LKB User Manual

(excerpted from Implementing Typed Feature Structure Grammars)

Ann Copestake

ISBN: 1-57586-260-3

6

LKB user interface

In this chapter, I will go through the details of the menu commands and other aspects of the graphical user interface. The default LKB top menu window has six main menus: Quit, Load, View, Parse, Debug and Options. If you select **Options / Expand menu**, you will obtain a menu which has nine main menus: Quit, Load, View, Parse, MRS, Generate, Debug, Advanced and Options. The expanded menu also makes more submenus available, and makes minor changes to one or two of the basic submenus. You can revert to the basic LKB top menu window with **Options / Shrink menu**.

The first section in this chapter describes the commands available from the LKB top menu, while subsequent sections describe the windows which display different classes of LKB data structure, which have their own associated commands. Specifically, the sections are as follows:

1. Top level commands. Describes the commands associated with the menus and submenus in the order in which they appear in the expanded LKB top menu window. Items which are only in the expanded menu are marked by $*$.
2. Type hierarchy display
3. Typed feature structure display
4. Parse output display
5. Parse tree display
6. Chart display

All the data structure windows have three buttons in common: **Close**, **Close all** and **Print**. **Close** will close the individual window, **Close all** will close all windows of that class — e.g., all type hierarchy windows etc. **Print** produces a dialog that allows a Postscript file to be output which can then be printed. Printing directly to a printer is not implemented yet.

Most commands that produce some output do so by displaying a new window. A few commands output short messages to the LKB interaction window. A small number of less frequently used commands send output to the standard Lisp output, which is generally the emacs `*common-lisp*` or `*shell*` buffer, if the LKB is being run from emacs, and the window from which the LKB was started, if emacs is not being used. These commands are all ones from which a large amount of text may be produced and the reason for outputting the text to an emacs buffer is that the results can be searched (it is also considerably faster than generating a new window).

Very occasionally it is useful to be able to run a command from the Lisp command line (i.e., the window where prompts such as `LKB(1):` appear, which will be the same window as the one displaying the standard Lisp output). This is easier to do using emacs, since commands can be edited.

Because the LKB system is under active development, some minor changes may be made to the commands described here and additional functionality will probably appear. Documentation for any major modifications will be available from the website.

6.1 Top level commands

The top level command window is displayed when the LKB is started up. In this section the LKB menu commands will be briefly described in the order in which they appear in the expanded interface. Commands which are only in the expanded menu are indicated by $*$. To switch between versions, use the **Shrink menu** or **Expand menu** commands under **Options**.

6.1.1 Quit

Prompts the user to check if they really do want to quit. If so, it shuts down the LKB.

6.1.2 Load

The commands allow the loading of a script file (see §4.5.1 and §8.2) and the reloading of the same file (normally this would be done after some editing). A script is used initially to load a set of files, and can be reloaded as necessary after editing.

Complete grammar This prompts for a script file to load, and then loads the grammar. Messages appear in the LKB interaction window.

Reload grammar This reloads the last loaded script file. Messages appear in the LKB interaction window.

Other reload commands∗ The other reload commands are for reloading parts of the grammar — they should not be used by the inexperienced, since under some conditions they will not give the correct behaviour. If something unexpected happens after using one of these commands, always reload the complete grammar.

6.1.3 View

These commands all concern the display of various entities in the grammar. Many of these commands prompt for the name of a type or entry. If there are a relatively small number of possibilities, these will be displayed in a menu.

Type hierarchy Displays a type hierarchy window.

Prompts for the highest node to be displayed. If the type hierarchy under this node is very large, the system will double-check that you want to continue (generally, large hierarchies won't be very readable, so it's probably not worth the wait). The check box allows 'invisible' types, such as glbtypes, to be displayed if set. Details of the type hierarchy window are in §6.2.

Type definition Shows the definition of a type constraint plus the type's parents.

Prompts for the name of a type. If the type hierarchy window is displayed, scrolls the type hierarchy window so that the chosen type is centered and highlighted. Displays the type's parents and the constraint specification in a TFS window: details of TFS windows are in §6.3.

Expanded type Shows the fully expanded constraint of a type.

Prompts for the name of a type. If the type hierarchy window is displayed, scrolls the type hierarchy window so that the chosen type is centered and highlighted. Displays the type's parents and the full constraint on the type.

Lex entry The expanded TFS associated with a lexical entry (or parse node label or start structure etc). The command is used for entries other than lexical entries to avoid having a very long menu.

Prompts for the identifier of a lexical entry (or a parse node label or start structure). Displays the associated TFS.

Word entries All the expanded TFSs associated with a particular orthographic form.

Prompts for a word stem. Displays the TFSs corresponding to lexical entries which have this stem.

Grammar rule Displays a grammar rule.

Prompts for the name of a grammar rule (if there are sufficiently few rules, they are displayed in a menu from which the name can be chosen), displays it in a TFS window.

Lexical rule Displays a lexical or morphological rule.

Prompts for the name of a lexical rule (if there are sufficiently few rules, they are displayed in a menu from which the name can be chosen), displays its TFS in a window.

All words∗ Displays a list of all the words defined in the lexicon, in the emacs `*common-lisp*` buffer (if emacs is being used), otherwise in the window from which the LKB was launched.

6.1.4 Parse

Parse input This command prompts the user for a sentence (or any string), and calls the parser (tokenizing the input according to the user-defined function `preprocess-sentence-string`, see §9.3). A valid parse is defined as a structure which spans the entire input and which will unify with the TFS(s) identified by the value of the parameter `*start-symbol*`, if specified (i.e., the start structure(s), see §4.5.6 and §4.2). (Note that `*start-symbol*` may be set interactively.) If there is a valid parse, a single window with the parse tree(s) is displayed (see §6.4).

It is sometimes more useful to run the parser from the Lisp command line interface, since this means that any results generated by post-processing will appear in an editor buffer and can be searched, edited and so on. It may also be useful to do this if you have to use emacs to enter diacritics. The command `do-parse-tty` is therefore available — it takes a string as an argument. For example:

```
(do-parse-tty "Kim sleeps")
```

The normal graphical parse output is produced.

Redisplay parse Shows the tree(s) from the last parse again.

Show parse chart Shows the parse chart for the last parse (see §6.6).

Batch parse This prompts for the name of a file which contains sentences on which you wish to check the operation of the parser, one sentence per line (see the file `test.items` in the sample grammars). It then prompts for the name of a new file to which the results will be output. The output tells you the number of parses found (if any) for each sentence in the input file and the number of passive edges, and gives a time for the whole set at the end. This is a very simple form of test suite: vastly more functionality is available from the [incr tsdb()] machinery which can be run in conjunction with the LKB (see §8.13).

Compare∗ This is a tool for treebanking: it displays the results of the last parse, together with a dialog that allows selection / rejection of rule applications which differ between the parses. It thus allows comparison of parses according to the rules applied. It is intended for collection of data on preferences but can also be useful for distinguishing between a large set of parse results. Specifying that a particular phrase is in/out will cause the relevant parse trees to be indicated as possible/impossible and the other phrases to be marked in/out, to the extent that this can be determined. The parameter `*discriminant-path*` can be set to identify a useful discriminating position in a structure: the default value corresponds to the location of the key relation in the semantic structure used by the LinGO ERG.

The treebanking tool is under active development at the time of writing, and so a full description is not given here. Documentation will be made available via the LKB webpage.

6.1.5 MRS∗

The MRS commands relate to semantic representation, but they assume a particular style of semantic encoding, as is used in the LinGO ERG. The grammars discussed in this book use a simplified version of MRS. MRS is briefly discussed in §5.4 and §8.11. MRS output can be displayed in various ways by clicking on the result of a parse in the compact parse tree representation (see §6.4) or displayed in the main editor window (`*common-lisp*` buffer or Listener), as controlled by the **Output level** command below. The parameterisation for MRS is controlled by various MRS-specific files, discussed in §8.11.

Load munger The term *munger* refers to a set of rules which manipulate the MRS in application-specific ways. Loading a new set of rules will overwrite the previously loaded set. Most users should ignore this.

Clear munger Deletes the munger rules.

Output level Allows the user to control the MRS output which is sent to the standard Lisp output (an emacs buffer, if emacs is being used). This command is provided since with large structures it is often more convenient to look at MRS output in emacs rather than in the MRS windows displayed by clicking on a tree in the parse output window. The default output level is NONE, but this may be changed by the grammar-specific MRS globals files.

- NONE
- BASE: a bracketed representation of the tree, plus an underspecified MRS, generally quite similar to the TFS representation.
- SCOPED: the scoped forms corresponding to the underspecified structure produced by the grammar. If no scoped forms can be produced, warning messages are output. If there are a large num-

ber of scoped forms, only a limited number are shown, by default. Because scoping can be computationally expensive, there is a limit on the search space for scopes: this is controlled by `mrs::*scoping-call-limit*`.

6.1.6 Generate∗

The generator was described in §5.4 and a few more details are given in §8.12, but is currently in a fairly early stage of development. It operates in a very similar manner to the parser but relies on the use of flat semantics such as MRS, thus it will only work with grammars that produce such semantics. Before the generator can be used, the command **Index** must be run from this menu. Alternatively, the script can include the command:

```
(index-for-generator)
```

At the moment, there is no interactive way of entering an MRS input for the generator other than by parsing a sentence which produces that MRS and then choosing **Generate** from the appropriate parse window.

Redisplay realisation Redisplays the results from the last sentence generated.

Show gen chart Displays a window showing a chart from the generation process (see §6.6). Note that the ordering of items on the chart is controlled by their semantic indices.

Load heuristics Prompts for a file which should contain a set of heuristics for determining null semantics lexical items (see §8.12).

Clear heuristics Clears a set of heuristics, loaded as above.

Index Indexes the lexicon and the rules for the generator. This has to be run before anything can be generated. Any error messages are displayed in the LKB top window.

6.1.7 Debug

Check lexicon Expands all entries in the lexicon, notifying the user of any entries which fail to expand (via error messages in the LKB top window). This will take a few minutes for a large lexicon. An alternative for small grammars is to have the command

```
(batch-check-lexicon)
```

in the script file.

Find features' type∗ Used to find the maximal type (if any) for a list of features (see §3.5.8 for a discussion of maximal types). Prompts for a list of features. Displays the maximum type in the LKB interaction window. Warns if feature is not known.

Print chart / Print parser chart∗ Displays the chart (crudely) to the standard Lisp output (e.g., the emacs buffer). This can be useful as an alternative display to the parse chart window, especially with very large charts.

Print generator chart∗ As above, but for the generator.

6.1.8 Advanced∗

Tidy up This command clears expanded lexical entries which are stored in memory. If accessed again they will be read from file and expanded again.

Expansion of a large number of word senses will tend to fill up memory with a large number of TFSs. Most commands which are likely to do this to excess, such as the batch parser, actually clear the TFSs themselves, but if a lot of sentences have been parsed interactively and memory is becoming restricted this option may be useful.

Create quick check file The check path mechanism constructs a filter which improves efficiency by processing a set of example sentences. It is discussed in more detail in §8.3.1.

The command prompts for a file of test sentences and an output file to which the resulting paths should be written. This file should subsequently be read in by the script. Note that constructing the check paths is fairly time-consuming, but it is not necessary to use a very large set of sentences. The mechanism is mildly grammar-specific in that it assumes the style of encoding where the daughters of a rule are given by an ARGS list — see §8.3.1 for details.

6.1.9 Options

Expand/Shrink menu Changes the LKB top menu so that the advanced commands are added/removed.

Set options Allows interactive setting of some system parameters. Note that the values of the boolean parameters are specified in the standard way for Common Lisp: that is, `t` indicates true and `nil` indicates false. I will not go through the parameters here: Chapter 9 gives full details of all parameters, including those that cannot be altered interactively.

If a parameter file has been read in by the script (using the load function `load-lkb-preferences`) the parameter settings are saved in the same file. Otherwise the user is prompted for the name of a file to save any preference changes to. This file would then have to be specified in the script if the changes are to be reloaded in a subsequent session.

Usually the preferences file is loaded by the script so that any preferences which are set in one session will be automatically saved for a subsequent session with that grammar. (In the cases of 'families' of grammars, the user-prefs file may be shared by all the grammars in the family.) The user should not need to look at this file and should not edit it, since any changes may be overwritten.

Save display settings Save shrunkenness of TFSs (see the description of **Shrink/Expand** in §6.3).

Load display options Load pre-saved display setting file.

6.2 Type hierarchy display

By default, a type hierarchy is displayed automatically after a grammar is loaded (though this default must be turned off for grammars that use very large numbers of types, see §9.1.1). The type hierarchy can also be accessed via the top level command **Type hierarchy** in the **View** menu, as discussed above in §6.1.3.

The top of the hierarchy, that is the most general type, is displayed at the left of the window. The window is scrollable by the user and is automatically scrolled by various **V**iew options. Nodes in the window are active; clicking on a type node will give a menu with the following options:

Shrink/Expand Shrinking a type node results in the type hierarchy being redisplayed without the part of the hierarchy which appears under that type being shown. The shrunk type is indicated by an outline box. Any subtypes of a shrunk type which are also subtypes of an unshrunk type will still be displayed. Selecting this option on a shrunk type reverses the process.

Type definition Display the definition for the constraint on that type (see §6.1.3, above).

Expanded type Display the expanded constraint for that type (see §6.1.3, above).

New hierarchy Displays the type hierarchy under the clicked-on node in a new window, via the same dialog as the top-level menu command. This is useful for complex hierarchies.

6.3 Typed feature structure display

Most of the view options display TFSs in a window. The usual orthographic conventions for drawing TFSs are followed; types are lowercased bold, features are uppercased. The order in which features are displayed in the TFS window is determined according to their order when introduced in the type specification file. For example, assume we have the following fragment of a type file:

```
sign := feat-struc &
 [ SYN *top*,
   SEM *top* ].

word := sign &
 [ ORTH string ].
```

then when a TFS of type **sign** is displayed, the features will be displayed in the order SYN, SEM; when a **word** is displayed the order will be SYN, SEM, ORTH. This ordering can be changed or further specified by means of the parameter `*feature-ordering*`, which consists of a list of features in the desired order (see §9.1.2).

The bar at the bottom of the TFS display window shows the path to the node the cursor is currently at.

Typed feature structure windows are active - currently the following operations are supported:

1. Clicking on the window identifier (i.e., the first item in the window) will display a menu of options which apply to the whole window.

   Output TeX Outputs the FS as LaTeX macros to a file selected by the user. The LaTeX macros are defined in `avmmacros` in the data directory.

   Apply lex rule Only available if the identifier points to something that might be a lexical entry. It prompts for a lexical or morphological rule and applies the rule to the entry. The result is displayed if application succeeds.

   Apply all lex rules This option is only available if the identifier points to something that might be a lexical entry. This tries to apply all the defined lexical and morphological rules to the entry, and to any results of the application and so on. (To prevent infinite recursion on inappropriately specified rules the number of applications is limited.) The results are displayed in summary form, for instance:

   ```
   dog + SG-NOUN_IRULE

   dog + PL-NOUN_IRULE
   ```

   Clicking on one of these summaries will display the resulting TFS.

   Show source Shows the source code for this structure if the system is being used with emacs with the LKB extensions. This is not available with all structures: it is not available for any entries which have been read in from a cached file.

2. Clicking on a reentrancy marker gives the following sub-menu:

   Find value Shows the value of this node, if it is not displayed at this point, scrolling as necessary.

   Find next Shows the next place in the display where there is a pointer to the node, scrolling as necessary.

3. Clicking on a type (either a parent, or a type in the TFS itself) will give a sub-menu with the following options:

   Hierarchy Scroll the type hierarchy window so that the type is centered. If the type hierarchy window is not visible, it will be redisplayed.

   Shrink/Expand Shrinking means that the TFS will be redisplayed without the TFS which follows the type being shown. The existence of further undisplayed structure is indicated by a box round the type. Atomic TFSs may not be shrunk. Shrinking persists, so that if the window is closed, and subsequently a new window opened onto that TFS, the shrunken status will be retained. Furthermore, if the shrunken structure is a type constraint, any TFSs which inherit from this constraint will also be displayed with equivalent parts hidden. For instance, if the constraint on a type has parts shrunk, any lexical entry which involves that type will also be displayed with parts hidden.
   If this option is chosen on an already shrunken TFS then the TFS will be expanded. Again this can affect the display of other structures.
   The shrunkenness state may be saved via and loaded via the **Save/Load display settings** commands on the **Options** menu (see §6.1.9).

   Show source Shows the source code for this structure if running from emacs with the LKB connection (not available with all structures).

   Type definition Display the definition for that type.

   Expanded type Display the expanded definition for that type.

   Select Selects the TFS rooted at the clicked node in order to test unification.

   Unify Attempts to unify the previously selected TFS with the selected node. Success or (detailed) failure messages are shown in the LKB Top window. See §6.3.1 for further details.

   Clicking on a type which is in fact a string, and thus has no definition etc, will result in the warning beep, and no display.

6.3.1 Unification checks

The unification check mechanism operates on TFSs that are displayed in windows. You can temporarily select any TFS or part of a TFS by clicking on the relevant node in a displayed window and choosing **Select** from the menu. Then to check whether this structure unifies with another, and to get detailed messages if unification fails, find the node corresponding to the second structure, click on that, and choose **Unify**. If the unification fails, failure messages will be shown in the top level LKB window. If it succeeds, a new TFS window will be displayed. This can in turn be used to check further unifications.

A detailed description of how to use this mechanism is in §7.4.1.

6.4 Parse output display

The parse output display is intended to give an easily readable overview of the results of a parse, even if there are several analyses. The display shows a parse tree for each separate parse, using a very small font to get as many trees as possible on the screen. Besides the standard **Close** and **Close all** buttons, the parse output display window has a button for **Show chart**: this has the same effect as the top-level menu command, it is just repeated here for convenience.

Clicking on a tree gives several options:

Show enlarged tree produces a full size parse tree window, as described in §6.5, with clickable nodes.

Highlight chart nodes will highlight the nodes on the parse chart corresponding to this tree. If the parse chart is not currently displayed, this option will bring up a new window (see §6.6 for details of the chart display).

Generate Tries to generate from the MRS for this parse. Note that in order to run the generator, the **Generate / Index** command must have been run. If generation succeeds, the strings generated are shown in a new window — clicking on the strings gives two options:

> **Edge** displays the tree associated with that realization,
>
> **Feature structure** displays the TFS associated with that realization.
>
> If generation fails, the message 'No strings generated' will appear in the LKB interaction window.

MRS Displays an MRS in the feature structure style representation.

Prolog MRS Displays an MRS in a Prolog compatible notation (designed for full MRSs, rather than simplified MRSs).

Indexed MRS Displays an MRS using the alternative linear notation.

Scoped MRS Displays all the scopes that can be constructed from the MRS: warning messages will be output if the MRS does not scope.

6.5 Parse tree display

Parse trees are convenient abbreviations for TFSs representing phrases and their daughters. When a sentence is successfully parsed, the trees which display valid parses are automatically shown, but parse trees may also be displayed by clicking on any edge in a parse chart (see §6.6). The nodes in the parse tree are labelled with the name of the (first) parse node label which has a TFS which matches the TFS associated with the node, if such a label is present. The matching criteria are detailed in §4.5.7 and §8.14.

The input words are indicated in bold below the terminal parse tree nodes — if any morphological rules have been applied, these are indicated by nodes beneath the words if the parameter `*show-morphology*` is `t`, but not shown otherwise. Similarly, there is a parameter `*show-lex-rules*` which controls whether or not the lexical rule applications are displayed. Both these parameters may be set interactively, via the **Options / Set options** menu command.

Clicking on a node in the parse tree will give the following options:

Feature structure - Edge X (where X is the edge number in the parse chart) displays the TFS associated with a node. Note that if the parameter `*deleted-daughter-features*` is set, the tree will still display the full structure (it is reconstructed after parsing). See §8.3.3.

Show edge in chart Highlights the node in the chart corresponding to the edge. The chart will be redisplayed if necessary. Currently not available for a tree produced by the generator.

Rule X (where X is the name of the rule used to form the node) displays the TFS associated with the rule.

Generate from edge This attempts to generate a string from the MRS associated with this node. Behaves as the **Generate** command from the parse output display. Can give strange results if the node is not the uppermost one in the tree. Currently not available with a tree produced by the generator. Note that in order to run the generator, the **Generate / Index** command must have been run.

Lex ids This isn't a selectable option - it's just here as a way of listing the identifiers of the lexical entries under the node.

6.6 Chart display

The chart is a record of the structures that the LKB system has built in the course of attempting to find a valid parse or parses (see §4.2). A structure built by the parser and put on the chart is called an *edge*: edges are identified by an integer (*edge number*). By default, all edges that are displayed on the chart represent complete rule applications.

The chart window shows the words of the sentence to the left, with lines indicating how the structures corresponding to these words are combined to form phrases. Each node in the chart display corresponds to an edge in the chart. A node label shows the following information:

1. The nodes of the input that this edge covers (where the first node is notionally to the left of the first word and is numbered 0, just to show we're doing real computer science here).
2. The edge number (in square brackets).
3. The name of the rule used to construct the edge (or the type of the lexical item).

For instance, in the chart for the sentence *the dogs chased the cats*, the nodes for the input are numbered

$._0$ *the* $._1$ *dogs* $._2$ *chased* $._3$ *the* $._4$ *cats* $._5$

In the chart display resulting from parsing this sentence in the `g8gap` grammar, one edge is specified as:

```
2-5 [19] HEAD-COMPLEMENT-RULE-1
```

Thus this edge is edge number 19, it covers *chased the cats*, and was formed by applying the `head-complement-rule-1`.

The chart display is sensitive to the parameters `*show-morphology*` and `*show-lex-rules*` in a similar way to the tree display.

Moving the cursor over an edge in the chart displays the yield of the edge at the bottom of the window. Clicking on a word node (i.e., one of the nodes at the leftmost side of the chart which just show orthography) will select it. When at least one word is selected, all the edges that cover all the selected words are highlighted. Clicking on a word node again deselects it.

Clicking on an edge node results in the following menu:

Highlight nodes Highlights all the nodes in the chart for which the chosen node is an ancestor or a descendant. This option also selects the node so that it can be compared with another node (see **Compare**, below).

Feature structure Shows the TFS for the edge. Unlike the parse tree display, this represents the TFS which is actually used by the parser, see the discussion in §4.2. It is not reconstructed if `*deleted-daughter-features*` is used (see §8.3.3).

Rule X Shows the TFS for the rule that was used to create this edge

New chart Displays a new chart which only contains nodes for which the chosen node is an ancestor or a descendant (i.e., those that would be highlighted). This is useful for isolating structures when the chart contains hundreds of edges.

Tree Shows the tree headed by the phrase corresponding to this edge

Compare This option is only available if another node has been previously selected (using **Highlight Nodes**). The two nodes are compared using the parse tree comparison tool described in §6.1.4.

Unify This is only shown if a TFS is currently **Select**ed for the unification test — see §6.3.1.

7

Error messages and debugging techniques

This chapter is intended to help with debugging. There are two sorts of problems which arise when writing grammars in the LKB. In the first class, the system doesn't accept your grammars files and generates error messages. This type of problem is very irritating when you are learning how to use the system, but with experience, such problems generally become easy to fix. In this chapter, the error messages are explained in detail with references back to the chapters discussing the LKB formalism. The second type of problems are more difficult: the system doesn't give explicit error messages, but doesn't do what you want it to. Some debugging tools that can be used in this situation are described in §7.4.

7.1 Error messages

The formal conditions on the type hierarchy and the syntax of the language were detailed in Chapters 3 and 4. Here we will go through those conditions informally, and discuss what happens when you try and load a file in which they are violated. If you do not understand the terminology, please refer back to the earlier chapters.

Many examples of errors are given below: these all assume that we have made the minimal change to the `g8gap` grammar to make it match the structures shown. The errors are not supposed to be particularly realistic!

IMPORTANT NOTE: Look at all the messages in the LKB Top window when you load a grammar and always look at the first error message first! Error messages may scroll off the screen, so you may need to scroll up in order to do this. Sometimes errors propagate, causing other errors, so it's a good idea to reload the grammar after you have fixed the first error, rather than try and fix several at once, at least until you have gained familiarity with the system.

7.1.1 Type loading errors: Syntactic well-formedness

If the syntax of the constraint specifications in the type file is not correct, according to the definition in §4.4.6, then error messages will be generated. The system tries to make a partial recovery from syntactic errors, either by skipping to the end of a definition or inserting the character it expected, and then continuing to read the file. This recovery does not always work: sometimes the inserted character is not the intended one and sometimes an error recovery affects a subsequent definition. Thus you may get multiple error messages from a single error. The system will not try to do any further well-formedness checking on files with any syntactic errors. In the examples below, an incorrect definition is shown followed by the error message that is generated. All the definitions are based on `g8gap/types.tdl`.

Example 1: missing character

```
agr : *top*.
```

```
Syntax error at position 132:
Syntax error following type name AGR
Ignoring (part of) entry for AGR
Error: Syntax error(s) in type file
```

The error is caused by the missing `=` following the `:`. The error message indicates the position of the error (using emacs you can use the command goto-char to go to this position in the file). The number given will not always indicate the exact position of the problem, since the LKB's TDL description reader may not be able to detect the problem immediately, but is likely to be quite close. The system then says what it is doing to try and recover from the error (in this case, ignore the rest of the entry) and finally stops processing with the error message `Syntax error(s) in type file` (I will omit this in the rest of the examples).

Example 2: missing character

```
semantics := *top* &
[ INDEX index,
  RELS *dlist* .
```

```
Syntax error: ] expected and not found in SEMANTICS
at position 403
Inserting ]
```

In this example, the system tries to recover by inserting the character it thinks is missing, correctly here.

Example 3: missing character

```
semantics := *top* &
[ INDEX index
  RELS *dlist*] .
```

```
Syntax error: ] expected and not found in SEMANTICS
at position 389
Inserting ]
Syntax error: . expected and not found in SEMANTICS
at position 389
Inserting .
Syntax error at position 394
Incorrect syntax following type name RELS
Ignoring (part of) entry for RELS
```

Here the system diagnosed the error incorrectly, since in fact a comma was missing rather than a '`]`'. The system's recovery attempt doesn't work, and the error propagates. This illustrates why you should reload the grammar after fixing the first error unless you are reasonably sure the error messages are independent.

Example 4: coreference tag misspelled

```
unary-rule := phrase &
[ ORTH #orth,
  SEM #cont,
  ARGS < [ ORTH #orth, SEM #comt ] > ].
```

```
Syntax error at position 821: Coreference COMT
only used once
```

In this example, the system warns that the coreference was only used once: it is assumed that this would only be due to an error on the part of the user.

Other syntax errors You may also get syntax errors such as the following:

```
Unexpected eof when reading X
```

eof stands for end of file — this sort of message is usually caused by a missing character.

7.1.2 Conditions on the type hierarchy

After a syntactically valid type file (or series of type files) is read in, the hierarchy of types is constructed and checked to ensure it meets the conditions specified in §3.2.

All types must be defined If a type is specified to have a parent which is not defined anywhere in the loaded files, an error message such as the following is generated:

```
AGR specified to have non-existent parent *TOPTYPE*
```

Although it is conventional to define parent types before their daughters in the file, this is not required, and order of type definition in general has no significance for the system. Note however that it is possible to redefine types, and if this is done, the actual definition will be the last one the system reads. If two definitions for types of the same name occur, a warning message will be generated, for instance:

```
    Type AGR redefined
```

Connectedness / unique top type There must be a single hierarchy containing all the types. Thus it is an error for a type to be defined without any parents, for example:

```
sign :=
[ ORTH *dlist*,
  HEAD pos,
  SPR *list*,
  COMPS *list*,
  SEM semantics,
  GAP *dlist*,
  ARGS *list* ].
```

Omitting the parent(s) of a type will cause an error message such as the following:

```
Error: Two top types *TOP* and SIGN have been defined
```

To fix this, define a parent for the type which is not intended to be the top type (i.e., **sign** in this example).

If a type is defined with a single parent which was specified to be its descendant, the connectedness check will give error messages such as the following for every descendant of the type:

```
NOUN not connected to top
```

(This situation is also invalid because cycles are not allowed in the hierarchy, but because of the way cycles are checked for, the error will be found by the connectedness check rather than the cyclicity check).

No cycles It is an error for a descendant of a type to be one of that type's ancestors. This causes an error message to be generated such as:

```
Cycle involving TERNARY-HEAD-INITIAL
```

The actual type specified in the messages may not be the one that needs to be changed, because the system cannot determine which link in the cycle is incorrect.

Redundant links This is the situation where a type is specified to be both an immediate and a non-immediate descendant of another type. For instance, suppose g8gap/types.tdl is changed so that **phrase** is specified as a parent of **unary-head-initial** even though it is already a parent of **head-initial** (and also a parent of **unary-rule** which is a parent of **unary-rule-passgap**):

```
unary-head-initial := unary-rule-passgap & head-initial
                      & phrase.
```

Then the error messages are:

```
Redundancy involving UNARY-HEAD-INITIAL
UNARY-HEAD-INITIAL: PHRASE is redundant
- it is an ancestor of UNARY-RULE-PASSGAP
UNARY-HEAD-INITIAL: PHRASE is redundant
- it is an ancestor of HEAD-INITIAL
```

The assumption is that this would only happen because of a user error. The condition is checked for because it could cause problems with the greatest lower bound code (see §4.5.3). After finding this error, the system checks for any other redundancies and reports them all.

7.1.3 Constraints

Once the type hierarchy is successfully computed, the constraint descriptions associated with types are checked, and inheritance and typing are performed to give expanded constraints on types (see §3.5).

Valid constraint description The check for syntactic well-formedness of the constraint description is performed as the files are loaded, but errors such as missing types which prevent a valid TFS being constructed are detected when the constraint description is expanded. For example, suppose the following was in the g8gap type file, but the type **foobar** was not defined.

```
sign := *top* &
[ ORTH *dlist*,
  HEAD foobar,
  SPR *list*,
  COMPS *list*,
  SEM semantics,
  GAP *dlist*,
  ARGS *list* ].
```

The first error messages are as follows:

```
Invalid type FOOBAR
Unifications specified are invalid or do not unify
Type SIGN has an invalid constraint specification
Type PHRASE's constraint specification clashes with
its parents'
```

Note that the error propagates because the descendants' constraints cannot be constructed either.

Another similar error is to declare two nodes to be reentrant which have incompatible values. For example:

```
sign := *top* &
[ ORTH *dlist*,
  HEAD pos & #1,
  SPR *list*,
  COMPS *list* & #1,
  SEM semantics,
  GAP *dlist*,
  ARGS *list* ].
```

The error messages would be very similar to the case above:

```
Unifications specified are invalid or do not unify
Type SIGN has an invalid constraint specification
Type PHRASE's constraint specification clashes with
its parents'
```

No Cycles TFSs are required to be acyclic in the LKB system (see §3.3): if a cycle is constructed during unification, then unification fails. In the case of construction of constraints, this sort of failure is indicated explicitly. For example, suppose the following is a type definition

```
wrong := binary-rule &
 [ ARGS < #1 & [ GAP < #1 > ], *top* > ] .
```

The following error is generated:

```
Cyclic check found cycle at < GAP : FIRST >
Unification failed - cyclic result
Unification failed: unifier found cycle at
                           < ARGS : FIRST >
Type WRONG has an invalid constraint specification
```

Consistent inheritance Constraints are constructed by monotonic inheritance from the parents' constraints. If the parental constraints do not unify with the constraint specification, or, in the case of multiple parents, if the parents' constraints are not mutually compatible, then the following error message is generated:

```
Type X's constraint specification clashes with its parents'
```

Maximal introduction of features As described in §3.5, there is a condition on the type system that any feature must be introduced at a single point in the hierarchy. That is, if a feature, F, is mentioned at the top level of a constraint on a type, **t**, and not on any of the constraints of ancestors of **t**, then all types where F is used in the constraint must be descendants of **t**. For example, the following would be an error because NUMAGR is a top level feature on both the constraint for **agr-cat** and **pos** but not on the constraints of any of their ancestors:

```
pos := *top* & [ MOD *list* ].

nominal := pos & [ NUMAGR agr ].

pseudonom := pos & [ NUMAGR agr ].
```

The error message is as follows:

```
Feature NUMAGR is introduced at multiple types
(POS PSEUDONOM)
```

To fix this, it is necessary to introduce another type on which to locate the feature. For example:

```
pos := *top* & [ MOD *list* ].

numintro := pos & [ NUMAGR agr ].

nominal := numintro.

pseudonom := numintro.
```

No infinite structures It is an error for a constraint on a type to mention that type inside the constraint. For example, the following is invalid.

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

The reason for this is that expansion of the constraint description would create an infinite structure (as discussed in §3.5.8). The following error message is produced:

```
Error in *NE-LIST*:
  Type *NE-LIST* occurs in constraint for type
  *NE-LIST* at (REST)
```

Similarly it is an error to mention a daughter of a type in its constraint. It is also an error to make two types mutually recursive:

```
foo := *top* &
[ F bar ].

bar := *top* &
[ G foo ].
```

The error message in this case is:

```
BAR is used in expanding its own constraint
                    expansion sequence: (FOO BAR)
```

Note that it *is* possible to define recursive constraints on types as long as they specify an ancestor of their type. For example, a correct definition of **list** is:

```
*list* := *top*.

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

*null* := *list*.
```

Type inference — features There are two cases where typing may fail due to the feature introduction condition. The first is illustrated by the following example:

```
noun-lxm := lexeme &
[ HEAD [ NUMAGR #agr,
         INDEX *top* ],
  SPR < [HEAD det & [NUMAGR #agr],
         SEM.INDEX #index ] >,
  COMPS < >,
  SEM [ INDEX object & #index ] ].
```

Here, the feature INDEX is only defined for structures of type **semantics**. This type clashes with **pos** which is the value of HEAD specified higher in the hierarchy. This example generates the following error messages:

```
Error in NOUN-LXM:
   No possible type for features (NUMAGR INDEX) at
   path (HEAD)
```

A different error message is generated when a type is specified at a node which is incompatible with the node's features. For instance:

```
test := lex-item &
  [ ARGS < pos &
        [ HEAD *top* ] > ].
```

```
Error in TEST:
  Type of fs POS at path (ARGS FIRST) is incompatible
with features (HEAD) which have maximal type SIGN
```

Type inference — type constraints The final class of error is caused when type inference causes a type to be determined for a node which then clashes with an existing specification on a path from that node. For instance:

```
nominal := pos & [ NUMAGR agr,
                   MOD <> ].


test1 := lexeme &
        [ ARGS < [ HEAD [ NUMAGR sg,
                          MOD < *top* > ]] > ].
```

Here the feature NUMARG in **test1** means that the node at the end of the path ARGS.FIRST.HEAD has to be of type **nominal**, but **nominal** specifies that its value for MOD is the empty list. The error message is

```
Unification with constraint of NOMINAL failed at
path (ARGS FIRST HEAD)
```

The **Debug / Find features' type** command on the expanded menu (§6.1.7) can be useful when trying to fix such problems.

7.2 Lexical entries

When lexicon files are loaded, they are generally only checked for syntactic correctness (as defined in §4.4.3) — entries are only fully expanded when they are needed during parsing or generation or because of a user request to view an entry. Thus when loading the lexicon, you may get syntactic errors similar to those discussed in §7.1.1 above, but not content errors since the TFSs are not expanded at load time. With a small lexicon, you can put the command `batch-check-lexicon` in the `script` file, in order to check correctness at load time, as was done with the example grammars for this book.

Incorrect lexical entries may therefore only be detected when you view a lexical entry or try to parse with it. The error messages that are obtained are very similar to some of those discussed for the type loading above, specifically:

Valid constraint description
No Cycles
Consistent inheritance
All types must be defined
Type inference — features
Type inference — type constraints

There is a menu option to do a complete check on a loaded lexicon for correctness: **Check lexicon** under **Debug**. See §6.1.7.

7.3 Grammar rules

Unlike lexical entries, grammar and lexical rules are always expanded at load time. Therefore you may get error messages similar to those listed above for lexical entries when the rules are loaded. Rules must expand out into TFSs which have identifiable paths for the mother and daughters of the rule (§4.4.4). For the grammars we have been looking at, the mother path is the empty path and the daughter paths are defined in terms of the ARGS feature. For example:

$$\begin{bmatrix}\textbf{binary-rule} & \\ \textsc{args} & \begin{bmatrix}\textsc{first}\ \textbf{sign} \\ \textsc{rest}\ \begin{bmatrix}\textsc{first}\ \textbf{sign}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Here the mother is given by the empty path, one daughter by the path ARGS.FIRST and another by the path ARGS.REST.FIRST. The mother path and a function which gives the daughters in the correct linear order must be specified as system parameters in the `globals` and `user-fns` files respectively (see §9.2.3 and §9.3).

7.4 Debugging techniques

If something unexpectedly fails to parse or if too many parses are produced, then isolating the problem can be quite tricky. The first thing to look at is usually the parse chart, though you may need to try and find a simpler sentence that produces the same effect, because the parse chart is difficult to navigate with long sentences. In extreme cases, the **Debug / Print chart** command may be useful (see §6.1.7). Note that the display at the bottom of the parse chart is extremely helpful in showing you what node you are looking at. Once the problem is isolated to a particular phrase, or small set of phrases, the unification checking mechanism discussed below becomes helpful.

Unfortunately debugging grammars is not a skill that can easily be described. After a while, a grammar writer becomes sufficiently familiar with a grammar that the likely sources of problems become reasonably obvious, but this takes practice. Most of the same principles apply as when debugging computer programs: in fact, the parse chart display can be thought of as a backtrace mechanism. Because the LKB is a development environment, we have kept the amount of grammar compilation to a minimum, so that the time taken to reload a grammar is fast, even with the LinGO ERG. For large scale grammars, the more sophisticated test suite techniques described in §8.13 are an invaluable tool. As soon as you start working with grammars, you should investigate the use of a version control system, such as the Concurrent Versions System (CVS): even if you are working by yourself, the ability to easily recover previous versions of grammars is extremely useful.

7.4.1 Use of the unification checking mechanism

As mentioned in §6.3.1 there is an interactive mechanism for checking unification via the TFSs that are displayed in windows. You can temporarily select any TFS or part of a TFS by clicking on the relevant node in a displayed window and choosing **Select** from the menu. Then to check whether this structure unifies with another, and to get detailed messages if unification fails, find the node corresponding to the second structure, click on that, and choose **Unify**. If the unification fails, failure messages will be shown in the top level LKB window. If it succeeds, a new TFS window will be displayed. This can in turn be used to check further unifications. Here we give a detailed example of how to use this.

Consider why *the dog chased* does not parse in the `g8gap` grammar. By looking at the parse chart for the sentence, you can determine that although the `head-specifier-rule` applies to give the phrase *the dog chased*, this isn't accepted as a parse. This must be because it fails the start symbol condition. To verify this:

1. Load the `g8gap` grammar.
2. Do **Parse / Parse input** on `the dog chased`. The parse will fail.
3. Do **Parse / Show parse chart**. A chart window will appear.
4. Click on the rightmost node in the chart, which is labelled HEAD-SPECIFIER-RULE and choose **Feature structure**. A TFS window will appear.
5. Click on the root node of the TFS (i.e., the node with type **binary-head-second-passgap**) and choose **Select**.
6. Display the TFS for `start` by choosing **Lex entry** from the **View** menu (`start` is not actually a lexical entry, but we wanted to avoid putting too many items on the **View** menu, so **Lex entry** is used as a default for viewing 'other' entries too).

7. Click on the root node of `start` (i.e., the node with the type **phrase**) and choose **Unify** from the pop-up menu.
8. Look at the LKB Top window. You should see:

   ```
   Unification of *NE-LIST* and *NULL* failed at path
                                          < GAP : LIST >
   ```

   This demonstrates that the parse failed because the phrase contains a gap.

It is instructive to redefine `start` in `g8gap` as below, reload the grammar and retry this experiment:

```
start := phrase &
[ HEAD verb,
  SPR <>,
  COMPS <>,
  GAP <! !> ].
```

To check why a rule does not work, a more complex procedure is sometimes necessary, because of the need to see whether three (or more) TFSs can be unified. Suppose we want to check to see why the head-specifier rule does not apply to *the dog bark* in `g8gap`. This is relatively complex, since both the TFSs for the NP for *the dog* and the VP for *bark* will individually unify with the daughter slots of the `head-specifier-rule`. So we need to use the intermediate results from the unification test mechanism.

The following description details one way to do this.

1. Parse *the dog bark*. The parse will fail.
2. Select **Show chart** to show the edges produced during the failed parse.
3. Bring up the TFS window for the `head-specifier-rule` (uninstantiated), either via the **View Rule** command in the top window menu or by choosing **Rule HEAD-SPECIFIER-RULE** from a head-specifier-rule node in the chart.
4. Find the node in the `head-specifier-rule` window corresponding to the specifier (i.e., the node labelled **phrase** at the end of the path ARGS.FIRST). Click on it and choose **Select**.
5. Find the node for `the dog` in the parse chart window and choose **Unify**. (Note that this is a shortcut which is equivalent to displaying the TFS for the phrase via the **Feature Structure** option in the chart menu and then selecting **Unify** from the menu on the top node of the TFS displayed.) Unification should succeed and a new TFS window (titled Unification Result) will be displayed.

6. Find the node in the new Unification Result window corresponding to the head in the rule, i.e., the node at the end of the path ARGS.REST.FIRST, click on it and choose **Select**.
7. Click on the node in the parse chart for the VP for `bark` (i.e., the one labelled `head-complement-rule-0`) and choose **Unify**.
8. This time unification should fail, with the following message in the LKB Top window:

```
Unification of SG and PL failed at path
< SPR : FIRST : HEAD : NUMAGR >
```

Note that it is important to **Select** the node in the rule and **Unify** the daughters rather than vice versa because the result shown always corresponds to the initially selected TFS. We need this to be the whole rule so we can try unifying the other daughter into the partially instantiated result.

8 Advanced features

The previous chapters described the main features of the LKB system which are utilized by most of the distributed grammars. There are a range of other features which are in some sense advanced: for instance because they concern facilities for using the LKB with grammars in frameworks other than the variety of HPSG assumed here, or because they cover functionality which is less well tested (in particular defaults and generation), or because the features are primarily used for efficiency. These features are described in this chapter, which is a series of more or less polished notes about various aspects of the LKB. The intention is that the LKB website:

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

will contain updated information. In any case, this chapter should only be read after working through the earlier material in the book in some detail. On the whole, it assumes rather more knowledge of NLP and of Lisp programming than I have in the earlier chapters.

8.1 Defining a new grammar

If possible, it is best to start from one of the sample grammars rather than to build a new grammar completely from scratch. However, when building a grammar in a framework other than HPSG, the existing sources may not be of much use. These notes are primarily intended for someone trying to build a grammar almost from scratch.

The first step is to try and decide whether the LKB system is going to be adequate for your needs. The system is not designed for building full NLP applications, though it can form part of such a system for teaching or research purposes, and has some utility for development of commercial applications. Grammars can be written and debugged in the LKB and then deployed using more efficient platforms such as Callmeier's PET system. There are some limitations imposed by the typed feature structure formalism.[56] There's no way of describing any form of transformation or movement directly, though, as we saw in §5.5, feature structure formalisms have alternative ways of achieving the same effects.

Even with respect to other typed feature structure formalisms, the LKB has some self-imposed limitations. As I have discussed, there is no way of writing a disjunctive or negated TFS (see §5.3.4). In many cases, grammars can be reformulated to eliminate disjunction in favour of the use of types which express generalisations. Occasionally it may be better to have multiple lexical entries or multiple grammar rules. The LKB does not support negation, although we do intend to release a version which incorporates inequalities (Carpenter, 1992) at some point. More fundamentally for HPSG, the system does not support set operations. Alternative formalisations are possible for most of the standard uses of sets. For example, operations which are described as set union in Pollard and Sag (1994) can be reformulated as a list append. We think we have good reasons for adopting these limitations, and that the LinGO ERG shows that a large scale HPSG grammar can be built without these devices, so the LKB system is unlikely to change in these respects.

There are other limitations which are less fundamental, though they might require considerable reimplementation. In these cases, it may be worth considering using the LKB system if you have some Lisp programming experience, or can persuade someone else to do some programming for you! For instance, the current system for encoding affixation is quite restricted. However, the interface to the rest of the system is well-defined, so it would be relatively easy to replace. As mentioned in previous chapters, the implementation of the parser has limitations for languages with relatively free word order. It would be possible to replace the parsing module to experiment with different algorithms. If you want to attempt any modifications like this, please feel free to email the LKB mailing list for advice (see the website for details of how to subscribe).

If you've decided you want to use the LKB system, then you should start by defining a very simple grammar. It will make life simpler if you copy the definitions for basic types like lists and difference lists from the existing grammars, so you do not have to redefine the global parameters unnecessarily. Similarly, if you are happy to use a list feature ARGS to

[56]The system can be used to build grammars which are Turing equivalent, so these comments aren't about formal power. There is a useful contrast between whether a language *supports* a technique, which means it supplies the right primitives etc, or merely *enables* its use, which means that one can implement the technique if one is sufficiently devious. What's of interest here is the techniques the LKB supports or doesn't support.

indicate the order of daughters in a grammar rule, you will not have to change the parameters which specify daughters or the ordering function. There are some basic architectural decisions which have to be taken early on. For instance, if you decide on an alternative to the **lexeme**, **word**, **phrase** distinction described in Chapter 5, this will affect how you write lexical and grammar rules.

As described in §4.5, you need to have distinct files for each class of object. You will have to write a script file to load your grammar files — the full details of how to do this are given below, but the easiest technique is to copy an existing script and to change the file names, then to look at the documentation if you find the behaviour surprising. To start off with, aim at a grammar which is comparable in scope to the `g5lex` grammar: i.e., use fully inflected forms, one or two grammar rules, a very small number of lexical entries and just enough types to make the structures well-formed (the `g5lex` grammar actually has more types than are strictly speaking needed, because it was designed to be relatively straightforward to extend using a predefined feature structure architecture).

There are many practical aspects to grammar engineering, most of which are similar to good practise in other forms of programming (see Copestake and Flickinger (2000) for some discussion). One aspect which is to some extent peculiar to grammar writing is the use of test suites (see e.g., Oepen and Flickinger, 1998). The LKB system is compatible with the [incr tsdb()] system, as discussed in §8.13. You should use a test suite of sentences as soon as you have got anything to parse, to help you tell quickly when something breaks. You should also adopt a convention with respect to format of description files right from the start: e.g., with respect to upper/lower case distinctions, indentation etc. Needless to say, comments and documentation are very important. As mentioned in §7.4, using a version control system such as CVS really does help when developing even moderately complex grammars. You may find the LKB's typing regime annoying at first, but we have found that it catches a large number of bugs quickly which can otherwise go undetected or be very hard to track down.

At some point, if you develop a moderate size grammar, or have a slow machine, you will probably start to worry about processing efficiency. Although this is partly a matter of the implementation of the LKB system, it is very heavily dependent on the grammar. For instance, the grammar based on Sag and Wasow (1999) (i.e., the 'textbook' grammar) is not a good model for anyone concerned with efficient processing, because it makes use of large numbers of non-branching rules. The LKB system code has been optimized with respect to the LinGO ERG, so it may well have considerable inefficiencies for other styles of grammar. For instance, the ERG has about 40 rules — grammars with hundreds of rules would probably benefit from an improvement in the rule lookup mechanism. On the other hand, we have put a lot of effort in making processing efficient with type hierarchies like the ERG's which contains thousands of types, with a reasonably high degree of multiple inheritance, and the LKB performs much better in this respect than many other typed feature structure systems.

8.2 Script files

Here is an example of a complex script file, as used for the CSLI LinGO ERG:

```
(lkb-load-lisp (parent-directory) "Version.lisp" t)
(lkb-load-lisp (this-directory) "globals.lsp")
(lkb-load-lisp (this-directory) "user-fns.lsp")
(load-lkb-preferences (this-directory) "user-prefs.lsp")
(lkb-load-lisp (this-directory) "checkpaths.lsp" t)
(lkb-load-lisp (this-directory) "comlex.lsp" t)
(load-irregular-spellings
   (lkb-pathname (parent-directory) "irregs.tab"))
(read-tdl-type-files-aux
   (list
      (lkb-pathname (parent-directory) "fundamentals.tdl")
      (lkb-pathname (parent-directory) "lextypes.tdl")
      (lkb-pathname (parent-directory) "syntax.tdl")
      (lkb-pathname (parent-directory) "lexrules.tdl")
      (lkb-pathname (parent-directory) "auxverbs.tdl")
      (lkb-pathname (this-directory) "mrsmunge.tdl"))
   (lkb-pathname (this-directory) "settings.lsp"))
(read-cached-leaf-types-if-available
   (list (lkb-pathname (parent-directory) "letypes.tdl")
         (lkb-pathname (parent-directory) "semrels.tdl")))
(read-cached-lex-if-available
   (lkb-pathname (parent-directory) "lexicon.tdl"))
(read-tdl-grammar-file-aux
   (lkb-pathname (parent-directory) "constructions.tdl"))
(read-morph-file-aux
   (lkb-pathname (this-directory) "inflr.tdl"))
(read-tdl-start-file-aux
   (lkb-pathname (parent-directory) "roots.tdl"))
(read-tdl-lex-rule-file-aux
   (lkb-pathname (parent-directory) "lexrinst.tdl"))
(read-tdl-parse-node-file-aux
   (lkb-pathname (parent-directory) "parse-nodes.tdl"))
(lkb-load-lisp (this-directory) "mrs-initialization.lsp" t)
```

I won't go through this in detail, but note the following:

1. The command to read in the script file is specified to carry out all the necessary initializations of grammar parameters etc. So although it might look as though a script file can be read in via `load` like a standard Lisp file, this would cause various things to go wrong.
2. The first load statement looks for a file called `Version.lsp` in the directory above the one where the script file itself is located. (As before, all paths are given relative to the location of the script file, so the same script will work with different computers, provided the directory structure is maintained.) The file `Version.lsp` sets a variable that records the grammar version. This is used for record-keeping purposes and also to give names to the cache files (see §8.8).
3. The user preferences file (`user-prefs.lsp`) is loaded automatically. It is kept in the same directory as the other globals file, which allows a user to set up different preferences for different grammars.
4. The `checkpaths.lsp` file is loaded to improve efficiency, as discussed in §8.3.1. The third argument to `lkb-load-lisp` is `t`, to indicate that the file is optional.
5. The `comlex.lsp` file contains code which provides an interface to a lexicon constructed automatically from the COMLEX lexicon which is distributed by the Linguistic Data Consortium. This acts as a secondary lexicon when the specially built lexicon is missing a word entry.
6. There is a list of type files read in by `read-tdl-type-files-aux` — the second argument to this function is a file for display settings (see §6.1.9).
7. Two type files are specified as leaf types (see §8.7). The leaf types are cached so that they can be read in quickly if the files are unaltered.
8. The lexicon is cached so that it can be read in quickly if it is unaltered: see §8.8.
9. The final file `mrs-initialization.lsp` contains code to initialize the behaviour of the MRS code (if present). This is responsible for loading the grammar-specific MRS parameters file.

8.2.1 Loading functions

The following is a full list of available functions for loading in LKB source files written using the TDL syntax. All files are specified as full pathnames. Unless otherwise specified, details of file format etc are specified in Chapters 4 and 5 and error messages etc are in Chapter 7.

`load-lkb-preferences` *directory file-name*

Loads a preferences file and sets `*user-params-file*` to the name of that file, so that any preferences the user changes interactively will be reloaded next session.

`read-tdl-type-files-aux` *file-names* &optional *settings-file*

Reads in a list of type files and processes them. An optional settings file controls shrunkenness (see §6.1.9). If you wish to split types into more than one file, they must all be specified in the file name list, since processing assumes it has all the types (apart from leaf types).

`read-tdl-leaf-type-file-aux` *file-name*

Reads in a leaf type file. There may be more than one such command in a script. See §8.7.

`read-cached-leaf-types-if-available` *file-name(s)*

Takes a file or a list of files. Reads in a leaf type cache if available (WARNING, there is no guarantee that it will correspond to the file(s) specified). If there is no existing leaf type cache, or it is out of date, reads in the specified files using `read-tdl-leaf-type-file-aux`. See §8.8.

`read-tdl-lex-file-aux` *file-name*

Reads in a lexicon file. There may be more than one such command in a script.

`read-cached-lex-if-available` *file-name(s)*

Takes a file or a list of files. Reads in a cached lexicon if available (WARNING, there is no guarantee that it will correspond to the file(s) specified). If there is no existing cached lexicon, or it is out of date, reads in the specified files using `read-tdl-lex-file-aux`. See §8.8.

`read-tdl-grammar-file-aux` *file-name*

Reads in a grammar file. There may be more than one such command in a script.

`read-morph-file-aux` *file-name*

Reads in a lexical rule file with associated affixation information. Note that the morphology system assumes there will only be one such file in a grammar, so this command may not occur more than once in a script, even though it takes a single file as argument.

`read-tdl-lex-rule-file-aux` *file-name*

Reads in a lexical rule file where the rules do not have associated affixation information. There may be more than one such command in a script.

`load-irregular-spellings` *file-name*

Reads in a file of irregular forms. It is assumed there is only one such file in a grammar.

`read-tdl-parse-node-file-aux` *file-name*

Reads in a file containing entries which define parse nodes.

`read-tdl-start-file-aux` *file-name*

This command simply defines the entries, without giving them any particular functionality — this works because start symbols are enumerated in the globals file.

`read-tdl-psort-file-aux` *file-name*

This is actually defined in the same way as the previous command: it simply defines the entries, without giving them any particular functionality. It is retained for backward compatibility.

8.2.2 Utility functions

The following functions are defined as useful utilities for script files.

`this-directory`

Returns the directory which contains the script file (only usable inside the script file).

`parent-directory`

Returns the directory which is contains the directory containing the script file (only usable inside the script file).

`lkb-pathname` *directory name*

Takes a directory as specified by the commands above, and combines it with a file name to produce a valid full name for the other commands.

`lkb-load-lisp` *directory name* &optional *boolean*

Constructs a file name as with `lkb-pathname` and loads it as a Lisp file. If optional is `t` (i.e., true), ignore the file if it is missing, otherwise signals a (continuable) error.

8.3 Parsing and generation efficiency techniques

8.3.1 Check paths

The check paths mechanism greatly increases the efficiency of parsing and generation with large grammars like the LinGO ERG. It relies on the fact that unification failure during rule application is normally caused by type incompatibility on one of a relatively small set of paths. These paths can be checked very efficiently before full unification is attempted, thus providing a filtering mechanism which considerably improves performance. The paths are constructed automatically by batch parsing a representative range of sentences.

The menu command **Create quick check file**, described in 6.1.8, allows a set of check-paths to be created on the basis of a set of test sentences. To do the same thing non-interactively, the macro `with-check-path-list-collection` is used. It takes two arguments: the first is is a call to a batch parsing function. For example:

```
(with-check-path-list-collection "~aac/checkpaths.lsp"
  (parse-sentences "~aac/grammar/lkb/test-sentences"
    "~aac/grammar/lkb/results"))
```

The file of checkpaths created in this way is then read in as part of the script. For instance:

```
(lkb-load-lisp (this-directory) "checkpaths.lsp" t)
```

To maintain filtering efficiency, the checkpaths should be recomputed whenever there is a major change to the architecture of the TFSs used in the grammar. However, even if they are out-of-date, the checkpaths will never affect the result of parsing or generation, only the efficiency.

8.3.2 Key-driven parsing and generation

When the parser or generator attempts to apply a grammar-rule to two or more daughters, it is often the case that it is more efficient to check the daughters in a particular order, so that if unification is going to fail, it will do so as quickly as possible. This mechanism allows the grammar developer to specify a daughter as the *key*: the key daughter is checked first. The value of the path `*key-daughter-path*` in the daughter structure of the grammar rule should be set to the value of `*key-daughter-type*` when that daughter is the key. By default, the value of `*key-daughter-path*` is (KEY-ARG) and the value of `*key-daughter-type*` is $+$. So, for instance, if the daughters in the rule are described as a list which is the value of the feature ARGS and the first daughter is the key, the value of (ARGS FIRST KEY-ARG) should be $+$. A rule is not required to have a specified key and the `*key-daughter-path*` need not be present in this case. Specifying a key will never affect the result of parsing or generation.

8.3.3 Avoiding copying and tree reconstruction

The HPSG framework is usually described in such a way that phrases are complete trees: that is, a phrase TFS contains substructures which are also phrases. This leads to very large structures and computational inefficiency. However, HPSG also has a locality principle, which means that it is not possible for a phrase to access substructures that are daughters of its immediate daughters. Any such information has to be carried up explicitly.

The locality principle therefore guarantees that it is possible to remove the daughters of a rule after constructing the mother without affecting the result. The LKB system allows the grammar writer to supply a list of features which will not be passed from daughter to mother when parsing via the parameter `*deleted-daughter-features*`. In fact, in the samples grammars we have looked at in this book, `*deleted-daughter-features*` could have been set to `(ARGS)` to improve efficiency. This feature must be used with care, since if the grammar writer has not obeyed the locality principle with respect to the specified features, setting this parameter will cause different results to be obtained.[57]

8.3.4 Packing

Simple context free grammars can be parsed in cubic time because not all derivations need be explicitly computed. For instance, consider a sentence like:

(8.65) The jack above the ace next to the queen is red.

The NP, *the jack above the ace next to the queen* could be analysed as having either of the following two structures:

(8.66) the ((jack (above the ace)) (next to the queen))

(8.67) the (jack (above (the (ace (next to the queen)))))

These have different semantics: in the first, the jack is above the ace and also next to the queen, while in the second, it is the ace that is next to the queen. However, if these structures were being accounted for by simple CFG rules, both would simply be an NP, and there could be no difference in how they subsequently interacted with the rest of the grammar. So a simple CFG parser can assume that there is only one edge for the NP, even though it can be derived in multiple different ways. In general, in a chart parser for simple CFGs, once an edge between two nodes n and m has been constructed, any subsequent edge with the same category is simply recorded but is not involved in further processing, because it would simply be duplicating the results from the first edge. This allows cubic time parsing, even for grammars where an exponential number of derivations is possible for a sentence.

With unification-based grammars, there are several potential problems in applying this technique. The first concerns the fact we are dealing with TFSs rather than simply atomic category symbols. This

[57]In many ways it would be more logical to define grammar rules with a separate mother feature, which is supported by the LKB system, but this doesn't fit in with the way that HPSG is generally described.

means that for maximal packing, the parser has to check for subsumption rather than equality. An edge which is more specific than an existing edge spanning the same nodes cannot result in any new parses, but further complexity arises because a new edge might be more general than an existing edge, in which case the new edge has to be checked but the existing edge could be frozen. This is discussed in detail by Oepen and Carroll (2000b).

The other problems concern the way the grammars are used. In the grammars we have seen, the structures that are produced for a node carry around information about their derivation, because of the ARGS. In terms of a CFG, it is as though the categories for the NPs above were not simply NP, but the following structures:

1. (NP (Det the) (N (N (N jack) (PP (P above) (NP (Det the) (N ace)))) (PP (P next-to) (NP (Det the) (N queen)))))
2. (NP (Det the) (N (N jack) (PP (P above) (NP (Det the) (N (N ace) (PP (P next-to) (NP (Det the) (N queen))))))))

Obviously these two structures are not the same. However, as we saw in the previous section, the locality principle of HPSG guarantees that the derivation information can be ignored because no required information is in the ARGS without also being coindexed with the rest of the phrase. Thus we can ignore information under ARGS when packing.

A related problem is that in many grammars, the semantics is built up in parallel with the syntax. This is convenient, but means that the two TFSs for 8.66 and 8.67 will be different. However, in the grammars used in this book and in the ERG, there is also a semantic locality principle, which guarantees that the internal structure of the list of elementary predications cannot be relevant to subsequent composition operations. This means that the semantics can be ignored too, for the purposes of packing, as long as we are prepared to reconstruct the TFSs when it is necessary to pass the semantic structures to another system component.

The LKB system contains code that implements packing, but it is not used by default. For further details and experimental results on the ERG, see Oepen and Carroll (2000b).

8.4 Irregular morphology

Irregular morphology may be specified in association with a particular lexical rule, in the way we saw in with `g6morph` in §5.2.1, but it is often more convenient to have a separate file for irregular morphemes. In the LKB, this file has to take the form of a Lisp string (i.e., it begins and ends with `")`[58] containing irregular form entries, one per line, each consisting of a triplet:

1. inflected form
2. rule specification
3. stem

For example:

```
"
dreamt PAST-V_IRULE dream
fell PAST-V_IRULE fall
felt PAST-V_IRULE feel
gave PAST-V_IRULE give
"
```

The file is loaded with the command `load-irregular-spellings` in the script file. For instance:

```
(load-irregular-spellings
   (lkb-pathname (this-directory) "irregs.tab"))
```

The interpretation of irregular forms is similar to the operation of the regular morphology: that is, the irregular form is assumed to correspond to the spelling obtained when the specified rule is applied to a lexical entry with an orthography corresponding to the stem. The rule specification should be the identifier for a morphology rule.

The default operation of the irregular morphology system is one case where there may be an asymmetry in system behaviour between parsing and generation: when parsing, a regularly derived form will be accepted even if there is also an irregular form if the value of the LKB parameter `*irregular-forms-only-p*` (see §9.2.3) is `nil`. Thus, for example, *gived* will be accepted as well as *gave*, and *dreamed* as well as *dreamt*. If the value of `*irregular-forms-only-p*` is `t`, an irregular form will block acceptance of a regular form, so only *gave* would be accepted. Generation will always produce the irregular form if there is one: with the current version of the LKB it will simply produce the first form given in the irregulars file for a particular rule even if there are alternative spellings. For instance, assume the following is included in the irregulars file:

```
dreamed PAST-V_IRULE dream
dreamt PAST-V_IRULE dream
```

It is a good idea to include both forms for parsing if the parameter `*irregular-forms-only-p*` is true, because it allows both variants to

[58]This rather clunky format is used for compatibility with other systems.

be accepted, but for generation, only *dreamed* would be produced.

8.5 Multiword lexemes

The LKB system allows lexical items to be specified which have a value for their orthography which is a list of more than one string. These items are treated as multiword lexemes and the parser checks that all the strings are present before putting the lexical entry on the chart.

Multiword lexemes may have at most one affixation site. This is specified on a per-entry basis, via the user-definable function `find-infl-pos` (see §9.3). By default, this allows affixation on the rightmost element: e.g., *ice creams*. It can be defined in a more complex way, for instance to allow *attorneys general* or *kicked the bucket* to be treated as multiword lexemes.

This component of the LKB is under very active development at the time of writing, so please check the website for possible updates.

8.6 Parse ranking

The current LKB parser supports a simple mechanism for ordering parses and for returning the first parse only. The application of this mechanism is controlled by the variable `*first-only-p*` which may be set interactively, via the **Options / Set options** menu command. The weighting is controlled by two functions, `rule-priority` and `lex-priority`, which may be defined in the grammar-specific file `user-fns.lsp`, if desired. Since the mechanism is under active development, I don't propose to document it in detail here, but I suggest that anyone who wishes to experiment with this capability looks at the definitions of `rule-priority` and `lex-priority` in the `user-fns` file for the LinGO ERG. However, please check the LKB website for updates.

8.7 Leaf types

The notion of a leaf type is defined for efficiency in grammar loading. A leaf type is a type which is not required for the valid definition or expansion of any other type or type constraint. Specifically this means that a leaf type must meet the following criteria:

1. A leaf type has no daughters.
2. A leaf type may not introduce any new features on its constraint.
3. A leaf type may not be used in the constraint of another type.
4. A leaf type only has one 'real' parent type — it may have one or more template parents (see 9.2.1).

Under these conditions, much more efficient type checking is possible, and the type constraint description can be expanded on demand rather than being expanded when the type file is read in.

In the current version of the LinGO ERG, most of the terminal lexical types (i.e., those from which lexical entries inherit directly) are leaf types, as are most of the relation types. The latter class is particularly important, since it means that a lexicon can be of indefinite size and expanded on demand, while still using distinct types to represent semantic relations.

Various checks are performed on leaf types to ensure they meet the above criteria. However the tests cannot be completely comprehensive if the efficiency benefits of leaf types are to be maintained. If a type is treated as a leaf type when it does not fulfill the criteria above, unexpected results will occur. Thus the leaf type facility has the potential for causing problems which are difficult to diagnose and it should therefore be used with caution.

8.8 Caches

The cache functionality is provided to speed up reloading a grammar when leaf types or the lexicon have not changed.

Lexicon cache Whenever a lexicon is read into the LKB, it is stored in an external file rather than in memory. Lexical entries are pulled in from this file as required. The caching facility saves this file and an associated index file so that they do not need to be recreated when the grammar is read in again if the lexicon has not changed. The location of these files is set by the `user-fns.lsp` function `set-temporary-lexicon-filenames`. The file names are stored in the parameters `*psorts-temp-file*` and `*psorts-temp-index-file*` (see §9.3.1). The default function uses the function `lkb-tmp-dir` to find a directory and names the files `templex` and `templex-index` respectively. A more complex version of `set-temporary-lexicon-filenames` is given in the `user-fns.lsp` file in the LinGO ERG: this uses a parameter set by `Version.lsp` to name the files.

The script command `read-cached-lex-if-available` (see §8.2.1), takes a file or list of files as arguments. If a cached lexicon is available in the locations specified by `*psorts-temp-file*` and `*psorts-temp-index-file*`, and is more recent than the file(s) given as arguments to the function, then it will be used instead of reading in the specified files from scratch. The code does not check to see whether the cached lexicon was created from the specified files, and there are other ways in which the system can be fooled. Thus you should definitely not use this unless you have a sufficiently large lexicon that the reload time is annoying.

Leaf type cache A similar caching mechanism is provided for leaf types (§8.7). The parameter storing the name of the file is `*leaf-temp-file*` but like the lexicon cache files this is set by the function `set-temporary-lexicon-filenames`.

8.9 Using emacs with the LKB system

We recommend the use of emacs (either gnuemacs or XEmacs) with the LKB for editing the grammar files. Details of how to obtain emacs and set it up for use with the LKB are on the LKB website. The LKB menu command **Show source** requires that emacs be used as the editor. The emacs interface also puts some LKB menu commands on the top menu bar of the emacs window, for convenience, and a TDL editing mode is available. The LKB website also contains a brief guide to emacs commands for beginners.

8.10 YADU

The structures used in the LKB system are not actually ordinary TFSs, but typed default feature structures (TDFSs). So far in this book I have assumed non-default structures and monotonic unification, but this is actually just a special case of typed default unification. The full description of TDFSs and default unification is given in Lascarides and Copestake (1999) (henceforth L+C). The following notes are intended to supplement that paper.

Default information is introduced into the description language by `/`, followed by an indication of the persistence of the default. In terms of the current implementation, the supported distinctions in persistence are between defaults which are fully persistent and those which become non-default when an entry TFS is constructed (referred to as description persistence). The value of the description persistence indicator is given by the parameter `*description-persistence*` — the default is `l` (i.e., the lowercase letter l, standing for 'lexical', not the numeral 1).

The modification to the BNF given for the TDL syntax in §4.4.6 is as follows:

Conjunction → *DefTerm* | *DefTerm* `&` *Conjunction*
DefTerm → *Term* | *Term* `/`*identifier Term* | `/`*identifier Term*

It is not legal to have a default inside a default.

As an example, the following definition of **verb** makes the features PAST, PASTP and PASSP indefeasibly coindexed, and PASTP and PASSP defeasibly coindexed. The definition for **regverb** makes the value of PAST be ed, by default.

```
verb := *top* &
[ PAST *top* /l #pp,
  PASTP #p /l #pp,
  PASSP #p /l #pp ].

regverb := verb &
[ PAST /l "ed" ] .
```

Any entry using these types will be completely indefeasible, since the defaults have description persistence. Further examples of the use of defaults, following the examples in L+C, are given in the files in data/yadu_test, distributed with the LKB.

You should note that the description language specifies dual TFSs from which BasicTDFSs are constructed as defined in §3.5 of L+C. See also the discussion of the encoding of VALP in §4.2 of L+C.

If you view a feature structure which contains defaults, you will see three parts. The first is the indefeasible structure, the second the 'winning' defeasible structure, and the third the tail. (Unfortunately this is very verbose: a more concise representation will be implemented at some point.)

Notice that the current implementation assumes that defaults are only relevant with respect to an inheritance hierarchy: default constraints which are specified on types other than the root node of a structure are ignored when expanding the feature structure.

8.11 MRS

MRS (minimal recursion semantics) is a framework for computational semantics which is intended to simplify the design of algorithms for generation and semantic transfer, and to allow the representation of underspecified scope while being fully integrated with a typed feature structure representation. It is decribed in detail in Copestake et al (1999). The LKB system supports MRS in providing various procedures for processing MRS structures (for printing, checking correctness, scoping, translation to other formalisms etc). The LKB generation component (described in §5.4.4 and §8.12) assumes an MRS style of representation. The LinGO grammar produces MRS structures, but the example grammars discussed in this book produce a form of simplified MRS which is not adequate to support scope resolution, though it can be used for generation. The MRS menu commands are described in §6.1.5 and §6.4. Further documentation will become available via the LKB webpage.

8.12 Generation

The generator requires a grammar which is using MRS or a relatively similar formalism The generator is described in Carroll et al (1999). A few remarks may be helpful to supplement §5.4.4.

1. The grammars that use MRS contain a file defining some necessary global parameters. This is called `mrsglobals.lsp` for the example grammars and `mrsglobals-eng.lsp` for the LinGO ERG.
2. Before generating, the lexicon and lexical and grammar rules must be indexed. The function which does this, `index-for-generator`, can be run with the **Index** command from the **Generate** menu, or it can be directly run from the script file as in §5.4.4.
3. If the grammar uses lexical entries which do not have any relations, warning messages will be issued by the indexing process. A set of grammar-specific heuristics can be defined which will determine which such lexical entries might be required for a given semantics. These are loaded via the **Load Heuristics** command on the **Generate** menu. If no such heuristics are defined, the system attempts to use all null semantics items, which can be very slow.
4. The generator supports a special treatment of intersective modification, discussed in Carroll et al (1999). The LKB parameters which control this are listed in §9.2.5.
5. There is a spellout component to the generator, which currently simply looks after the *a/an* alternatives.
6. A LKB parameter, `*duplicate-lex-ids*`, is used to specify lexical entry identifiers which should be ignored, because they are simply alternative spellings of another lexical entry. This mechanism is expected to be refined in the near future.

8.13 Testing and Diagnosis

The batch parsing support which is distributed with the LKB is very simple. For extensive grammar development, much more sophisticated tools are available through the [incr tsdb()] package that can be used with the LKB. The following description is by Stephan Oepen (to whom all comments should be directed: see the website below for contact information).

The [incr tsdb()] package combines the following components and modules:

- test and reference data stored with annotations in a structured database; annotations can range from minimal information (unique test item identifier, item origin, length et al.) to fine-grained linguistic classifications (e.g., regarding grammaticality and linguistic phenomena presented in an item) as represented by the TSNLP test suites (Oepen et al, 1997).

- tools to browse the available data, identify suitable subsets and feed them through the analysis component of a processing system like the LKB or PAGE;
- the ability to gather a multitude of precise and fine-grained system performance measures (like the number of readings obtained per test item, various time and memory usage metrics, ambiguity and non-determinism parameters, and salient properties of the result structures) and store them as a *competence and performance profile* in the database;
- graphical facilities to inspect the resulting profiles, analyze system competence (i.e., grammatical coverage and overgeneration) and performance at highly variable granularities, aggregate, correlate, and visualize the data, and compare profiles obtained from previous grammar or system versions or other platforms.
- a universal pronounciation rule: *tee ess dee bee plus plus* is the name of the game.

Please see:

`http://www.coli.uni-sb.de/itsdb/`

for links to more information about the [incr tsdb()] package and details of how to get it.

8.14 Parse tree labels

The grammars in this book use a simple node labelling scheme in parse trees, as described in 4.5.7. However, if `*simple-tree-display*` is set to `nil` rather than `t`, the parse tree labelling is more complex, and this more complex scheme is outlined here.

There are two classes of templates: *label* and *meta* structures (see §9.1.3 for the parameters). Each label TFS specifies a single string at a fixed path in its TFS: LABEL-NAME in the ERG and textbook grammars. For instance, the following is a label from the textbook grammar.

```
np :=  label &
 [ SYN [ HEAD noun,
         SPR < > ],
   LABEL-NAME "NP" ].
```

Meta templates are used for things such as `/` (e.g., to produce the label `S/NP`). If meta templates are specified, each meta template TFS must specify a prefix string and a suffix string (by default, at the paths META-PREFIX and META-SUFFIX). For instance, the following is specified as a meta template in the textbook grammar:

```
slash := meta &
  [ SYN [ GAP [ LIST ne-list ] ],
    META-PREFIX "/",
    META-SUFFIX "" ].
```

To calculate the label for a node, the label templates are first checked to find a match. Matching is tested by unification of the template feature structure, excluding the label path, with the TFS on the parse tree node. For instance, if a parse tree node had the following description:

```
[ SYN [ HEAD *top*,
        SPR < > ]].
```

it would unify with

```
 [ SYN [ HEAD noun,
         SPR < > ]].
```

and could thus be labelled `NP`, given the structure above. There is a parameter, `*label-fs-path*`, which allows templates to be checked on only the substructure of the node TFS which follows that path, but this parameter is set to the empty path in the textbook grammar.

If meta templates are specified, the TFS at the end of the path specified by the parameter `*recursive-path*` ((SYN GAP LIST FIRST) in the textbook grammar) is checked to see whether it matches a meta template. If it does, the meta structure is checked against the label templates (the parameter `*local-path*` allows them to be checked against a substructure of the meta structure). The final node label is constructed as the concatenation of the first label name, the meta prefix, the label of the meta structure, and the meta suffix.

8.15 Linking the LKB to other systems

The LKB has a non-graphical user interface which can be used by people who for whatever reason can't run the graphical user interface. Apart from the graphical user interface, the LKB system mostly uses ANSI-standard Common Lisp. The tty interface is also useful when using the LKB in conjunction with another system. I won't list the tty commands here (though see 6.1.4 for `do-parse-tty`, which is the most generally used). Some information on the tty commands is given on the LKB website.

The easiest way to hook up the output of the LKB parser to another system is to use a semantic representation compatible with MRS. A range of different output options are available for MRS, and writing alternative output formats is reasonably straightforward. Code which links MRS output to a canonical form representation suitable for use with theorem provers is in the LKB source directory `tproving`.

A lower-level interface is available via user-defined functions which are specified as the value of the parameter `*do-something-with-parse*`. This allows access to the parser output in the form of TFSs, which can be manipulated via the interface functions defined in the LKB source code file `mrs/lkbmrs.lisp`.

Further details of interfacing to the LKB will be given on the website, as time permits!

9

Details of system parameters

Various parameters control the operation of the LKB and some feature and type names are regarded as special in various ways. The system provides default settings but typically, each grammar will have its own files which set these parameters, redefining some of the default settings, though a set of closely related grammars may share parameter files. Each script must load the grammar-specific parameter files before loading any of the other grammar files (see §4.5.1 and §8.2). Other parameters control aspects of the system which are not grammar specific, but which concern things such as font size, which are more a matter of an individual user's preferences. This class of parameters can mostly be set via the **Options** menu (described in §6.1.9). Changing parameters in the **Options** menu results in a file being automatically generated with the preferences: this shouldn't be manually edited, since any changes may be overridden. There's nothing to prevent global variables which affect user preferences being specified in the grammar-specific globals files, but it is better to avoid doing this, since it could be confusing.

There are also some functions which can be (re)defined by the grammar writer to control some aspects of system behavior.

This chapter describes all the parameters and functions that the LKB system allows the user/grammar writer to set, including both grammar specific and user preference parameters. We sometimes refer to the parameters as *globals*, since they correspond to Common Lisp global variables. Note that the parameters in the grammar-specific globals files are all specified as either:

```
(def-lkb-parameter global-variable value comment)
```

or

```
(defparameter global-variable value comment)
```

where the comment is optional. Thus only a very basic knowledge of

Common Lisp is required to edit a globals file: the main thing to remember is that symbols have to be preceded by a quote, for instance:

```
(def-lkb-parameter *string-type* 'string)
```

See §4.5.2 for a lightening introduction to Common Lisp syntax.

The descriptions below give the default values for the variables and functions (as they are set in the LKB source files `main/globals` and `main/user-fns`). The description of the global variables is divided into sections based on the function of the variables: these distinctions do not correspond to any difference in the implementation with the exception that the globals which can be set via the **Options**/ **Set options** menu are nearly all ones that are specified as being grammar independent. (Not all grammar independent variables are available under **Options**, since some are too complex to set interactively, or are rarely used.)

To set a value interactively, use the **Options**/ **Set options** command on the parameter.

9.1 Grammar independent global variables

9.1.1 System behaviour

***gc-before-reload*, nil** — This boolean parameter controls whether or not a full garbage collection is triggered before a grammar is reloaded. It is best set to `t` for large grammars, to avoid image size increasing, but it is `nil` by default, since it slows down reloading.

***sense-unif-fn*, nil** — If set, this must correspond to a function. See `make-sense-unifications` in §9.3, below.

***maximum-number-of-edges*, 500** — A limit on the number of edges that can be created in a chart, to avoid runaway grammars taking over multi-user machines. If this limit is exceeded, the following error message is generated:

```
Error: Probable runaway rule: parse/generate aborted
(see documentation of *maximum-number-of-edges*)
```

When parsing a short sentence with a small grammar, this message is likely to indicate a rule which is applying circularly (that is, applying to its own output in a way that will not terminate). But this value must be increased to at least 2000 for large scale grammars with a lot of ambiguity such as the LinGO ERG. May be set interactively.

***maximum-number-of-tasks*, 50000** — A limit on the number of tasks that can be put on the agenda. This limit is only likely to be exceeded because of an error such as a circularly applicable rule.

***chart-limit*, 100** — The limit on the number of words (actually tokens as constructed by the tokenizer function) in a sentence which can be parsed. Whether the system can actually parse sentences of this length depends on the grammar!

***maximal-lex-rule-applications*, 7** — The number of lexical rule applications which may be made before it is assumed that some rules are applying circularly and the system signals an error.

***display-type-hierarchy-on-load*, t** — A boolean variable, which controls whether the type hierarchy is displayed on loading or not. This must be set to nil for grammars with large numbers of types, because the type hierarchy display becomes too slow (and if the type hierarchy involves much multiple inheritance, the results are not very readable anyway). May be changed interactively.

9.1.2 Display parameters

***feature-ordering*, nil** — A list which is interpreted as a partial order of features for setting the display ordering when TFSs are displayed or printed. See §6.3.

***show-morphology*, t** — A boolean variable. If set, the morphological derivations are shown in parse trees (see §6.5). May be set interactively.

***show-lex-rules*, t** — A boolean variable. If set, applications of lexical rules are shown in parse trees (see §6.5). May be set interactively.

***simple-tree-display*, nil** — A boolean variable which can be set in order to use the simple node labelling scheme in parse trees. See §4.5.7 and §8.14.

***substantive-roots-p*, nil** — A boolean variable which can be set to allow the structures constructed when checking the start conditions to be regarded as real edges for the purposes of chart display.

***display-type-hierarchy-on-load*, t** — see §9.1.1 above. May be set interactively.

***parse-tree-font-size*, 12** — The font size for parse tree nodes. May be set interactively.

***fs-type-font-size*, 12** — The font size for nodes in AVM windows. May be set interactively.

***fs-title-font-size*, 12** — The font size for titles in AVM windows. May be set interactively.

***type-tree-font-size*, 12** — The font size for the nodes in the type hierarchy. May be set interactively.

***dialog-font-size*, 12** — The font size used in dialogue windows. May be set interactively.

***maximum-list-pane-items*, 50** — The maximum number of rules, lexical entries etc that will be offered as choices in menus that allow selection of such entities.

9.1.3 Parse tree node labels

These parameters are only used when `*simple-tree-display*` is `nil`. For details of how this operates, see §8.14.

***label-path*, (LABEL-NAME)** — The path where the name string is stored.

***prefix-path*, (META-PREFIX)** — The path for the meta prefix symbol.

***suffix-path*, (META-SUFFIX)** — The path for the meta suffix symbol.

***recursive-path*, (NON-LOCAL SLASH LIST FIRST)** — The path for the recursive category.

***local-path*, (LOCAL)** — The path inside the node to be unified with the recursive node.

***label-fs-path*, (SYNSEM)** — The path inside the node to be unified with the label node.

***label-template-type*, label** — The type for all label templates.

9.1.4 Defaults

See §8.10.

***description-persistence*, l** — The symbol used to indicate that a default should be made hard (if possible) when an entry is expanded into a complete TFS.

9.2 Grammar specific parameters

9.2.1 Type hierarchy

toptype*, top** — This should be set to the value of the top type: ***top in the example grammars. See §3.2.

***string-type*, string** — The name of the type which is special, in that all Lisp strings are recognised as valid subtypes of it. See §3.2.

9.2.2 Orthography and lists

***orth-path*, (orth lst)** — The path into a sign, specified as a list of features, which leads to the orthographic specification. See §4.1 and also the functions `make-sense-unifications` and `make-orth-tdfs` in

§9.3.

***list-head*, (hd)** — The path for the first element of a list. See §4.4.2.

***list-tail*, (tl)** — The path for the rest of a list. See §4.4.2.

list-type*, *list — The type of a list. See §4.4.2.

empty-list-type*, *null — The type of an empty list — it must be a subtype of ***list-type***. See §4.4.2.

diff-list-type*, *diff-list — The type of a difference list (see §4.4.2).

***diff-list-list*, list** — The feature for the list portion of a difference list. See §4.4.2.

***diff-list-last*, last** — The feature for the last element of a difference list. See §4.4.2.

9.2.3 Morphology and parsing

***lex-rule-suffix*, nil** — If set, this is appended to the string associated with an irregular form in the irregs file in order to construct the appropriate inflectional rule name. See §8.4. Required for PAGE compatibility in the LinGO ERG.

***mother-feature*, 0** — The feature specifying the mother in a rule. May be NIL (is nil with all grammars discussed in this book).

***start-symbol*, sign** — A type which specifies the type of any valid parse. Can be used to allow relatively complex start conditions. See §4.5.6. Unlike most grammar specific parameters, this can be set interactively to allow a switch between parsing fragments and only allowing full sentences, for instance.

***deleted-daughter-features*, nil** — A list of features which will not be passed from daughter to mother when parsing. This should be set if efficiency is a consideration in order to avoid copying parts of the TFS that can never be (directly) referenced from rules pertaining to higher nodes in the parse tree. This will include daughter features in HPSG, since it is never the case that a structure can be directly selected on the basis of its daughters. See §8.3.3.

***key-daughter-path*, (key-arg)** — A path into a daughter in a rule which should have its value set to `*key-daughter-type*` if that daughter is to be treated as the key. See §8.3.2.

***key-daughter-type*, +** — . See above.

***check-paths*, nil** — An association list in which the keys are feature paths that often fail — these are checked first before attempting unification to improve efficiency. See §8.3.1. The value of this parameter could be set in the globals file, but since the list of paths is automatically generated, it is normally kept in a distinct file. The parameter should ideally be set to `nil` in the globals file for grammars which do not use check paths, in case the grammar is read in after a previous grammar which does set the paths.

***check-path-count*, 30** — The number of check paths which are actually used when parsing (see above): set empirically to give maximum performance.

***irregular-forms-only-p*, nil** — If set, the parser will not invoke the morphological analyzer on a form which has an irregular spelling. This prevents the system analyzing words such as *mouses*. Note that this means that if the grammar writer wants to be able to analyze *dreamed* as well as *dreamt*, both entries have to be in the irregular morphology file. Also note that we currently assume (incorrectly) that spelling is not affected by sense. For instance, the system cannot handle the usage where the plural of *mouse* is *mouses* when referring to computers. Note that this flag does not affect generation, which always treats an irregular form as blocking a regular spelling. See §8.4.

***first-only-p*, nil** — If set, only one parse will be returned, where any preferences must be defined as specified in §8.6. May be set interactively.

9.2.4 Compare parameters

***discriminant-path*, (synsem local cont key)** — A path used by the Compare display to identify a useful discriminating position in a structure — see §6.1.4.

9.2.5 Generator parameters

***gen-first-only-p*, nil** — If set only one realization will be returned, where any preferences must be defined as specified in §8.6. May be set interactively.

***semantics-index-path*, (synsem local cont index)** — The path used by the generator to index the chart. See §5.4.4.

***intersective-rule-names*, (adjh_i nadj_i hadj_i_uns)** — The names of rules that introduce intersective modifiers. Used by the generator to improve efficiency. Default value is appropriate for the LinGO grammar. It should be set to NIL for grammars where the intersective modifier rules do not meet the necessary conditions for adjunction. See §8.12 and also the function `intersective-modifier-dag-p` in §9.3.

***duplicate-lex-ids*, (an)** — Used in grammars which do not have any way of representing alternative spellings, this is a list of lexical identifiers that should be ignored by the generator. (The *a/an* alternatives are chosen by a post-generation spelling realization step.) See §8.12.

9.2.6 MRS parameters

I will not go through most of the MRS parameters here because they are currently frequently being revised. Documentation will be available on the website. They are stored in a separate file from the other globals (the file is called `mrsglobals.lsp` for the grammars in this book and `mrsglobals-eng.lisp` for the LinGO grammar).

mrs::*scoping-call-limit*, 10000 — Controls the search space for scoping.

9.3 User definable functions

make-sense-unifications — If this function is defined, it takes three arguments so that the orthographic form of a lexical entry, its id and the language are recorded in an appropriate place in the TFS. The value of `*sense-unif-fn*` must be set to this function, if it is defined. The function is not defined by default.

The idea is that this function can be used to specify paths (such as `ORTH.LIST.FIRST`) which will have their values set to the orthographic form of a lexical entry. This allows the lexicon files to be more succinct. It is assumed to be exactly equivalent to specifying that the paths take particular atomic values in the lexical entry. For example, instead of writing:

```
teacher_1 := noun-lxm &
[ ORTH.LIST.FIRST "teacher",
  SEM.RELS.LIST.FIRST.PRED teacher1_rel] .
```

the function could be defined so it was only necessary to write:

```
teacher_1 := noun-lxm.
```

since the value of the orthography string and the semantic relation name are predictable from the identifying material.

make-orth-tdfs — A function used by the parser which takes a string and returns a TFS which corresponds to the orthographic part of a sign corresponding to that string. The default version assumes that the string may have spaces, and that the TFS will contain a list representation with each element corresponding to one word (i.e., sequence of characters separated by a space). For instance, the string `"ice cream"` would give rise to the structure

```
[ ORTH [ LST [ HD ice
               TL [ HD cream ]]]]
```

establish-linear-precedence — A function which takes a rule TFS and returns the top level features in a list structures so that the ordering corresponds to mother, daughter 1, daughter 2 ... daughter n. The default version assumes that the daughters of a rule are indicated by the features 1 through n. See §4.1.

spelling-change-rule-p — A function which takes a rule structure and checks whether the rule has an effect on spelling. It is used to prevent the parser trying to apply a rule which affects spelling and which ought therefore only be applied by the morphology. The current value of this function checks for the value of NEEDS-AFFIX being **true**. If this matches a rule, the parser will not attempt to apply this rule. (Note that the function will return false if the value of NEEDS-AFFIX is anything other than **true**, since the test is equality, not unifiability.) See §5.2.1.

redundancy-rule-p — Takes a rule as input and checks whether it is a redundancy rule, defined as one which is only used in descriptions and is not intended to be applied productively. See §5.2.1. For instance, the prefix *step-*, as in *stepfather*, has a regular meaning, but only applies to a small, fixed set of words. A redundancy rule for *step-* prefixation can be specified to capture the regularity and avoid redundancy, but it can only be used in the lexical descriptions of *stepfather* etc, and not applied productively. (There is nothing to prevent productive lexical rules being used in descriptions.)

The default value of this function checks for the value of PRODUCTIVE being **false**. If this matches a rule, the parser will not attempt to apply that rule. (Note that the function will return false if the value of PRODUCTIVE is anything other than **false**, since the test is equality, not unifiability.)

preprocess-sentence-string — The function takes an input string and preprocesses it for the parser. The result of this function should be a single string which is passed to a simple word identifier which splits the string into words (defined as things with a space between them). So minimally this function could be simply return its input string. However, by default some more complex processing is carried out here, in order to strip punctuation and separate *'s*. Thus, in effect, this function controls the tokenizer: see §4.1.

find-infl-poss — This function is called when a lexical item is read in, but only for lexical items which have more than one item in their orth value (i.e., multiword lexemes). It must be defined to take three arguments: a set of unifications (in the internal data structures), an orthography value (a list of strings) and a sense identifier. It should return an integer, indicating which element in a multi-word lexeme may be inflected (counting left to right, leftmost is 1) or `nil`, which indicates that no item may be inflected. The default value for the function allows inflection on the rightmost element. See §8.5.

hide-in-type-hierarchy-p — Can be defined so that certain types are not shown when a hierarchy is displayed (useful for hierarchies where there are a very large number of similar leaf types, for instance representing semantic relations).

rule-priority — See §8.6. The default function assigns a priority of 1 to all rules.

lex-priority — See §8.6. The default function assigns a priority of 1 to all lexical items.

intersective-modifier-dag-p — Used by the generator to test whether a structure is an intersective modifier. Default value is applicable for the LinGO grammar. It should be set to NIL in grammars where intersective modifiers do not meet the conditions the generator requires for adjunction. See also the parameter `*intersective-rule-names*` in §9.2.5.

9.3.1 System files

There are two user definable functions which control two system files. The file names are associated with two global variables — these are initially set to `nil` and are then instantiated by the functions. The global variables are described below, but should not be changed by the user. The functions which instantiate them may need to be changed for different systems.

lkb-tmp-dir — This function attempts to find a sensible directory for the temporary files needed by the LKB. The default value for this on Unix is a directory `tmp` in the user's home directory: on a Macintosh it is `Macintosh HD:tmp`. The function should be redefined as necessary to give a valid path. It is currently only called by the function `set-temporary-lexicon-filenames` (below).

set-temporary-lexicon-filenames — This function is called in order to set the temporary files. It uses lkb-tmp-dir, as defined above. It is useful to change the file names in this function if one is working with multiple grammars and using caching to ensure that the lexicon file associated with a grammar has a unique name which avoids overriding another lexicon (see §8.8).

The files are defined by the following variables:

psorts-temp-file This file is constructed by the system and used to store the unexpanded lexical entries, in order to save memory. Once a lexical entry is used, it will be cached until either a new

lexicon is read in, or until the **Tidy up** command is used (§6.1.8). If the temporary lexicon file is deleted or modified while the LKB is running, it will not be possible to correctly access lexical entries. The file is retained after the LKB is exited so that it may be reused if the lexicon has not been modified (see §8.8, §8.2.1 and the description of `*psorts-temp-index-file*`, below).
The pathname is actually specified as:

```
(make-pathname :name "templex"
               :directory (lkb-tmp-dir))
```

psorts-temp-index-file This file is used to store an index for the temporary lexicon file. If the option is taken to read in a cached lexicon (see §8.8 and §8.2.1), then the lexicon index is reconstructed from this file. If this file has been deleted, or is apparently outdated, the lexicon will be reconstructed from the source file.

leaf-temp-file This file is used to store cached leaf types.