Implementing Typed Feature Structure Grammars

Ann Copestake

ISBN: 1-57586-260-3

Contents

2

A first session with the LKB system

The following chapter takes the new user through an initial session with the LKB system. It covers the basics of:

1. Obtaining and starting the LKB
2. Using the LKB top menu
3. Loading an existing grammar
4. Examining typed feature structures and type constraints
5. Parsing sentences
6. Viewing a semantic representation
7. Generating from parse results
8. Adding a lexical entry
9. Adding a type with a constraint description

If you have no previous exposure to typed feature structure formalisms you will find that you don't fully understand all the terminology and notation used here. In the subsequent chapters, I will go through a sequence of grammars, starting with a very simple one and finishing off with the one that is illustrated in this chapter, explaining all the details of the formalism, so that you end up with a full understanding of how everything works. The idea is that the very detailed information will be easier to digest after this quick guided tour has given you an intuitive idea of where we are going.

2.1 Obtaining and starting the LKB

The instructions in this section outline how to get an executable version of the LKB for Windows, Linux or Solaris. Because the details may change slightly, you should also refer to the instructions for downloading on the LKB website:

`http://cslipublications.stanford.edu/lkb.html`

In case of any conflicts, follow the instructions given there rather than the ones in this book.

2.1.1 System requirements

Windows systems

1. Windows 95, 98, NT or 2000. The system may run on other versions of Windows but this has not been tested at the time of writing (check the website for updates).
2. At least 128 Megabytes of memory.
3. 30 Megabytes of free disk space.
4. WinZip or PowerArchiver for extraction of the downloaded files.

Linux

1. The system has been extensively used with Red Hat Linux (6.0 and later). Other versions should also work but have not been tested (check the website for updates).
2. At least 128 Megabytes of memory.
3. 30 Megabytes of disk space.
4. A suitable version of Motif: currently the LKB works with Metro Link Incorporated's Metro Motif 1.2.4 and OpenMotif. The LKB does NOT work with lesstif at the time of writing.
5. gzip and tar are needed to extract the files.

Solaris Solaris requirements are similar to Linux, but Motif is generally already installed on Solaris systems. OpenMotif is not available for Solaris.

emacs We recommend the use of emacs (either gnuemacs or XEmacs) with the LKB for editing the grammar files but this is not essential, and it is not needed for this chapter. Instructions for obtaining emacs and setting it up with the LKB are given on the website.

2.1.2 Downloads

You will need to download two archives of files from the LKB website:

`http://cslipublications.stanford.edu/lkb.html`

One is the executable version of the LKB for whichever platform you intend to use, the other is a collection of example grammars.

Before downloading anything, I suggest that you make a new directory/folder[9] for all the LKB files. I will refer to this as your LKB directory.

[9] A *directory* is the equivalent in the Linux/Unix world to a *folder* in Windows/Macintosh terminology. I will use the term directory rather than folder throughout the rest of this book. I will also generally use the Linux/Unix notation,

Use your browser to locate the relevant version of the LKB from the LKB website and save the file to your LKB directory. You will then have to extract the compressed files: on Linux or Solaris you can use `gzip` followed by `tar xf` while on Windows, WinZip or PowerArchiver should uncompress and extract as one operation.

You can then download and extract the data files into the same directory. This should result in a directory `data`, with a number of subdirectories, including `itfs`, which is the directory for all the grammars we will use in this book.

Instructions for building the LKB from source files are on the website.

2.1.3 Starting the LKB

Before you start the LKB for the first time, you must create a temporary directory/folder. On Linux or Solaris, this should be a directory called `tmp` in your home directory. On Windows, create the empty folder `C:\tmp`. If necessary, the location of this directory can be varied, details of how to do this are given on the LKB website.

To start the LKB on Windows, simply double click on `lkb.exe`. On Linux or Solaris, cd to your lkb directory and type `lkb` at the command line. If you have successfully started the LKB, you should see the LKB top menu window, as described in the next section. You will also see a command line, with prompts such as `LKB(1):` — this allows the user to type in commands but can be ignored for the purposes of this chapter.

Warning note: do not use the NumLock key when using the LKB system with Linux. There is a bug in the software on which the LKB is built which causes menus to stop working intermittently when the NumLock key is on. Once this has happened, restarting the LKB itself will not help, you have to log out and restart your X session.

2.1.4 Installation problems

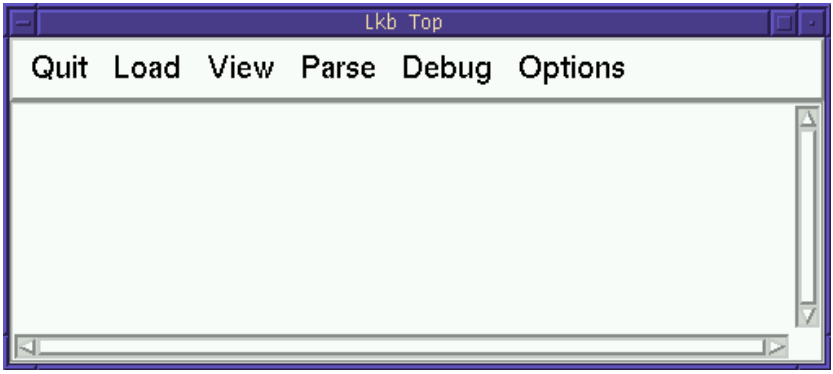
In case of installation or other problems which don't seem to be covered in the documentation, please look at the instructions on the LKB website. We will make fixes for known problems available there. The website also contains details of how to report any bugs that you find.

2.2 Using the LKB top menu

.

The main way of interacting with the LKB is through the LKB top menu window which is displayed once you have started the LKB, as shown below.

with forward slash (`/`) separating directories, rather than the Windows notation, but the equivalence should be obvious.



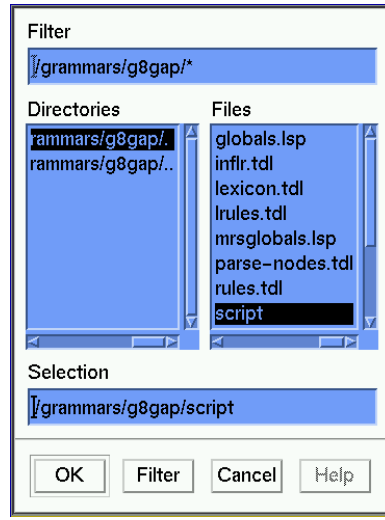
This is a general purpose top level interaction window, with the menu displayed across the top — most LKB system messages appear in the pane below the menu buttons.[10] I will use the term *LKB interaction window* for the window in which messages appear. Most of the menu commands are not available when the LKB is first started up, because no grammar has been loaded.

2.3 Loading an existing grammar

The first step in this guided tour is to load an existing grammar: i.e., a set of files containing types and constraints, lexical entries, grammar rules, lexical rules, morphological rules and ancillary information. The LKB comes supplied with a series of grammars: the ones we will use in this book are all in the directory `data/itfs/grammars`. In this section, we will assume that you are working with the grammar in `g8gap`.

To load a grammar it is necessary to select a *script* file which controls how the grammar files are loaded into the system. Select **Complete grammar** from the LKB **Load** menu, and choose the file `script` from the `g8gap` directory as shown below.

[10]The Macintosh version of the LKB has a slightly different user interface which I will not describe in this book: for a brief discussion of the main differences, see the LKB web page.



You should see various messages appearing in the interaction window, as shown in Figure 1 (the interaction window has been enlarged). If there are any errors in the grammar which the system can detect at this point, error messages will be displayed in this window. With this grammar, there should be no errors, unless there is a problem associated with the temporary directory (see §2.1). If you get an error message when trying to load `g8gap/script`, it is possible you have selected another file instead of `script` — try again.[11]

Once a file is successfully loaded, the menu commands are all available and a *type hierarchy* window is displayed (as shown below). You can enlarge this window to show the complete hierarchy or scroll it in the usual way.

[11] In case of genuine problems, please see the LKB webpage section on known bugs and bug reporting.

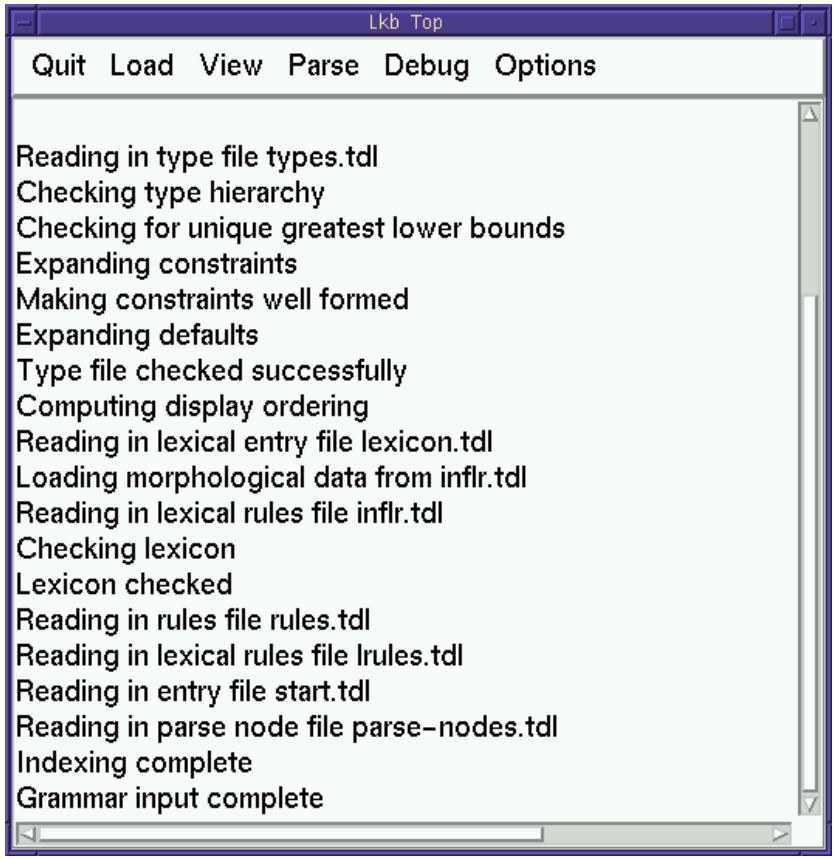
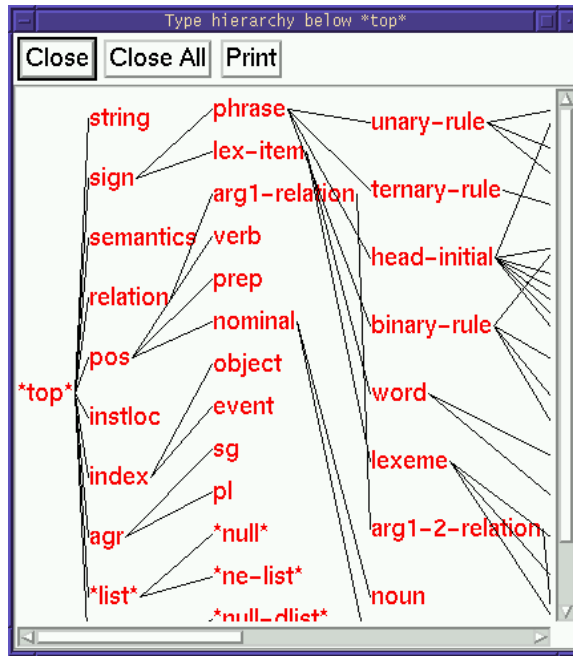


FIGURE 1 Loading a grammar



2.4 Examining typed feature structures and type constraints

In this section I will go through some of the ways in which you can look at the data structures in the grammar, such as the types, type constraints, lexical entries and grammar rules.

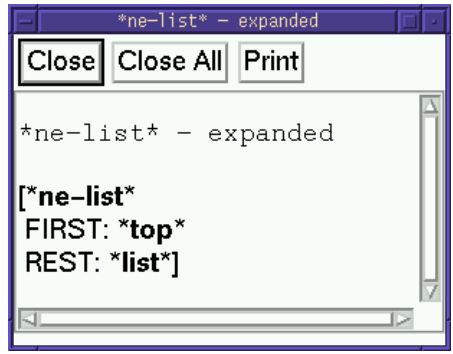
2.4.1 The type hierarchy window

The backbone of any grammar in the LKB is the type system, which consists of a hierarchy of types, each of which has a *constraint* which is expressed as a *typed feature structure*. Constraints are used to capture generalizations: the type hierarchy allows for inheritance of constraints. In the LKB system, the type hierarchy window is shown with the most general type displayed at the left of the window.[12] In this grammar, as in all the grammars I will discuss, the most general type is called ***top***. You will notice that there is some *multiple inheritance* in the hierarchy

[12]In the book, except where showing a screen dump, I will show type hierarchies with most general type towards the top of the page: the reason for the alternative orientation in the LKB itself is that this is a more efficient layout for typical hierarchies, which tend to be broad rather than deep.

(i.e., some types have more than one parent). You will see a few types with names such as **glbtype1**: these are types which are automatically created by the system, for reasons which will be explained in the next chapter.

Click on the type ***ne-list*** which is a daughter of ***list***, which is a daughter of ***top***, and choose **Expanded Type** from the menu. A window will appear as shown below.



This window shows the constraint on type ***ne-list***: the constraint is expressed as a typed feature structure (TFS). It has two *features*, FIRST and REST. The value of FIRST is ***top***, which indicates it can unify with any TFS since ***top*** is the most general type. The value of REST is ***list*** which indicates it can only unify with something which is of type ***list*** or one of its subtypes. ***list*** and its daughters are important because they are used to implement list structures which are found in several places in the grammar. In this book, and in the LKB system windows, types are shown in lowercase, bold font, while features are shown in uppercase (small capitals in the book).

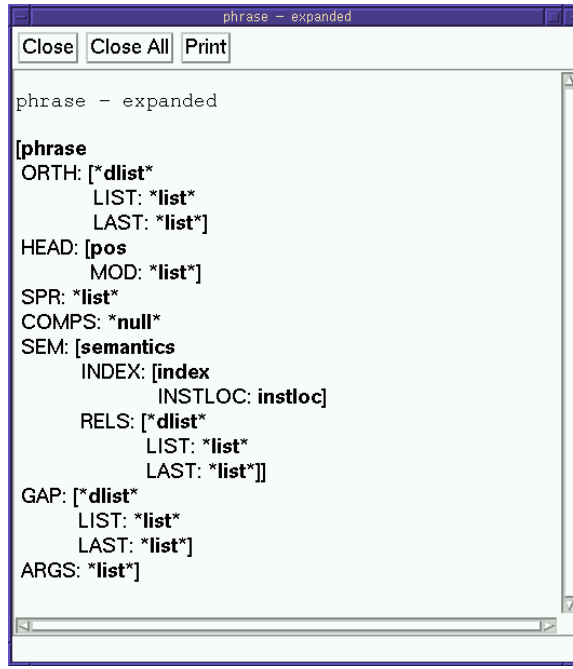
Look at the entry for the type ***ne-list*** in the actual source file `g8gap/types.tdl`, by opening that file in your editor (if you are using Windows, and have not installed emacs, you should use Notepad). If you search for `*ne-list*`, you will see the following definition:

```
*ne-list* := *list* &
 [ FIRST *top*,
   REST *list* ].
```

The language in which the type and its constraint are defined in the files is referred to as a *description language*. The type definition must specify the parent or parents of a type (in this case, ***list***) and optionally gives a constraint definition. In this particular case, the constraint described

in the file corresponds very closely to the expanded constraint shown in the typed feature structure window, because the only parent of ***ne-list*** is ***list*** and this does not have any features in its constraint. However, in general, type constraints inherit a lot of information from the ancestors of the type, so the description of a constraint is usually very compact compared to the expanded constraint.

To see a more complicated type constraint, click on **phrase** in the type hierarchy window (found via **sign** from ***top***) and again choose **Expanded Type**. The TFS window is shown below:



This illustrates that types can have complex constraints — the value of a feature in a constraint can be a TFS.

Look at the definition of **phrase** in the source file `g8gap/types.tdl`:

```
phrase := sign &
 [ COMPS < > ].
```

The notation `< >` is an abbreviation for ***null***, which represents a list with no elements. In contrast to ***ne-list***, the expanded constraint on **phrase** has inherited a lot of information from other types in the hierarchy: you can get an idea of how this inheritance operates by looking at

the constraint of **phrase**'s parent (**sign**) in the type hierarchy window. Note that **sign** has a feature SEM which has the value **semantics**: some of the information in the expanded constraint on **phrase** comes from the constraint on **semantics**.

You will find that you can click on the types within the TFSs to get menus and also on the description at the top of the window (e.g., `phrase - expanded`). I won't go through all these menu items here, however, but they are discussed in Chapter 6.

2.4.2 The View commands

The view commands let you see objects such as *lexical entries* which are not types and therefore cannot be accessed from the type hierarchy window.

Select **View** from the LKB top menu and then select **Word entries**. You will be prompted for a word which corresponds to a lexical entry. Enter `dog` (case doesn't matter), deleting the default that is specified (unless of course the default is `dog`, in which case just select `OK`). You should get a TFS window corresponding to the entry which has orthography `"dog"` in the `g8gap` grammar as shown in Figure 2. If there were multiple entries with the spelling `"dog"` they would all be displayed.

You should compare the TFS shown in the window with the lexical description in the file `g8gap/lexicon.tdl`, to see how the inheritance from type constraints operates. The lexical description for *dog* is simply:

```
dog := noun-lxm &
[ ORTH.LIST.FIRST "dog",
  SEM.RELS.LIST.FIRST.PRED "dog_rel" ].
```

Nearly all the detail in the full TFS comes from the type **noun-lxm**.

Now try **View Grammar rule** and enter `head-specifier-rule` (or choose it from the selections if a menu is displayed). You will see that the grammar rule is also a TFS, which is shown in Figure 3. I do not reproduce it in full here, because it will not fit on one page, but the boxes indicate which parts of the structure have been 'shrunk' (this can be done by clicking on a node in a TFS window, and choosing the menu option **Shrink/expand**). A TFS that encodes a rule can be thought of as consisting of a number of 'slots', into which the phrases for the daughters and the mother fit. In this grammar, as in all the others we will look at in this book, the mother is the TFS as a whole, while the daughters are the elements in the list which is the value of the ARGS feature.

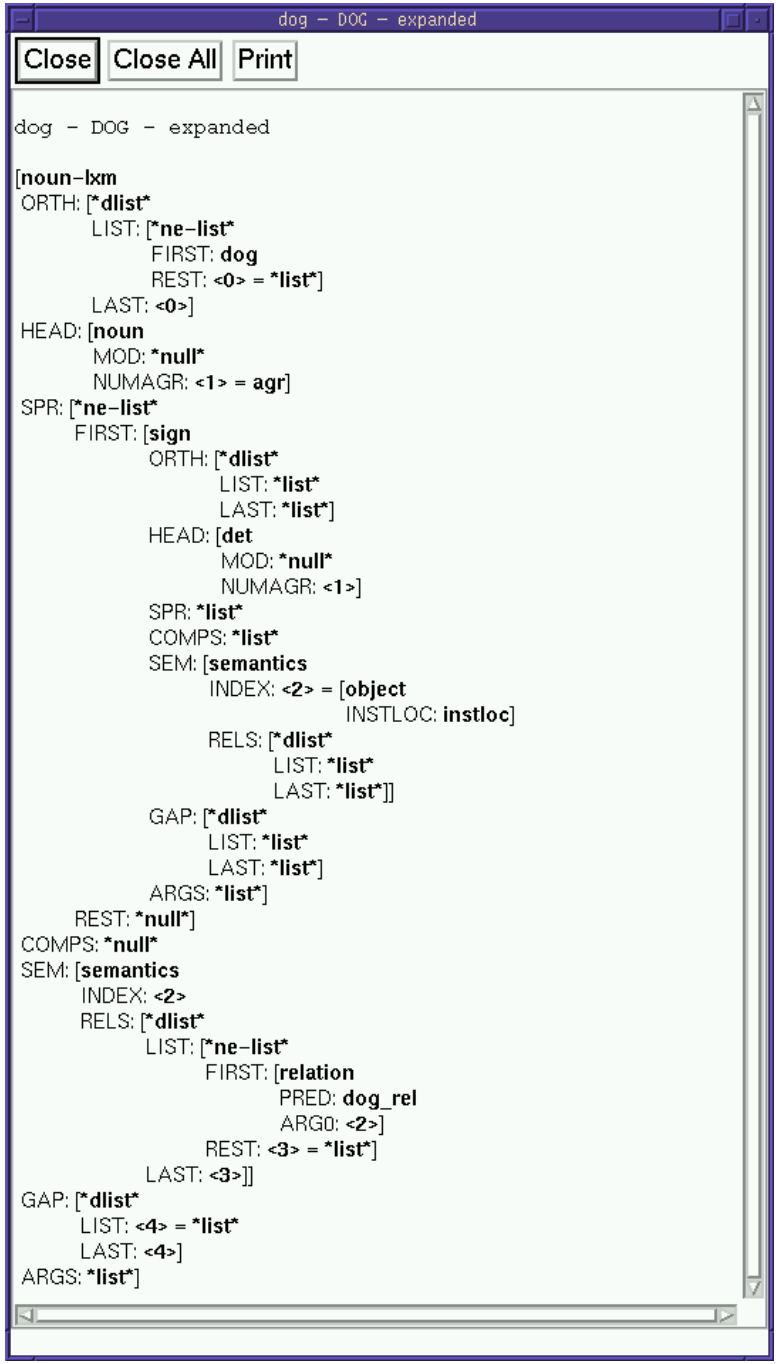


FIGURE 2. Expanded lexical entry for *dog*

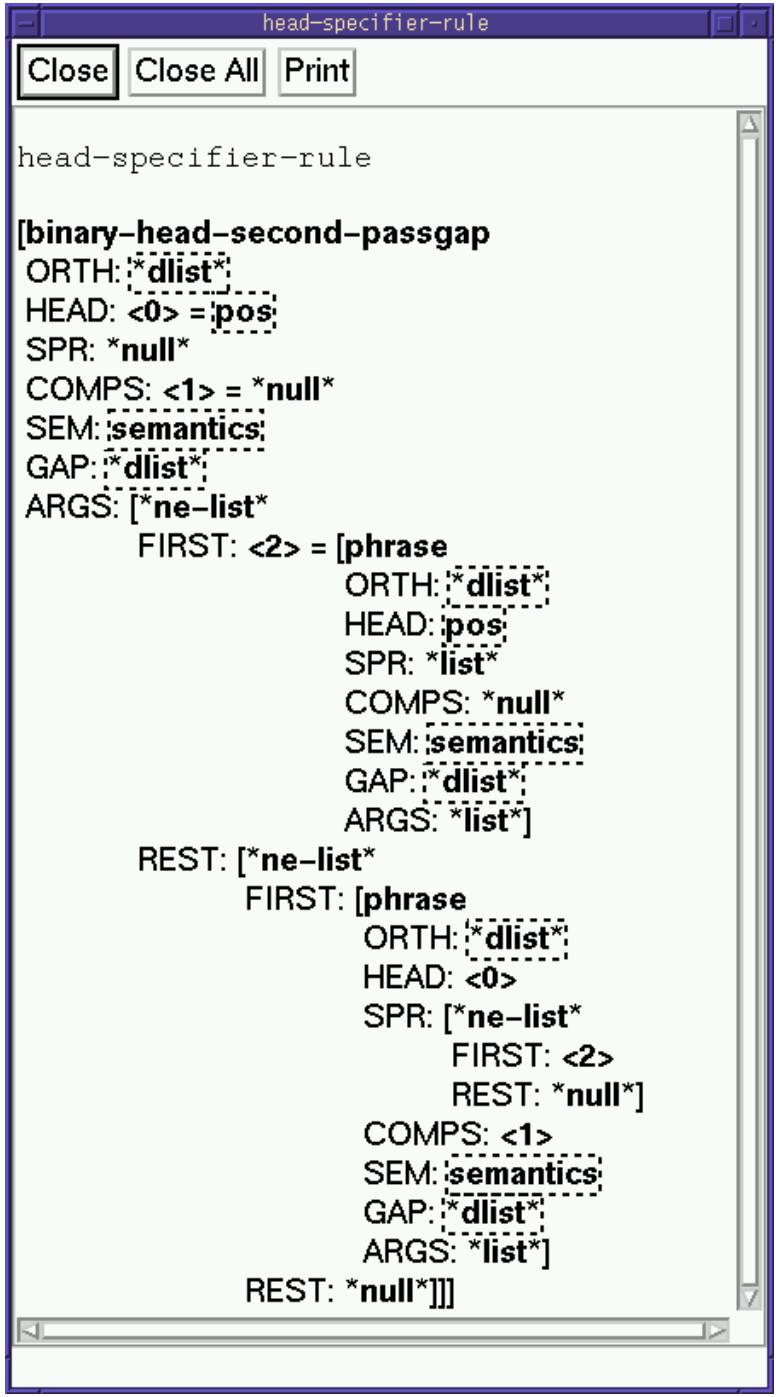
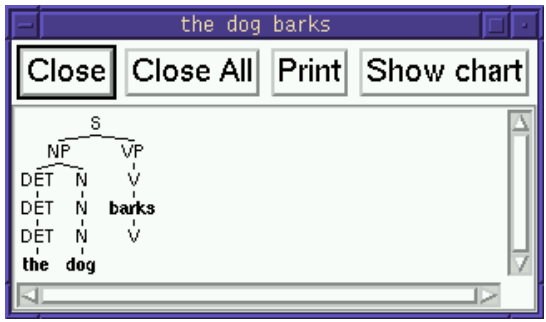


FIGURE 3 The head specifier rule

2.5 Parsing sentences

To parse a sentence, click on **Parse** / **Parse input**. A suitable sentence to enter is `the dog barks`. Click `OK` to start parsing. You will get a window with one tiny parse tree, as shown below.[13]

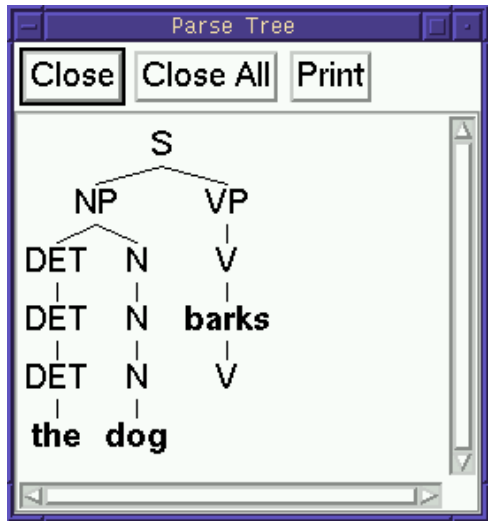


Although this grammar illustrates some quite complicated linguistic phenomena, it has a really tiny lexicon, so you can't type in arbitrary sentences and expect them to parse. To get an idea of what the grammar will parse, look at the `test.items` file.

2.5.1 Parse trees

If you click on the tiny parse tree in the window that shows the parse results, you will get a menu with an option **Show enlarged tree**. If you choose this, you will see a window with a more readable version of the tree, as shown below.

[13]The reason that the interface uses such a small size is that with non-trivial grammars and sentences, the parse trees can be very large and numerous, so this display is designed to allow a succinct overview.



In the LKB system, a parse tree is just a convenient user interface device, which is shorthand for a much larger TFS. Click on the uppermost (S) node of the enlarged parse tree and choose the option **Feature structure — Edge 11**. You will see a large TFS, which is shown in Figure 4. As before, I have 'shrunk' some parts of the structure so that it can be displayed on the page. This structure represents the entire sentence. It is actually an instantiation of the `head-specifier-rule` shown in Figure 3.

The top node in the parse tree corresponds to the root node of the TFS shown in Figure 4. The structure for the phrase *the dog* is the node which is the value of the path ARGS.FIRST (a *path* is a sequence of features). The structure for the verb *barks* is the value of the path ARGS.FIRST.REST. The parse trees are created from the TFSs by matching these substructures against a set of node specifiers defined in the file `parse-nodes.tdl`. I will go into a lot more detail about how grammar rules work in the next chapters.

2.5.2 Morphological and lexical rules

You will notice that the parse tree has two nodes labelled V, one above and one below **barks**. They represent the application of a morphological rule: the lexicon contains an entry with the spelling `"bark"`, and the rule for third person singular verbs generates the inflected form `"barks"` from the lexical form. Morphological rules are used for inflectional and derivational processes which are associated with affixation: lexical rules

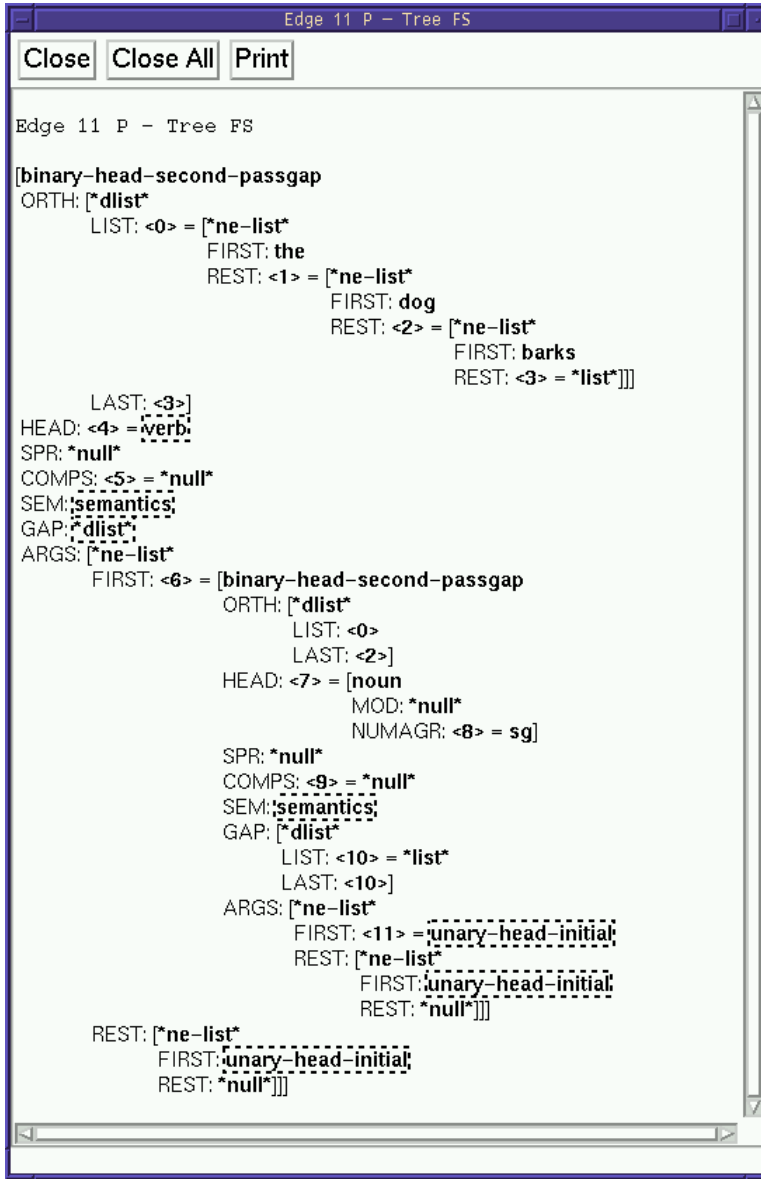
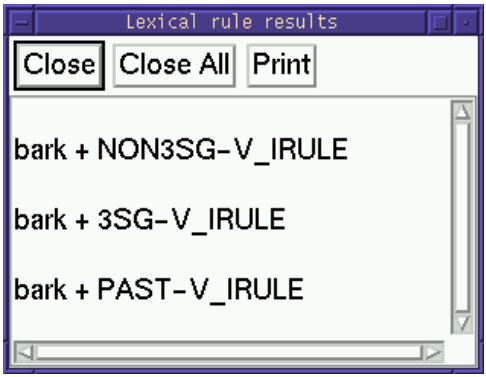


FIGURE 4 A TFS representing the sentence *the dog barks*

are used where there is no affixation. Morphological and lexical rules are very similar to ordinary grammar rules in the LKB system.

Try viewing the lexical entry for *bark* via **View / Lex entry**. This behaves much like **View / Word entries** that you used before, because in this small grammar, the identifiers for the lexical entries are the same as the orthography. Hence when prompted for a Lex-id, you can just enter `bark`. At the top left of the window, it will say `bark - expanded`. If you click on this, you will get a menu, which among other things has the option **Apply all lex rules**. If you select this, you will get a window which shows which lexical rules apply to *bark*: you can click on the nodes in the result window to display the feature structures corresponding to the inflected forms.



2.5.3 Batch parsing

Now try **Parse / Batch Parse**. You will be prompted for the name of a file which contains a test suite: i.e., a list of sentences which either should or should not parse in a grammar. A suitable file, `test.items`, already exists in the `g8gap` directory. Select `test.items` and enter the name of a new file for the output, e.g., `test.items.out`. The system will now parse all the sentences in `test.items` (this should only take a few seconds, unless you are using a very slow machine or one with very little memory). When you open the output file in your editor, it will show the following for each sentence:

1. the number of the sentence
2. the sentence itself
3. the number of parses (0 if there were no parses)
4. the number of passive edges (roughly speaking a *passive edge* represents a phrase that the system constructed while attempting to

parse a sentence)

For instance:

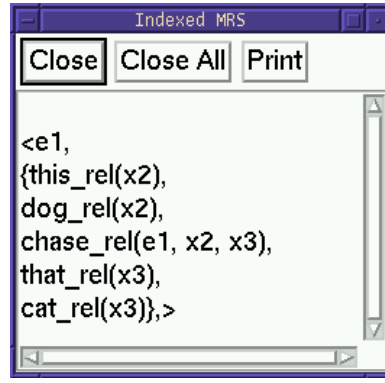
```
1 The dog barks. 1 11
2 *The dog bark. 0 10
```

Note that ungrammatical sentences are marked with an asterisk in the test suite file, and though a preprocessor strips off the asterisk and any punctutation symbols before attempting to parse, the results file shows the sentence in the form in which it was input. At the end of the results file, the total parsing time for all the sentences in the file is reported: of course, this time will depend on what sort of machine you have.

Have a look at the sentences in `test.items` to get some idea of the coverage of the grammar. You should try parsing some of these sentences individually and looking at the trees that result.

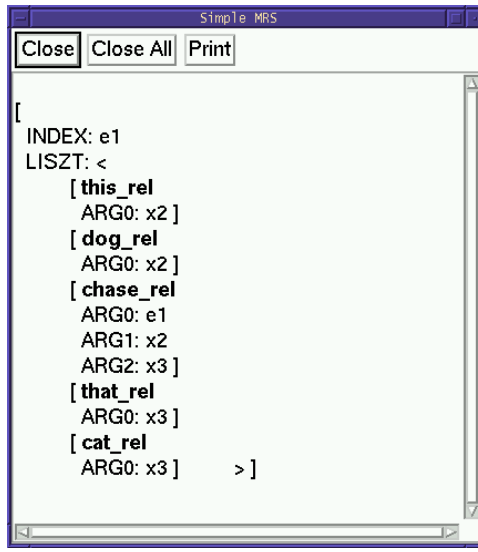
2.6 Viewing a semantic representation

Try parsing *this dog chased that cat*, and choosing **Indexed MRS** from the menu that you get by clicking on the small parse tree. This will give you a representation for the semantics of the sentence. (MRS is a semantic representation language that can be converted into more familiar languages such as predicate calculus, as is explained in more detail in later chapters.)



The actual semantics is constructed as a TFS, and can be seen as the value of the feature SEMANTICS in the TFS representing the parse for the sentence. The representation that is displayed when you select **Indexed MRS** is much easier to read, although some information has been omitted. A representation which is closer to the TFS structures can be obtained by choosing **MRS** instead of **I**ndexed MRS from the

menu associated with the small parse tree.



Semantic structures like this can be conveniently passed to other programs which use the results of the parse. One of the other options on the menu is **Prolog MRS**, which illustrates a format suitable for input to a Prolog system. (The **Scoped MRS** option which is also on the menu is not useful with this particular grammar, because it requires a fuller representation of quantifiers.)

2.7 Generating from parse results

The availability of semantics with this grammar allows us to try generating sentences. For convenience, this can be done simply by clicking on one of the small parse trees that results from parsing a sentence and choosing **Generate**. However, the actual input to the generator is the MRS representation, as shown in the windows you have just seen. Try generating from the result of parsing *this dog chases that cat*. The sentences generated are displayed in a window, as shown below.

You will notice that you get back four sentences. One is the input sentence and another is the present tense version of that sentence. The reason for getting both is that the semantic representation in this particular grammar does not include a representation of tense. You will also see two *topicalized* sentences: *that cat this dog chased* and *that cat this dog chases*. Topicalization may seem weird, but linguists generally assume that grammars should produce such sentences and in fact they are perfectly acceptable in some contexts.

If you click on one of the generated sentences, you will see a menu which has the options **Edge** and **Feature structure**. If you click **Edge**, you will see a tree for the sentence corresponding to the parse tree which would be produced if the sentence were parsed, although without any nodes corresponding to the inflectional rules.

Try parsing an ambiguous sentence, such as *this dog chased that cat near the aardvark* and compare the semantics for the different trees. Then generate from each tree. You should observe that you obtain a slightly different set of sentences. Some of these sentences will seem very weird, even if you have become used to topicalization! This grammar actually accepts/generates some ungrammatical sentences: this is what is formally referred to as *overgeneration*.

2.8 Adding a lexical entry

In this section, I will describe how to add a new lexical entry. You may want to make a backup copy of the `g8gap` directory before you start editing the files. First open the file `g8gap/lexicon.tdl` in your text editor. You will see that the grammar has three lexical entries for nouns — *cat*, *dog* and *aardvark*. Suppose you want to add another noun entry, perhaps for *rabbit*. This will look exactly like the entry for *dog*,

but with the value of the orthography path ORTH.LIST.FIRST replaced by `"rabbit"` and the value of the semantic path SEM.RELS.LIST.FIRST.PRED replaced with a suitable predicate, which I will call `"rabbit_rel"`. The "s round the orthography and semantic predicate values tell the system that these are not 'proper' types: they are *string types* which do not need to be declared. `_rel`, on the other hand, is just a naming convention: the semantic predicate could equally well be `"rabbit'"` or `"gavagai"`. Add a new lexical entry to the lexicon file by copying and pasting the entry for *dog* and changing `dog` to `rabbit`, as shown below:

```
rabbit := noun-lxm &
[ ORTH.LIST.FIRST "rabbit",
  SEM.RELS.LIST.FIRST.PRED "rabbit_rel" ].
```

Save the file, and then select **Load/ Reload grammar**. This will reload the script file that you loaded before, this time loading the changed version of the lexicon. You may get some error messages in the LKB interaction window at this point, if you have left out a character from the new entry, for example. If you cannot see what is wrong, the description of the error messages in §7.1.1 may help you track it down. When you have successfully reloaded the system, you should be able to parse some additional sentences, such as:

```
the rabbit chased the dog
```

Once you have successfully parsed some new sentences, you could add them to the list of test sentences for batch parsing (i.e., `test.items`).

2.9 Adding a type with a constraint description

Substantial changes to the grammar always involve changing types. I will illustate this with a very simple example. Consider nouns like *scissors*, *binoculars* and *trousers*, which in most dialects of English always take plural agreement. I'll refer to these as *pair nouns*. We don't want to give such nouns a normal entry as a **noun-lxm**, because they would end up behaving as standard nouns and having both singular and plural forms. One very simple way to get approximately the correct behaviour with this rather simple grammar is to make a new type which specifies that the number agreement has to be plural. The instructions below walk you through the process of creating the type: you are not expected to understand exactly what is going on at this point, but just to get some idea of how grammars may be modified.

Open the file `g8gap/types.tdl`. Search the file for the definition of **noun-lxm**. Then add a new definition for **pair-noun-lxm** which should inherit from **noun-lxm**, but specify that the value for HEAD.NUMAGR

is **pl**. The type definition you need to add is:

```
pair-noun-lxm := noun-lxm &
[ HEAD [ NUMAGR pl ]].
```

This specification can go anywhere in the file, though putting it under the definition of **noun-lxm** will improve readability,

You can then save the `types.tdl` file and check that you can load the revised grammar with **Reload grammar**. You should be able to see the new type in the type hierarchy and view its constraint. However, in order to demonstrate that the new type works as designed, we have to add a new entry to the lexicon file that uses it. As in the last section, you can do this by copying an existing entry, but this time you have to change the type to **pair-noun-lxm** as well as changing the orthography, the semantics and the identifier. For the sake of the example, we will enter the orthography as `"scissor"`, which will get the correct form when it has gone through the rule for plural noun inflection.[14]

```
scissor := pair-noun-lxm &
[ ORTH.LIST.FIRST "scissor",
  SEM.RELS.LIST.FIRST.PRED "scissor_rel" ].
```

Save the file and select **Reload grammar**. As before, check the LKB interaction window to make sure reloading was successful. Try parsing some new sentences. You should find that you cannot parse *the dog chased this scissor* but you can parse *the dog chased the scissors*.

Try generating from the result of parsing *the dog chased the scissors*. Note that the sentence with singular *scissor* is not generated. While accepting ungrammatical sentences is not necessarily too problematic for applications which are only intended to analyse sentences, we certainly don't want to generate them. One of the fundamental principles behind systems such as the LKB is that grammars can be *bidirectional*: that is, they can be used for both parsing and generation.

2.10 Summary

This tour has only touched on some of the features of the LKB system — there are several menu options which have not been described (these are all listed in Chapter 6). You should try playing around with the grammar, parsing and generating some more sentences, looking at how the TFSs are built up and adding a few more lexical entries that use the types already in the grammar.

The main aim of this chapter was to give you a rough idea of what

[14] This isn't unreasonable: a stem form *scissor* has to exist for some compound nouns, such as *scissor kick*, though we don't deal with compounds in this grammar.

can be done with a typed feature structure system and a simple grammar. Concepts such as types, typed feature structures, lexical entries, grammar rules, parsing, generation and semantics were introduced very briefly and informally. The next three chapters discuss all of this is full detail. Although the grammar in `g8gap` is very small, it does illustrate the main aspects of grammar engineering with typed feature structures and can be used as a basis for understanding or writing much larger grammars.