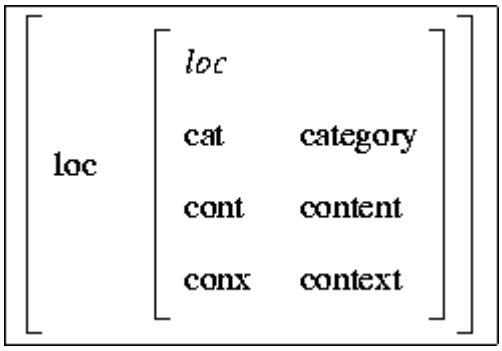
Introduction

This is the eighth instalment of the course on HPSG and ALE. The only major parts of the HPSG feature structures which we have not discussed contain the semantics information which is represented by the CONTENT and CONTEXT features. This part of the course therefore looks at some of the issues and at how semantic representations can be constructed in ALE.

It should be noted that the account of the semantics of quantification changes in the book -- the early discussions assume an older analysis which, as stated on p.47, cannot handle certain problems concerning quantifier scoping. We shall therefore ignore the relevant parts of the first analysis and concentrate on the final representation used in the book in chapter 8. This does mean, however, that some aspects of most of the semantic structures which appear in the first 7 chapters are incompatible with the discussion below.

Semantic in HPSG: the Content and Context Features

The theory as described in the book introduces the CONTENT and CONTEXT features, which contain the semantic information, as part of the Local information in a sign:



The CONTEXT feature handles various kinds of background semantic information relating to things like time and place of utterance, presuppositions, and so on. We shall ignore this here and concentrate on the `core' semantic information represented by the CONTENT feature.

The general semantic theory which is assumed is SITUATION SEMANTICS (see, for example Richard Cooper (1990)). The details of the theory are not important here -- the basic idea is that semantic representations should contain information on the context of utterance, and so on. An important aspect of this is the notion of a `state-of-affairs', which is a partial description of some situation, sometimes called `circumstances'

or (in the Situation Semantics literature) `infons'. See section 8.2.2 below for a description of states-of-affairs. HPSG semantics is also closely related to Discourse Representation Theory (DRT) -- Kamp (1981) is a standard reference.

The structure of the CONTENT feature differs for different types of object -- nominal semantics is clearly different from verb semantics, and so forth. We shall assume that the value of the CONTENT feature `cont` is the type `cont`, and that this has three sub-types -- `nom_obj` (Nominal-Object), `psoa` (State-of-Affairs), and `quant` (Quantifier). We shall look at these in turn, beginning with the Nominal-Object type, which represents the semantics of nouns. Following the above points, though, note that the ALE representation of the top-level structure of semantics is simply:

```
loc sub []
     intro [cat:cat,
            cont:cont].

cont sub [nom_obj,psoa,quant].
```

Nominal Objects

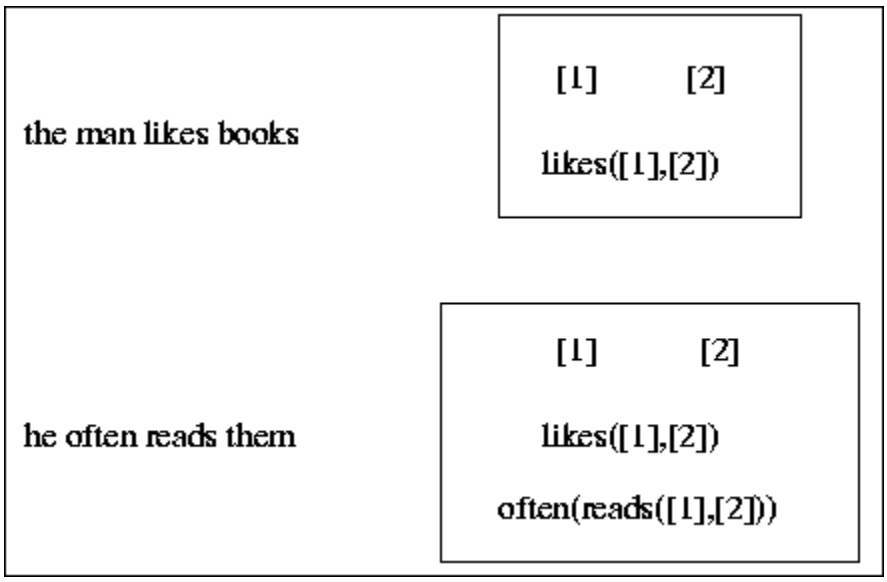
The `nom_obj` type, which is appropriate for nouns, introduces two features, INDEX and RESTRICTION:

```
nom_obj sub []
  intro [index:ind,
         restr:set_psoa].
```

We shall look at the RESTRICTION feature below. Firstly, the following section looks at the INDEX feature as this is a crucial part of the HPSG analysis of discourse structure and binding.

The Index Feature

The INDEX feature works in much the same way as reference markers in DRT, in which the use of a nominal normally results in a marker being introduced into the current representation. Thus, for instance, a sentence such as *the man likes books* will result in a discourse representation in which there are two markers representing *the man* and *books*, and it is these markers which will be identified if the subsequent sentence is something like *he often reads them* -- *he* should be identified with the marker for *the man*, and *them* should pick out *books*. The following diagram summarises this:



These indices are also used in the HPSG account of binding theory -- thus, for instance, in a sentence such as *John shaved himself*, the indices for *John* and *himself* should be shared. The INDEX feature takes the type `ind` as its value, and this in turn has three subtypes -- `it`, `there`, and `ref`. The first two are used to account for verbs which take `expletive' subjects, as in the following examples:

(8.1) It snowed last week

(8.2) There appears to be a problem

The `ref` (Referential) type covers most other cases in which nouns are used to refer to objects.

One important point about the INDEX feature is that it contains agreement features -- for present purposes, we shall assume that these are represented by GENDER, NUMBER, and PERSON. The relevant part of the ALE type hierarchy will therefore be:

```
ind sub [it, there, ref]
  intro [gen:gen,
         num:num,
         per:per].
```

The type `gen` will have the basic subtypes `masc`, `fem`, and `neut` (not appropriate in all languages, of course), `num` will include `sing` and `pl` (and `dual', and so on, where necessary), and finally `per` will have subtypes `first`, `second`, and `third`.

The Restriction Feature

The other aspect of nominal semantics which we should look at here concerns the RESTRICTION feature which, as noted above, takes a set of states-of-affair (`psoa`)

types as value. The idea behind this feature is that it contains the basic semantic information on a referent -- so the RESTRICTION value in the lexical entry for *book*, for example, contains the basic semantic information that the object concerned is a book. In an NP such as *red book*, the restriction is a set representing the facts that the object is a book and is red. These facts are contained in the type `psoa`, and the following section investigates this.

States-of-Affairs

The analysis of states-of-affairs is the main part of the semantics which changes in chapter 8 of the Pollard & Sag book, and so this discussion is not relevant to the earlier chapters. The `psoa` type now represents a `possibly quantified state-of-affairs', and its ALE type structure is:

```
psoa sub []
  intro [quants:list_quant,
         nucleus:qfpsoa].
```

The `quants` feature contains a (possibly empty) list of quantifiers, while the `nucleus` feature holds the core semantic information in the type `qfpsoa` (`quantifier-free psoa'). Having the quantifiers in a list like this means that quantifier scope ambiguities can be captured -- the following sentence is assumed to exemplify this (Pollard & Sag, p.47):

(8.3) Every man loves some woman

The argument is that this represents a situation in which there is a single woman who is loved by all the men, or that there is a unique woman for each of the men -- the former meaning is perhaps more obvious in sentences such as:

(8.4) Everyone went to a restaurant

The ordering of the quantifiers in a list allows the different interpretations to be extracted.

The `qfpsoa` type will have as many subtypes as are deemed necessary to capture the basic facts about noun and verb semantics. Thus the following ALE type hierarchy is possible:

```
qfpsoa sub [property,un_relation, bin_relation, tri_relation].

property sub [book,red, .... ]
   intro [instance:ref].

book sub [].

un_relation sub [snore,walk,run, .... ].
```

```
snore sub []
  intro [snorer:ref].

bin_relation sub [like,hit, .... ].

like sub []
   intro [liker:ref,
          likee:ref].

tri_relation sub [give,sell, .... ].

give sub []
   intro [giver:ref,
          given:ref,
          gift:ref].
```

Here we are assuming that `properties' such as being red, or being a book, are `atomic' semantic representations introducing the feature INSTANCE which will be bound to the index of a referential NP. Unary relations such as *snore*, however, introduce a `snorer` feature, whose value is again a referential NP which will ultimately be associated with the subject of the sentence, and so on for the other verb types.

Quantifiers

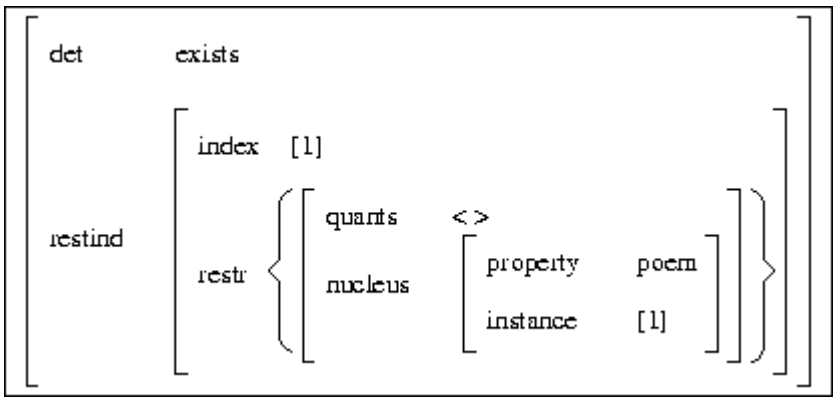
The `quant` type itself (the last subtype of `cont`), can be represented in ALE as follows:

```
quant sub []
  intro [det:sem_det,
         restind:nom_obj].
```

The first feature, DETERMINER, is used simply to name one of three types of semantic determiner:

```
sem_det sub [forall,exists,the].

forall sub [].
exists sub [].
the sub [].
```

The other feature, RESTIND, contains a `restricted index', which is a nominal with a non-empty restriction set -- so quantification must range over properties, and so on. Based on this approach to the semantics of quantifiers, Pollard & Sag suggest (almost) the following representation for an NP like *a poem*:



The next section looks at how these features appear in lexical entries and also at how these can be implemented in ALE. For the moment, though, it should be noted that we need to refine the definition of lists and sets that we have been using -- the above discussion introduces types such as `list_quant` and `set_psoa`. This can be implemented easily by defining new subtypes of `list` and `set`:

```
list sub [ne_list,list_sign,list_synsem,list_quant].

set sub [ne_set,set_psoa,set_loc].
```

Each of the new subtypes of list will have different types of member -- for example:

```
ne_list sub [ne_list_sign, ne_list_synsem,ne_list_quant]
  intro [hd:bot,
         tl:list].

list_synsem sub [e_list,ne_list_synsem].

ne_list_synsem sub []
       intro [hd:synsem,
              tl:list_synsem].
```

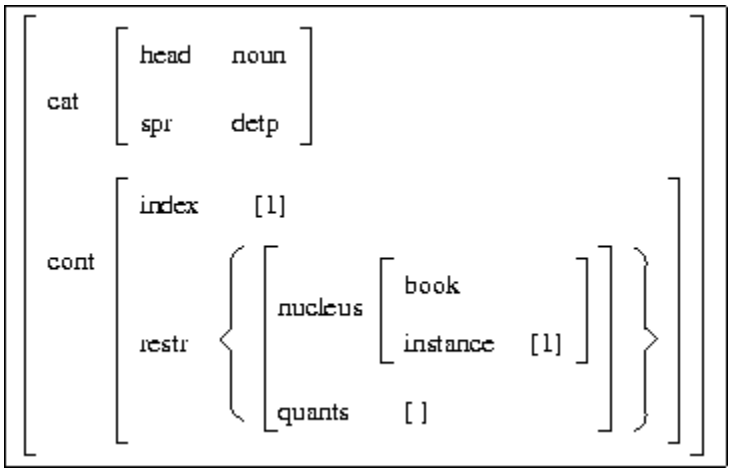
We can now be more precise in the specification of lists and sets as values -- the value of the valency features, for instance, will be of type `list_synsem` rather than just `list`.

Semantics in ALE Lexical Entries

Each lexical item must now include semantic information. The following subsections look at the various types of lexical entry that we have been working with -- nouns, verbs, determiners, and adjectives.

Noun Semantics

Common nouns will include the INDEX and `restr` features as they are clearly of the `nom_obj` type:



The main point here is that the values of the INSTANCE and INDEX features are unified -- as discussed above, the restriction imposed by the common noun is associated with a marker in the discourse, represented by the index.

This can be straightforwardly captured in ALE:

```
book --->
    synsem:((@ nbar((per:third,
                     num:sg,
                     gen:neut)),
               @ mod(none),
               @ spr([(@ detp(_))]),
               @ cont((index:Ind,
                       restr:(elt:(nucleus:(book,
                                            instance:Ind),
                                   quants:[]),
                              elts:e_set))),
               @ no_slash)).
```

Proper nouns are much simpler -- they just introduce a referential index with no restrictions:

```
kim --->
   synsem:((@ np((per:third,
                  num:sg,
                  gen:masc)),
              @ mod(none)),
              loc:cont:(index:ref,
                        restr:e_set),
              @ no_slash).
```

Note the use of a number of simple new macros in the latter two entries -- `(@ mod)`, `(@ spr)`, and `(@ cont)` -- and also the addition of the agreement features as an argument in the NBar and NP macros. The former will now be:

```
nbar(Ind) macro
  loc:(cat:(  ..... ),
       cont:index:Ind).
```

Verb Semantics

The lexical entries for verbs will ensure that the indices representing their subjects, and any complements, are used in the definition of the semantic relations that the words describe. For example, *likes* will relate the index introduced by its subject to the `liker` element in its semantics, while the object NP will be the `likee`:

```
like --->
   synsem:(loc:(cat:(head:(verb,
                           vform:bse,
                           aux:no,
                           inv:no),
                     spr:[(@ comp)],
                     subj:[(@ np(Ind1))],
                     comps:[(@ np(Ind2))],
                     marking:unmarked),
                cont:(nucleus:(like,
                               liker:Ind1,
                               likee:Ind2),
                      quants:[])),
                @ mod(none),
           @ no_slash).
```

Auxiliary verbs will simply pick up the semantics of their complement VPs:

```
does --->
  synsem:(loc:(cat:(head:(verb,
                          mod:none,
                          vform:fin,
                          aux:yes),
                    spr:Spr,
                    subj:Subj,
                    comps:[(@ vp(Prop),
                            @ subj(Subj),
                            @ spr(Spr),
                            loc:cat:head:vform:bse)],
                    marking:unmarked),
               cont:Prop),
          @ noslash).
```

The `cont` value on the auxiliary is simply unified with the head VP's `cont` value. There are some new macros in use here too -- `@ subj` and `@ spr` -- and the `vp` macro now has an argument, which will be the `proposition' expressed by the verb's semantics:

```
vp(Proposition) macro
  loc:(cat:head:verb,
            comps:[],
            subj:[_]),
       cont:Proposition).
```

Determiner Semantics

The quantifiers will be classified depending on whether they are of the `forall`, `exists`, or `the` types -- for instance:

```
every --->
   synsem:(loc:(cat:(head:(det,
                           spec:((@ cont(Cont),
                                  @ nbar((num:sg))))),
                     spr:[],
                     subj:[],
                     comps:[],
                     marking:unmarked),
                cont:(det:forall,
                      restind:Cont)),
           @ noslash).
```

Here the value of the RESTIND feature in the determiner's CONTENT is unified with the CONTENT value of the specified Nbar.

Adjunct Semantics

Perhaps the most complicated semantics belongs to the adjuncts. As noted earlier, we want an NP like *the red book* to introduce an index which is restricted by the semantic attributes `red-ness' and `book-ness'. Adjuncts therefore add a new restriction to whatever the set of restrictions is on their head:

```
red --->
  synsem:(loc:(cat:(head:(adj,
                          prd:no,
                          mod:(pre_mod_synsem,
                               @ nbar(Ind),
                               @ restr(Restrs))),
                    spr:[],
                    subj:[],
                    comps:[],
                    marking:unmarked),
               cont:(index:Ind,
                     restr:(elt:(nucleus:(red,
                                          instance:Ind),
                                 quants:[]),
                            elts:Restrs))),
          @ noslash).
```

Note that yet another macro is in use here - `@ restr` -- and that the value of the `restr` feature on the modified element is unified with the `elts` set in the CONTENT feature. Thus, effectively, the restriction `red` is being added to the already-existing set.

The only other type of lexical entry we have been using is for complementisers, which it is assumed have no semantics. The CONTENT feature can therefore just be left unspecified.

There is an important aspect of HPSG semantics that we have not seen, and this is the principle which ensures that the CONTENT values are passed from daughters to mothers. This is handled by the Semantics Principle, discussed in the next section. Before going on, however, note that the lexical rules also have to be edited to include the CONTENT feature, and that the inflected verb rules (3rd person singular, and so on) should now ensure that the correct agreement features appear.

The Semantics Principle

The basic idea behind the Semantics Principle is that the CONTENT value of a mother is unified with the head daughter's CONTENT value, except when the non-head is an adjunct, in which case the CONTENT on the mother comes from the adjunct daughter. This is a slightly simplified version of the final principle, which ultimately includes a notion of quantifier `storage' which we shall ignore for present purposes -- see Pollard & Sag pp323ff for further details.

All we need to do to implement the Semantics Principle in ALE is to include the following definition:

```
semantics_principle(synsem:loc:cont:Cont,
                    synsem:loc:cont:Cont)
   if true.
```

This principle should now be added to all the rules as a goal:

```
semantics_principle(Mother,HeadDtr)
```

With the exception, of course, of the adjunct rules, in which the goal should be:

```
semantics_principle(Mother,AdjDtr)
```

Viewing Semantics Features

The file `grammar9.pl` contains an implementation of all the semantics features discussed above. The grammar is clearly becoming quite complicated, and the feature structures representing a complete parse can be extensive. ALE allows the printing of certain features to be suppressed using the `no_write_feat` command:

```
| ?- no_write_feat(cat).

yes
| ?- no_write_feat(nonloc).

yes
```

If we subsequently parse a sentence like *Kim likes Sandy* and view the results, we should get:

```
sign
SYNSEM synsem
       LOC loc
           CAT ...
           CONT psoa
                NUCLEUS like
                        LIKEE ref
                              GEN fem
                              NUM sg
                              PER third
                        LIKER ref
                              GEN masc
                              NUM sg
                              PER third
                QUANTS e_list
       NONLOC ...
```

As the lexical entries for *Kim* and *Sandy* now include agreement features, and we have specified that *Kim* is masculine and *Sandy* feminine, the semantics reflects this in the analysis of the sentence -- the `liker' is masculine, and so on. The topicalised version of the sentence, *Sandy Kim likes*, will have exactly the same semantics.

Similarly, the grammar will produce almost identical analyses for the passive sentence *Kim was given the book* and the active form *Sandy gave Kim the book*. The only difference is that in the latter case the `giver' is unspecified -- the rule ensures that *Kim* is the `givee' (`receiver') in both cases. With the addition of an appropriate analysis of prepositional phrases, the semantics of *Kim was given the book by Sandy* should be identical to the active form.

And finally, note that the grammar in `grammar9.pl` excludes the verb *believe* as its semantic representation introduces problems which are the subject of the next part of the course.

11 Raising and Control

Introduction

The previous eight parts of this course were concerned with an explanation of the various features which form the core of current HPSG, and with how the theory can be implemented in ALE. All the main apparatus is in place in the grammar in the file `grammar9.pl`. Here we are concerned with using the theoretical and practical system in the analysis of some classical linguistic problems, namely an account of RAISING and CONTROL structures.

Before looking at the data immediately concerning raising and control, some background discussion of the current HPSG account of complementation is useful. We then look at some basic raising and control structures and finally at the behaviour of the English expletive pronouns.

Complement Structures

In the earlier chapters of the Pollard & Sag book, there is an important discussion concerning `unsaturated complements' (pp123ff). The relevant data are initially cited as follows:

(9.1) Sandy preferred [(for Lee) to leave]

(9.2) * Sandy preferred [Lee to leave]

(9.3) We have plans [(for Dana) to leave]

(9.4) * We have plans [Dana to leave]

(9.5) It is possible [(for Leslie) to leave]

(9.6) * It is possible [Leslie to leave]

The complement structures in square brackets are `unsaturated' when the optional subject is missing -- in the analysis assumed in the earlier chapters, this means that the SUBCAT feature contains an NP, representing the subject, which has still to be found. In the chapter 9 analysis, the situation is different -- the complements list is fully satisfied in both cases, and so `unsaturated complements' is a bit of a misnomer in terms of the later account. However, the central problem remains -- why are the examples in 9.1, 9.3, and 9.5 grammatical while the others are not?

The explanation offered in the book is that infinitival VPs with NP subjects and complementisers, and `bare' infinitivals, form a natural class -- the book gives examples

of a number of environments in which these structures are acceptable, including the following:

(9.7) What did they prefer?
– For Kim to be reassigned
– * Kim to be reassigned
– That Kim be reassigned
– * Kim be reassigned

(9.8) a. That Dana was unhappy was obvious
b. * Dana was unhappy was obvious

(9.9) What did they believe?
– It was that they will be reassigned
– * It was they will be reassigned

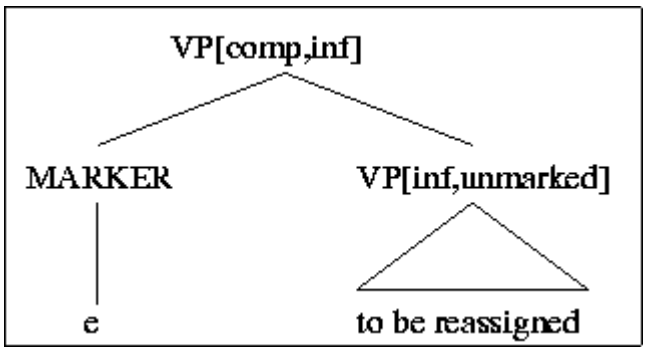
(9.10) a. What they promised was that Kim would be reassigned
b. * What they promised was Kim would be reassigned

The HPSG book refers to these contexts as *free* environments, and assumes that central observation is that that the bare infinitival VPs appear in almost exactly the same positions:

(9.11) What did they prefer?
– To be reassigned
– It was to be reassigned

(9.12) To be happy is difficult

The analysis in the book in chapter 3 assumes that the free environments are specified as +comp positions -- so a verb such as *prefer*, for example, specifies that its complement must have a complementiser. In order to get this to work with the bare infinitives too, Pollard & Sag assume that there is a null complementiser which marks the VPs appropriately:



Much of the relevant section in the book is aimed at contrasting the HPSG account with the standard GB approach. The general trend in the later part of the book is to avoid empty categories altogether, and there are probably two obvious ways of handling the data. Firstly, a lexical rule could be used to ensure that verbs like *prefer*, and so on, are subcategorised for either bare infinitives or infinitival sentences with *for* complementisers. This solution does not capture the idea that these structures form a natural class, and so the second solution would be to use the type hierarchy to define a clausal supertype which subsumes the two structures.

Raising and Control Structures

In the Pollard & Sag book, the term EQUI is used more or less as a synonym for control. As the book notes, the terms equi and raising relate to early transformational accounts of data such as the following:

(9.13) Equi verbs and adjectives:

VP[inf] complements	try, hope, persuade, eager (*try to leave*)
VP[ger] complements	consider, try (*considered leaving*)
AP[+PRD] complements	feel, look (*felt sick*)
NP[+PRD] complements	make (*made him a good student*)
PP[+PRD] complements	got, count (*got under the table*)

(9.14) Raising verbs and adjectives:

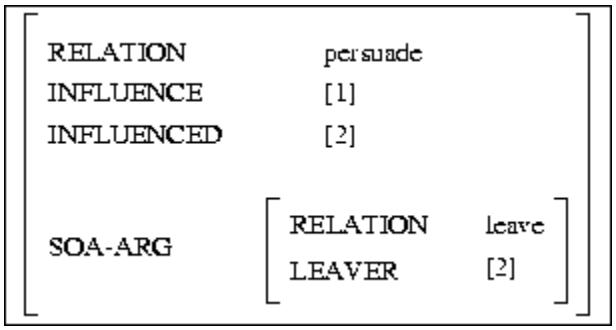
VP[inf] complements	appear, seem, likely (*seem to be happy*)
VP[prp] complements	begin, keep, be (*began keeping bees*)
AP[+PRD] complements	become, seem, be (*became sick*)
NP[+PRD] complements	become, be (*became a good student*)
PP[+PRD,as] complements	regard, strike (*struck them as stupid*)
PP[+PRD] complements	be, seem (*is in trouble*)

There are three main reasons for distinguishing these classes, which we shall investigate shortly. Firstly, however, it is useful to look briefly at the semantics associated with these structures.

Semantics of Control

A central assumption in HPSG is that the distinguishing characteristics in many of the above cases are basically semantic. Thus, for instance, for verbs such as *order* and *permit*, the type of the semantics is a psoa of the type *influence*, and this type introduces three semantic roles -- the INFLUENCE (possibly an agent), the INFLUENCED (typically animate), and SOA-ARG (the action that is involved). Thus in a sentence such as *Kim ordered Sandy to leave*, *Kim* is the INFLUENCE, *Sandy* the INFLUENCED, and

leave is the SOA-ARG. Similarly, the semantics of a sentence such as *Kim persuaded Sandy to leave* can be partially represented as follows:



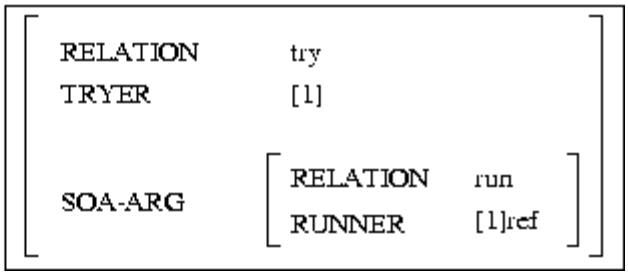
Here [1] will be identified with the subject, *Kim*, and [2] with the object, *Sandy*. As shown, the SOA-ARG feature includes the information that *Sandy* did the leaving.

The other cases are treated in a similar manner -- there are various roles which the arguments play, and there is a SOA-ARG feature which defines the action, or state-of-affairs, which is relevant for the particular lexical item in question. There is a full discussion of this material in chapter 7 of the HPSG book. The main point here is to note how the semantic relationships are captured.

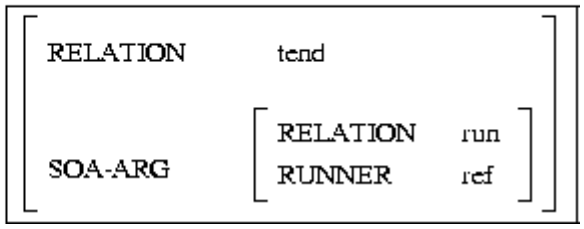
Equi/Raising Distinctions

It was noted above that there are three main distinctions drawn between equi and raising structures. Firstly, and most importantly, equi items assign one more semantic role, even though the surface syntactic relationships appear identical. Thus, for instance, Pollard & Sag provide the following analyses for the sentences *they try to run* and *they tend to run*:

(9.15) They try to run:



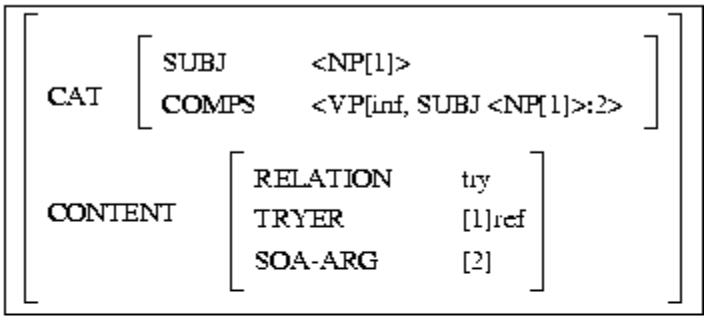
(9.16) They tend to run:



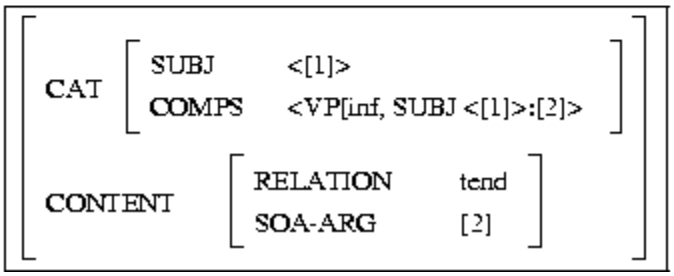
Thus the referential index representing the subject of an equi verb like *try* is assigned a role in the verb's content, and in addition the index is structure-shared with the index of the embedded subject. With a raising verb such as *tend*, the subject has no role in the matrix psoa.

The essential fact about the distinction, then, is that all the subcategorised dependents of equi verbs are assigned a semantic role, while raising verbs don't assign a role to one of their subcategorised objects. The lexical entries below illustrate this:

(9.17) Try:



(9.18) Tend:



Further examples of such verbs, including more complicated cases such as *persuade* and object raising *believe*, can be found in chapter 3 of the HPSG book.

Another important distinction is that only the raising expressions allow expletive subjects:

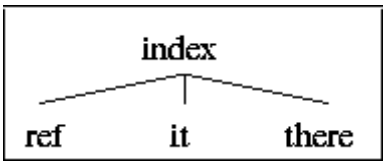
(9.19) There seems to be a problem

(9.20) * There tries to be a problem

(9.21) It is likely that they will lose

(9.22) * It is eager that they will lose

These facts are partly captured by the sorts of index assumed in HPSG:



The *it* and *there* sorts identify the two expletive NPs in English, while all other NPs are referential. The lexical entries for equi expressions like *try* include the information the their subjects are referential, while the raising type makes no such restriction. Thus a raising verb like *appear* can have any type of subject:

(9.23) It appears to be a problem

(9.24) There appears to be a solution

(9.25) Kim appears to have an answer

The final distinction concerns the fact that in equi structures the unexpressed subject of the VP complement is coindexed with one of the other objects, while in raising structures the entire SYNSEM value of the missing subject is shared with another subcategorised dependent. This results in a number of subtle differences which tend to rely on information from languages such as Icelandic (see Pollard & Sag pp138-9). In English, there is some evidence based on the fact that PPs can be controllers in equi situations, but not in raising constructions. However, the arguments are fairly abstract, and it is enough here to point the relevant parts of the book.

Expletive Pronouns

The expletive pronouns *it* and *there* appear in only a few places in English. This follows from the fact that most NPs are assigned a semantic role, which is associated only with the referential type of NP. Typically, expletive *there* appears as the subject of a copula:

(9.26) There is a problem

(9.27) There's a good reason

Expletive *it*, on the other hand, appears as the subject of weather verbs, temporal expressions, and in a few other environments:

(9.28) It snowed all winter

(9.29) It's now 12 noon

(9.30) It annoys me that he snores

There are also a number of cases of *it* turning up in non-subject positions, for instance:

(9.31) We'll wing it this evening

(9.32) Make it quick

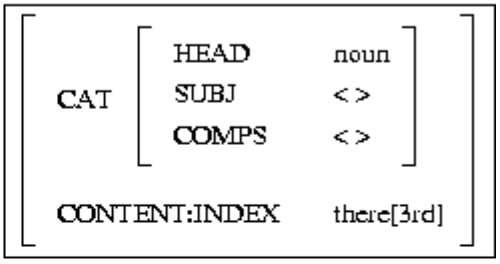
(9.33) He's right out of it

(9.34) I take it you'll pay

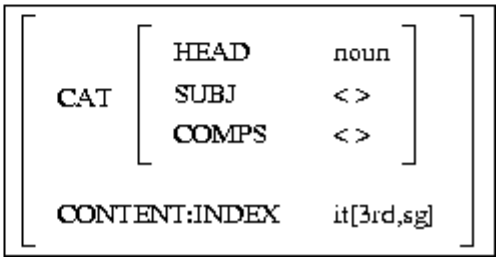
(9.35) Don't spread it around that he's stupid

The HPSG book's analysis of these involves the following signs:

(9.36) There:

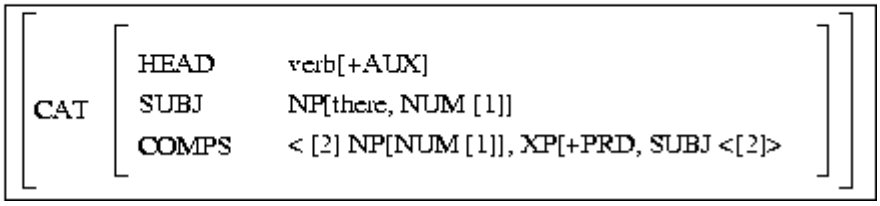


(9.37) It:



The relevant entry for copula *be* assumed in the book is as follows:

(9.38) Be:



The latter entry is enough to handle VPs such as *is no-one absent* with the following feature structure:

(9.39) VP[fin, SUBJ¡NP, there[3rd,sing]]

The content of this VP will be `complete', and it will only be able to combine with an expletive *there* subject.

ALE Implementation

Most of the above AVMs can be directly implemented in ALE. The grammar in `grammar10.pl`, which is a copy of some work by Suresh Manandhar and Claire Grover, contains examples of everything mentioned in the text. This grammar can be regarded as a starting point for research into a wide range of linguistic issues.

12 Weak Unbounded Dependencies and Binding Theory

Introduction

This is the final part of a course in implementing HPSG grammars in ALE; here I will look at a couple of important linguistic issues which the course has not yet covered. In fact, the HPSG book discusses a number of issues that the course ignores -- in general, these are either fairly straightforward to implement (such as the analysis of prepositions) or represent work which may be regarded as continuing research (such as `coerced complements').

However, it is useful to look further at two fairly traditional areas of concern which the book covers -- the treatment of `weak' unbounded dependencies such as the `tough-movement' cases, and the analysis of binding.

Weak Unbounded Dependencies

In Chapter 4, Pollard & Sag discuss a number of issues concerning unbounded dependency constructions (UDCs) -- we have already looked in some detail at the treatment of topicalised sentences such as *Kim Sandy likes*, which are classified as `strong' UDCs along with WH-questions, WH-relatives, it-clefts and pseudoclefts. We have so far ignored the weak UDCs, however, such as tough-movement, non-WH-relatives, and some others.

The basic distinction between strong and weak UDCs is that the former have an overt filler in a non-argument position which is strongly associated with a gap, for example (cf p157):

(10.1) *Strong UDCs*:

Kim Sandy likes (topicalisation)
I wonder who Kim likes (WH-question)
This is the book Kim likes (WH-relative)
It's Kim who Sandy likes (it-cleft)
What Kim likes is Sandy (pseudo-cleft)

The weak UDCs, on the other hand, have a constituent in an argument position that is in some sense co-referential with the gap:

(10.2) *Weak UDCs*:

I bought the book for Sandy to read (purpose infinitive)
Kim is easy to like (tough-movement)
This is the book Sandy likes (non-WH-relative)
It's Sandy Kim likes (it-cleft)

One important point to make about the latter cases is that the gap and the co-referential constituent may have different case:

(10.3) I (*nom*) am easy to please *e* (*acc*)

The HPSG assumption is that adjectives such as *easy* subcategorise for infinitive complements containing an accusative gap which is coindexed with the subject. Pollard & Sag suggest a partial lexical entry similar to the following:

(10.4) Partial entry for *easy*:

```
[ LOCAL:CAT [ HEAD  adj
              SUBJ  <NP[1]>
              COMPS <VP[inf, INHER:SLASH {[2] NP[acc]:ppro[1]}]> ]

  NONLOCAL:TO-BIND:SLASH:{[2]} ]
```

The COMPS value would normally include a `for' PP also in order to account for *Charles is easy for Camilla to please*, and so on. The main point to note is that the VP complement has an accusative NP gap, and this NP is classified as a personal pronoun (ppro) -- we shall see why in the discussion of binding theory which follows. The NP in question is abbreviated as NP[acc]:ppro[1] in the latter example, and the index is shared with the index of the subject. Pollard & Sag note that the index must be referential in order to rule out the expletive examples below:

(10.5) * There is easy to please

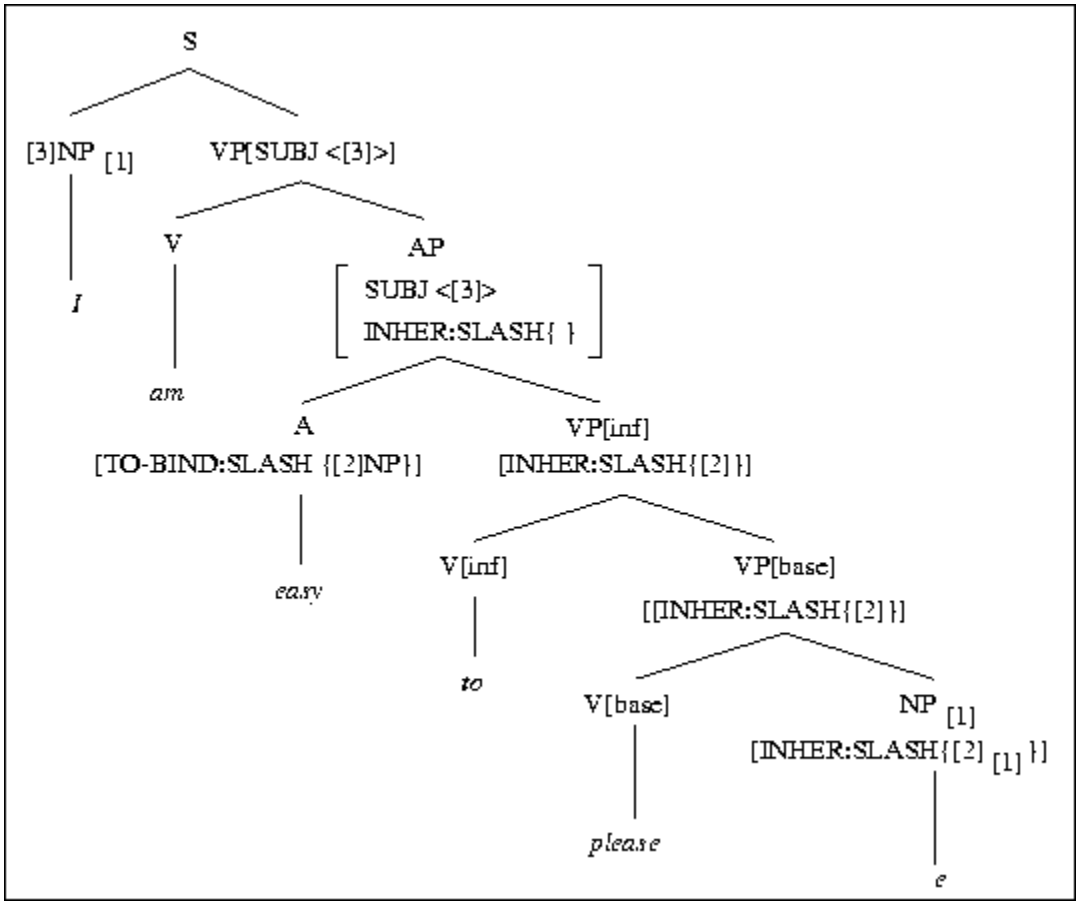
(10.6) * It is easy for Kim to devour (expletive *it*)

Note further that it is the gap which is referential, and so in cases where there is no gap, an expletive subject is still possible:

(10.7) It is easy to please Kim

To emphasise the behaviour of the lexical entry in 10.4, Pollard & Sag provide the following tree structure for example 10.3:

(10.8)



Here, in a similar manner to the way the filler-head schema operates, the value of TO-BIND:SLASH on *easy* ensures that the NP gap is satisfied at the required level. The normal feature-passing conventions ensure that the missing NP is associated with the relevant part of the lexical entry for *easy*, and hence that it is coindexed with the matrix subject. Further examples discussing double extractions (*The play which Kim is easy to talk to [e] about [e]*) can be found in the book.

A couple of final points about the tree in 10.8 should be noted. Firstly, it assumes that there is an empty NP in the structure, and as we have seen in the earlier parts of this course, current HPSG attempts to eliminate these by means of lexical rules -- the complement extraction rule will produce the required form of *please* in this case. Also -- although the book does not discuss this -- it is probable that the lexical entry for *easy* in 10.3 should be produced using a lexical rule from the `base' form which would appear in cases such as *it is easy to please Kim* in which there is no missing object in the VP.

Binding Theory

Chapter 6 in the HPSG book introduces binding theory, much of the discussion being aimed at distinguishing HPSG from GB. This section provides a brief account of the main general points which binding theory attempts to cover, the basic problem being explanations of data such as the following:

(10.9) Sandy$_i$ likes him$_j$

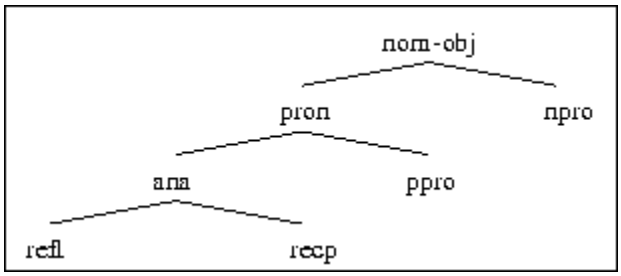
(10.10) * Sandy$_i$ likes him$_i$

(10.11) Sandy$_i$ likes himself$_i$

The subscripts indicate coindexing -- so in 10.10, for instance, it is not possible for the pronoun to refer to *Sandy*. A reflexive, as in 10.11, must be used.

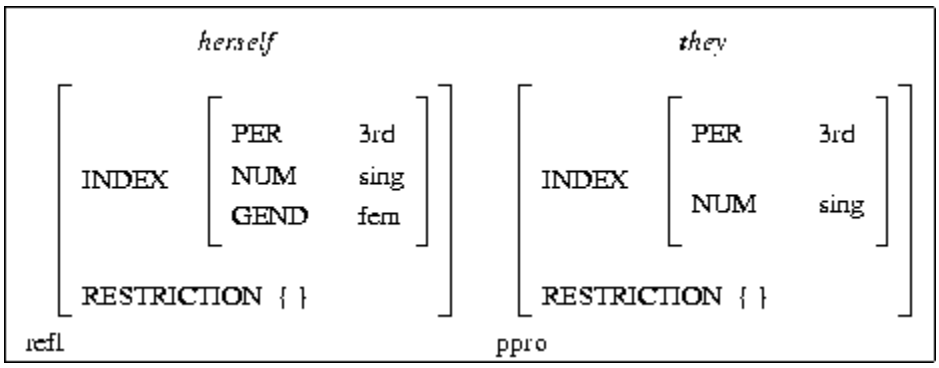
The root of the HPSG account of binding lies in the classification of NPs. This is done by introducing various subtypes to the NOMINAL-OBJECT sort, as shown in the following diagram (p249):

(10.12) Nominal sort hierarchy:



Thus normal referential nouns are of type *npro*, pronouns are either personal (*he*, *they*, and so on) or anaphoric, and anaphoric pronouns are either reflexives (such as *himself*) or reciprocals (*each other*, for example). Pollard & Sag provide the following example lexical entries (p250):

(10.13)



Expletives are assumed to be of type *ppro*, which explains their appearance in tag questions in which only pronominals are possible:

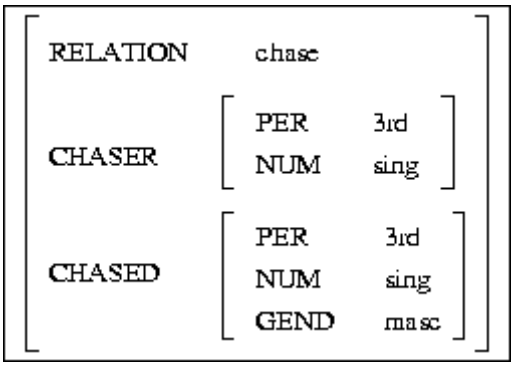
(10.14) Kim will be here, won't he?

(10.15) * The book is finished, isn't the book?

(10.16) There's a problem, isn't there?

(10.17) It's easy to annoy Sandy, isn't it?

To illustrate the core of the binding problem, Pollard & Sag suggest the following representation for the content of the sentence *Fido chased himself*:



What needs to be added to this structure is the information that the `chaser' and the `chased' are coindexed. In the book, this is discussed in terms of the SUBCAT lists, which in the present example would contain the following:

(10.18) [SUBCAT⟨NP:npro, NP:ana⟩]

One of the binding principles, then, is a constraint which says that an anaphoric NP must be coindexed with a less oblique item in the SUBCAT list. The obliqueness hierarchy is simply the left-to-right ordering in the list, so an anaphor must be coindexed with something to its left. This requirement is formally stated in terms of *local obliqueness-command* (*local o-command*) as follows (p253):

(10.19) *local o-command*:

Let Y and Z be *synsem* objects with distinct LOCAL values, Y referential. Ther
locally o-commands Z just in case Y is less oblique than Z.

This statement is not enough to cover cases where one of the coindexed elements is inside a larger constituent in the SUBCAT list -- as would be the case in a sentence such as *Kim^i thinks that Sandy likes him^i*. For these the more general *o-command* statement is necessary (p253):

(10.20) *o-command*:

Let and Z be *synsem* objects with distinct LOCAL values, Y referential. Then
o-commands Z just in case Y locally o-commands X dominating Z.

The final statement required concerns the notion of *o-binding*, which defines what it means for some grammar object to `bind' another (p254):

(10.21) *o-bind*:

Y (*locally*) *o-binds* Z just in case Y and Z are coindexed and Y (locally) o-comma
Z. If Z is not (locally) o-bound, then it is said to be (*locally*) *o-free*.

With these definitions in place, the HPSG binding principles can be stated (p254):

(10.22) *HPSG Binding Theory*:

Principle A. A locally o-commanded anaphor must be locally o-bound.
Principle B. A personal pronoun must be locally o-free.
Principle C. A non-pronoun must be o-free.

The book then works through a few standard binding cases to show how these principles apply. This can be illustrated using the examples in 10.9, 10.10, and 10.11 above, repeated below:

(10.23) Sandy$_i$ likes him$_j$

(10.24) * Sandy$_i$ likes him$_i$

(10.25) Sandy$_i$ likes himself$_i$

Sentences such as 10.25 are acceptable because the anaphor is locally o-bound, as Principle A insists. The SUBCAT list for the verb would be:

(10.26) [SUBCAT⟨NP$_i$, NP:ana$_i$⟩]

The related examples below are ruled out by the same principle, however:

(10.27) * Sandy$_i$ likes himself$_j$

(10.28) * Sandy$_i$ believes Kim likes himself$_i$

In 10.24, the SUBCAT list for the verb would have to be as follows:

(10.29) * [SUBCAT⟨NP$_i$, NP:ppro$_i$⟩]

This is ill-formed because the pronoun *him* is locally o-bound by the subject, which violates Principle B. Cases such as *Sandy$_i$ believes Kim likes him$_i$* are fine, however, because the pronoun is o-bound, but not locally o-bound. Finally, Principle C excludes analyses in which a referential expression (type *npro*) is bound.

Looking again at sentences such as *Kim is easy to please* as discussed in § 10.2, we can now see why the NP gap following *please* is assumed to be a personal pronoun in the lexical entry for *easy* in 10.3. If it were an anaphor, it would have to be bound by the subject of *please*, which is clearly wrong, and if it were referential, it could not be bound by the matrix subject. As a personal pronoun, it satisfies the binding principles.

The book discusses many other facts, and problems, concerning binding theory -- again, often with a view to contrasting HPSG and GB. There is also some discussion of how the analysis can be maintained when the SUBCAT feature is split into the three valency features in chapter 9 (p375, p379n, p401).

ALE Implementation

The analysis of tough-movement expressions suggested in § 10.2 can be implemented fairly directly in ALE. Lexical entries such as that proposed for *easy* in 10.3 have obvious ALE counterparts, although there are no examples in the final grammar (`grammar10.pl`) at the moment.

The situation with the binding constraints is different. As with some of the other HPSG principles -- the nonlocal feature principle being a good example -- some amount of prolog code would be necessary to effect the necessary checks. The obvious place to apply the principles is to the mother category in rules -- in each case, the SUBCAT list on the mother (sometimes called the `argument' list in recent HPSG) could be checked to ensure that the principles are being satisfied.

However, just checking that the principles are not being violated is not actually enough -- some mechanism is necessary in order to effect coindexing in the first place. In all the examples of structure sharing that we have seen, either of full categories or indices, the matching has been imposed by the schemas (and associated principles) and/or lexical entries. In the case of binding, some mechanism would be necessary in order to produce the two analyses below, for instance:

(10.30) Kim put the book$_i$ on the table to protect it$_i$

(10.31) Kim put the book on the table$_i$ to protect it$_i$

There is nothing in HPSG, and certainly not in the ALE grammars we have seen, which will do the actual coindexing. It may be that this can be left to an independent discourse model, but however it is done, there is clearly no point in implementing the binding principles until the bindings are actually in place.

Conclusion

This completes the HPSG/ALE course. The 10 parts have covered most of the central concerns in the HPSG book, and have also covered most of the functional aspects of the HPSG grammar which accompanies ALE. The latter grammar is based on the analysis presented in the earlier chapters in the book, so the grammar developed in the course is effectively a revised version which takes into account some of the points made in chapter 9. The final grammar (grammar10.pl) is a copy of one written by Suresh Manandhar and Claire Grover for a course in advanced grammar writing, and it implements analyses of all the main linguistic issues covered in the current course apart from the material discussed in this final part.