Grammar Writer's Workbench

Ronald M. Kaplan and John T. Maxwell III

Version 3.1, November, 1996

Table of Contents

References

Appendix: Editing Text with TEdit

Introduction

The LFG Grammar-writer's Workbench is a computational environment that assists in writing and debugging Lexical Functional Grammars (Kaplan & Bresnan, 1982). It provides linguists with a facility for writing syntactic rules, lexical entries, and simple morphological rules, and for testing and editing them. The system applies these specifications to sentences or other strings and then provides the following analytic information:

Constituent-structures — whether or not they have valid f-structures.

The chart — containing all complete or incomplete bracketings of the input string that the grammar allows

Functional-structures — including display of inconsistencies, incompletenesses, and incoherencies

Functional-descriptions — the instantiated equations corresponding to particular c-structure nodes or particular f-structures

The system was originally written in the early 1980's, but it has evolved since then and also implements some of the later additions to the LFG formalism. It includes functional uncertainty (Kaplan & Zaenen, 1989b), functional precedence (Kaplan & Zaenen, 1989a), generalization over sets (Kaplan & Maxwell, 1988), and a rich notation for expressing c-structure patterns. The system allows correspondences between multiple levels of linguistic representation to be defined, as described by Kaplan (1987) and Halvorsen and Kaplan (1988). As a simple application of this capability, the system can display the properties of a semantic representation that is characterized and associated with a string by lexical and syntactic schemata. Parameterized macros and templates are supported so that common constituent and functional patterns can be expressed as independent generalizations.

The system provides a powerful interface for defining and manipulating linguistic rules and representations. After installing a collection of syntactic rules and lexical entries into the Grammar-writer's Workbench (henceforth GWB), you can see whether those items are sufficient to analyze sentences or phrases in the language in question. You can also easily mix and match different sets of linguistic specifications as you experiment with different versions of particular rules and lexical entries, whether you have written them yourself or they have been provided by other users of the system.

GWB also supports other activities that surround the business of grammar development. It can keep track of a number of editing tasks, essentially making available to the user an entire desktop's worth of memos, files, diagrams, versions of the paper being written about the grammar being developed, as well as the files of an entire working group. System output such as c-structures and f-structures can be copied into papers, and sentences from a paper can be used as input to the parser.

The Grammar Writer's Workbench is implemented in the Medley Lisp programming environement. This descendant of the original Xerox Interlisp-D system is now available from Venue Corporation. Medley runs on the original Xerox AI workstations and on a wide variety of Unix workstations (Sun, DEC, IBM, MIPS, HP, etc.) and on certain PC compatible platforms running MS-DOS. GWB and this document try to be self-contained, so that a linguist can successfully use the system without detailed knowledge of the programming environment. However, it may be helpful to obtain some familiarity with Medley's window, menu, mouse, and editing conventions from the more general Medley documentation.

The first chapter of this manual is tutorial in nature. It guides you through the process of entering a new grammar into GWB. It also describes how to use the system to parse sentences or other strings, to evaluate and edit your linguistic descriptions, and finally to save a grammar on a file for use in a later session. It assumes some familiarity with the Medley display facilities (window and mouse manipulation, etc), file management, and the TEdit text-editor, as well as the linguistic principles of Lexical Functional Grammar. All the editing windows used in GWB, for defining, displaying and modifying syntactic rules, lexical entries, morphological rules, templates, configurations, and other linguistic specifications, make use of the TEdit system. A brief description of basic TEdit operations is included as an Appendix at the end of this manual; it will be helpful to read that description if you are not already familiar with TEdit keyboard conventions.

If you follow through the first chapter, you will enter a grammar that can analyze simple sentences such as `The girl walks`. Not very impressive, but going through that procedure should give you the mechanics you need to enter grammars that are substantially more complex. Chapter II continues the tutorial with a more complicated grammar, showing in more detail how GWB can help you understand the properties of a given grammar and aid you in identifying and correcting its defects; this is the process of grammar debugging. The later chapters are less tutorial in nature. Chapter III gives formal characterizations of the notations used for the various linguistic specifications. Chapter IV provides reference material on the various windows and menus that are used to interact with the system.

Although GWB tries to insulate you from the details of the underlying Medley programming environment, various exceptional circumstances can arise that interrupt the normal operation of the system. For example, the system might run out of some critical resource such as stack space or memory, it might lose connections to important network services, or it might be unable to locate files, fonts, or printers that you specify. Typically in such situations a special "help" or "break" window will appear on the screen with some indication of the source of the problem. If you are familiar with Lisp and with the Medley system, you may know how to enter commands in this window that will repair the problem. Then, by typing `OK` followed by a carriage return, you can allow the troubled computation to proceed. If you do not know how to make the repair, you can simply type `STOP` and a carriage return. This will abort the broken computation, close the break window, and return you to the usual mode of operation described in the coming chapters.

We would like to acknowledge the contributions that Kris Halvorsen, Atty Mullins, Kelly Roach, and Michael Wescoat have made to the facilities that exist in this system. We are also indebted to Carol Kiparsky for drafting a precursor to this reference manual.

New in this release. . .

The Grammar Writer's Workbench continues to evolve, partly in response to independent developments in LFG theory and partly in response to linguistic and practical issues that emerge from use of the system. You can help improve the system and this manual by sending e-mail with bug reports, comments, and suggestions to kaplan@parc.xerox.com. This manual is revised at each release to clarify and expand the discussion of various points, and also so that it continues to describe the current state of the system. Here we call attention to the most significant changes in order to help users who are upgrading from previous versions.

A short section has been added to the manual describing how to exit from Medley. (Section I.8)

The rule, lexicon, and template menus contain a new item, `Find Symbol`, that helps you find occurrences of particular symols used as feature values.

The lexicon menu also contains a second new item, `Find Root`, that helps you keep track of dependencies among irregular morphological forms. (Sections I.3.4)

The display of all active templates now has the same format as the display of individual templates. (Section I.5.2)

The local name `%stem` has been given a special, built-in meaning as a reference to the headword of the lexical entry in which it is encountered. This can simplify the invokation of lexically-invoked templates. (Section III.4.1)

A new notation has been introduced for instantiated symbols so that they are clearly distinguished from semantic forms. (Section III.4.1)

Attributes can be declared as "nondistributive" so that they will not be distributed across the elements of a set. (Sections III.4.1 and III.6)

History of changes

Below is a record of changes that were introduced in previous releases of the manual and system:

A section has been added to the tutorial chapter that introduces the use of templates for sharing functional specifications, and a graphical display has been provided to show the "inheritance" relationships of template references. (Section I.5 and IV.3)

The Executive in GWB's initial configuration is now an Interlisp executive, so that all symbols will be in the Interlisp package by default. (Section I.6.4)

Words that closely resemble other words can be defined more conveniently, using the "like" feature in a Lexicon Window. (Section I.3.2)

The "bottom" element of the f-structure lattice is now seen as a bona fide entity in the f-structure ontology, representing the complete absence of information. As a consequence, certain constraints are no longer regarded as indeterminate and hence unacceptable. (Section II.3.3)

GWB recognizes that certain existential constraints and the completeness condition for certain structures are *globally* unsatisfiable because the required features are not instantiated anywhere on a given tree. (Section II.3.3)

Macro and template parameters can be marked as optional, so that a single definition can be used in more flexible ways. (Sections III.3.2 and especially III.5)

A c-structure rule can be referenced as a macro invocation, in which case that invocation is treated as a meta-category substitution. The invocation gives rise to a "phantom" node in that there is no distinct subtree corresponding to the rule-expansion. (Sections III.3.3 and IV.6)

Instantiated symbols provide a new kind of designator to describe unique syntactic feature values. (Section III.4.1)

Local names, symbols prefixed by %, can be used as designators so that it is convenient to refer to the same entity several times in the schemata associated with a particular category or lexical entry. (Section III.4.1)

New designators * and M* have been introduced to denote a matching c-structure node and its mother. This permits projections in addition to f-structure to be put in correspondence with c-structure nodes. (Section III.4.1)

The f-description language has been extended with a new predicate CAT that imposes constraints on the "category" of a functional or other abstract structure by restricting the categories of the c-structure nodes to which that structure corresponds. (Section III.4.2)

The SYMBOLSINLINE configuration parameter controls the display of f-structures. (Section III.6)

The display of f-structures and other projections has been improved: A single f-structure with alternative further projections now appears only once in the f-structure window rather than being displayed once for each of its further-projection alternatives. Also, subsidiary structures that are common to many different surrounding structures are only displayed once in full; other appearances are marked with a reference to the first occurrence. (Section II.1)

Typing CTRL−F in editing windows shows how macro and template invocations are expanded. (Sections III.3 and III.5 and Chapter IV)

Typing CTRL−S in Sentence Input and editing windows causes selected item to be "shown". (Section I.2.3 and Chapter IV)

The menu of the Sentence Input Window has two new items. *Print Structures* causes the structures resulting from the analysis of a set of sentences to be printed on a file where they can be accessed by other programs. *Show phantom nodes* toggles a switch that determines whether phantom nodes are to be interpreted as ordinary categories and thus made visible in the c-structure for easier debugging. (Section IV.6)

Chapter I

Entering a New Grammar

This chapter presents a tutorial introduction to the procedures for typing in syntactic rules, lexical entries, and morphological rules, the linguistic specifications that are necessary for sentence analysis. It also tells you how to create a new configuration, the repository for information about the rules, lexical entries and other information that make up a consistent sentence analysis environment. We begin with a brief discussion of the tedious but necessary matter of how you must refer to your linguistic specifications once you have entered them into the system.

1. Named linguistic specifications

The Grammar Writer's Workbench keeps track of and operates on a variety of different kinds of linguistic specifications: c-structure rules, lexical entries, tables for morphological analysis, different abbreviatory conventions, and other kinds of specifications that might be introduced as the theory and system evolve. At any one time the user may be working with (and the system may contain in its internal database) many different versions of a given rule or lexical entry. There may be several experimental versions of an S rule for English, for example, or a different S rule for each of several languages that are currently under investigation. The system must treat these different versions in different ways. Only one of them can be active when the system analyzes a sentence, you must be able to examine and edit one of them independently of the others, and you must be able to store them and retrieve them from different files.

Thus, at the outset, before you begin to write any grammatical descriptions, you must have some sense of the naming conventions by which you can refer to the linguistic specifications that you enter into the system. When instructing GWB to manipulate a particular rule or lexical entry, it is not sufficient merely to indicate the category or word you intend. You must identify the exact version you intend by using its unique *name*. Each linguistic specification in GWB has a name or handle by which you refer to it when calling for the system to manipulate it in some way. The name of an item is composed of three parts, its version, its language, and its item-identifier, and these are represented as an ordered triple enclosed in parentheses. Thus, the S rule in the Toy version of English has the name `(TOY ENGLISH S)` and this identifies a completely different rule than the names `(BASIC ENGLISH S)`, `(TOY DUTCH S)`, or `(TOY ENGLISH NP)`. Similarly, the lexical entry for `walk` in the standard English lexicon might have the unique name `(STANDARD ENGLISH walk)`.

All the rules or lexical entries in a given version-language can be identified by providing a list of their individual names, repeating the version and language separately for each:

```
(TOY ENGLISH S) (TOY ENGLISH VP) (TOY ENGLISH NP)
```

To avoid this redundancy, GWB allows a simple abbreviation for a set of names. The first item-identifier in a name can be followed by other item-identifers. The abbreviatory name is understood as specifying all of the items within the common version and language. With this convention the compact specification `(TOY ENGLISH S VP NP)` is equivalent to the longer name specification above.

An even more concise form is allowed if you want to refer to *all* the items in a particular version-language. These can be indicated by simply omitting the item-name entirely, as in `(TOY ENGLISH)`. If this form is used, for example, to indicate what rules should be stored on a file or which rules should be activated for use in sentence analysis, it denotes all the rules in that version-language that exist at the time the form is first used, and also any new rules in that version-language that are later introduced to the system. In other words, the exact set of items that a language-version pair refers to will change over time as you develop and extend your grammar.

You will use these naming conventions quite frequently as you enter, modify, and test your linguistic descriptions. Let's start by entering some syntactic rules for a simple version of English, called `TOY ENGLISH`.

2. Syntactic rules

2.1. Typing rules: punctuation and format

Rules are entered into GWB's internal database by typing in to the Rule Window. This is a TEdit window and all the common text-editing commands and operations described in the Medley Reference Manual and summarized here in the Appendix can be applied to modify the text of rules that you type into it. It differs from an ordinary window that you might use to edit a paper you are writing in two small but important ways. First, certain control characters have been assigned special LFG and Rule-related functions. These are entered by typing a character while holding down the `CTRL` key. Second, the normal TEdit menu, which has commands for getting and putting files and formatting text in various ways, does not appear on the screen when you click the left or middle mouse button in the black title-bar at the top of the window. Instead, you will see a menu with commands specialized to the rule-editing task.

You can start to enter a new rule simply by clicking in the Rule Window so that the blinking input caret moves to that window and then typing the text of the rule. But you can use the rule-window menu to get a little initial guidance in this process. If you left- or middle-click in the title bar of the rule window, you will see the menu with special rule commands:

```
Rule Menu
TEdit Menu
Find Attribute
Find Symbol
Find Template
Find Category
Edit Template
Edit Rule
```

The items `Find Attribute`, `Find Symbol`, `Find Template`, `Find Category`, and `Edit Rule` permit you to select rules for editing according to different criteria. Clicking the `Edit Rule` command, for example, will bring up a menu of rule-categories that are known in the current sentence analysis environment:

In this case, there are no rules defined, so there is only a special item named `New` in the menu. If you left-click the `New` item, GWB will fill in a little text in the rule window and then it will move the input-caret into that window:

The caret will be blinking at the point where subsequent type-in will appear, at the beginning of the line of hyphens. The Version and Language are the two parts of the rule's name that were introduced above. You should begin by replacing the `Version Language` place-holder strings in the window by the version and language that you want your new rule to belong to. Since the caret is positioned just after those place-holder strings, you can simply type `CTRL-W` twice to backspace over those words and type in the version/language that you want (say `TOY ENGLISH`). You are then ready to enter the rule itself, above the line of hyphens. As you will see, the hyphens serve to terminate a sequence of rules belong to one version/language.

The format for a single simple rule is the following:

```
Category -->  Category1: Schemata1;
              Category2: Schemata2;
                 Etc.
```

The notation for the rule is closely related to the conventional appearance of LFG rules in published papers, but it includes a little more punctuation so that GWB can clearly understand the different aspects of the rule. (GWB's notation is also more expressive in many ways, as detailed in Chapter III.) Compare the appearance of a Lexical Functional Grammar rule you might write on paper and the way you type it into GWB. You might write a simple S rule like this:

```
S    →        NP           VP
          (↑ SUBJ)=↓      ↑=↓
         (↓ CASE)=NOM
```

In this conventional notation, the association between categories and schemata is marked by their spatial arrangement: schemata are written immediately below their associated categories. This two-dimensional layout is difficult to type and also difficult for GWB to interpret. Thus rules are typed in to GWB linearly, using punctuation marks to indicate how categories and schemata are grouped:

```
S --> NP:(↑ SUBJ)=↓ (↓ CASE)=NOM;   VP:↑=↓.
```

Keyboard notes: The down-arrow can be typed in using the ! key. It may appear on the screen as you type it as ↓, if a "virtual keyboard" file is available for your workstation (see Section 7). Similarly, on keyboards without an explicit up-arrow key the up-arrow can be typed in with a caret `^`. The assignment of keys while typing to an LFG editing window can be examined by clicking `Keyboard` in the right-button command menu and displaying the LFG virtual keyboard. On some keyboards the arrows on the number-pad can also be used to enter up-arrow and down-arrow. Similarly, the rewriting arrow in rules can be typed in either as --> or by using the right-arrow key. Section 7 provides a full description of the various ways in which special characters may be typed in.

In GWB, schemata come after a colon following a category. The punctuation is crucial. The system understands that `(↑ SUBJ)=↓` is a schema attached to the `NP` only because of the colon which precedes it. The semicolon acts as a "close colon"; it informs the system that you have finished with the schemata for one category and the next symbol will be a new category. The final period in the rule says, "This is the end of the `S` rule. If anything follows, it must be the beginning of a new rule." Any string of alphanumeric characters can serve as a category name, and a few punctuation marks such as ' (single-quote) are also allowed. A category name may not contain punctuation marks that are also used as operators in the regular-predicate notation that more complex rules are written in (see Chapter III).

Apart from these system-specific items, the usual paper-and-pencil punctuation for rule-writing is used. The right side of a rule can be a regular expression of category/equation pairs, with parentheses indicating optional elements, Kleene star (*) to say that zero or more repetitions of a regular expression are allowed, vertical bar (|) to separate disjuncts within curly brackets ({,}) surrounding the whole disjunction. Symbols and categories must be separated from each other by white-space, which can be indicated by any number of spaces, carriage-returns, or tabs.

Now you have clicked `NEW` in the `Edit Rule` menu to indicate you are about to write a rule for a new category, and you have named the version/language that the rule will belong to. The input caret is blinking in the Rule Window just after `TOY ENGLISH` and before the line of four hyphens. Type in a first stab at a rule for Sentence (making sure that at least one space or carriage return separates its category from the word `ENGLISH`):

```
S --> NP VP.
```

The daughter categories, `NP` and `VP`, need some schemata. To indicate that the `NP` is the subject and is in the nominative case, point just after the `NP`, click Left to move the caret there, and type:

```
:(↑ SUBJ)=↓    (↓ CASE)=NOM;
```

Don't forget the colon and semicolon. The `VP` is the head of the `S`, so it should be followed by

```
:↑=↓;
```

(This annotation is optional. If there is no equation mentioning ↓ associated with a category, then GWB assumes that ↑=↓). Type this in before the period and the rule for sentence becomes

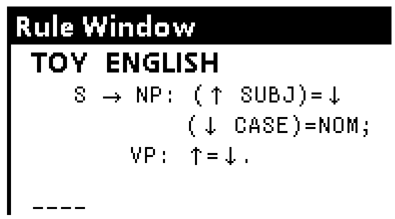
```
S --> NP: (↑ SUBJ)=↓  (↓ CASE)=NOM;  VP: ↑=↓.
```

As mentioned, the semi-colon is used to indicate where the annotations for a particular category end. But it is not needed after the annotations on the `VP` since the closing period is sufficient to show where the annotations end. Nor is it needed after a simple category name unembellished by a colon and schemata, or where there is some other punctuation mark (such as a Kleene star) that is unmistakably not part of a schema.

2.2. Installing rules into the internal database

The next step is to notify GWB that you have finished formulating the rule, and that the rule should be "installed", that is, added to GWB's internal store of rules. If your rule's name had been specified as part of the current analysis configuration (Section 6), it would also be activated at this time for use in analyzing sentences.

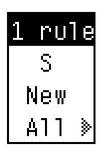
To install a rule, type `CTRL-X` in the rule window. This instructs the system first to analyze the rule to insure correct notation, and then to install it. The window is cleared and the rule is reprinted as the system understood it, in its canonical form, a process known as "pretty-printing". After you have typed `CTRL-X`, the window looks like this:



Another control character, `CTRL-C` (for Cancel), lets you tell the system to forget about what you've typed, but not installed, in a particular window. It aborts the edit, moving the caret back to the Sentence Input window and making the edit window available for re-use. Try this out: Type the erroneous `S-->NP: (↑ OBJ)=↓`. In the same window, type `CTRL-C`. The text is still visible, but the system will erase it the next time you Left-click a category, if this window is used for the new edit.

This prettyprinting feature puts the rule in a standard, nicely tabulated format. You don't have to worry about alignment as you type the rule. Moreover, as you get used to the system's canonical alignment of rule elements, it helps you to tell whether you typed in what you intended to, for example, that you didn't leave a category out of a parenthetic grouping that you really intended to be optional.

When `CTRL-X` is typed, GWB verifies that the form of the rule makes sense, and adds it to the rules of `TOY ENGLISH`. If you now invoke the `Edit Rule` command from the menu in the rule window's title bar, you will see your new category in the menu of rules that can be edited, plus a new menu item labeled `All` that allows you to look at all of the rules at once:



The arrow to the right of the `All` item is the standard indication that a submenu of further selections exists. Selections from that submenu can be made by clicking the mouse on `All` and then sliding off to the right, in the direction of the arrow. The submenu you see when you do this has two items, `Abbreviations` and `Rules`, that can be used to restrict what is brought up in the window for editing. If you select `Rules`, you will see all the node-defining rules (at this point, just `S`). The `Abbreviations` item brings up the definitions only for categorial abbreviations that you might have specified (at this point, none); these are the meta-categories and regular macros described in Chapter III.

2.3. Editing Rules: Menus, visiting, and showing

Once a rule is installed there are a number of ways to display it for inspection and to make it available for editing:

Pop-up Menu : In addition to clicking the `Edit Rule` menu item in the rule window, you can select that item in the menu that appears when you click left or middle in the LFG Icon Window (in the upper right corner of the screen) or when you click left in the Sentence Input

Window (upper left corner). This will produce a menu of the rules that attention is currently focused on (those that are active for sentence analysis, if any, or those in the version you are working on if other versions do not yet exist). Click with left or middle while the cursor is over the category of the rule that you wish to edit to get the definition for that rule displayed in the rule window. You can also display and edit rules that are not currently in focus. If you slide the mouse to the right (following the little triangle) after clicking on the `Edit Rule` menu, another menu containing the version/language identifiers for all currently existing rules will pop up (now just `TOY ENGLISH`). Clicking on one of these will display a menu of the categories that exist in the chosen version/language and you can then choose the one you want.

The `Edit Rule` menu item allows you to edit rules if you have their categories in mind. The other items in the Rule Window menu, `Find Attribute`, `Find Symbol`, `Find Template`, and `Find Category` give access to collections of rules that have certain specified properties. If you select `Find Attribute` instead of `Edit Rule`, a menu of all the symbols that appear as an attribute (the name of a function or feature) in the schemata of some rule (among either the rules currently in focus or, as with `Edit Rule`, in a particular language/version that is first selected by sliding off the `Find Attribute` item). Having just installed the `TOY ENGLISH S` rule, the attribute menu that pops up is

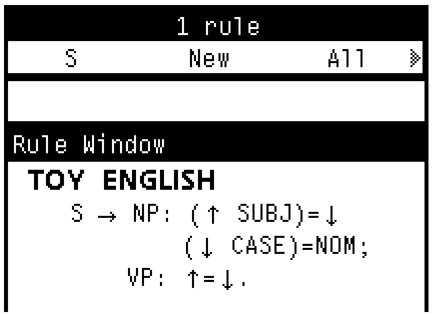
Rule attributes
CASE
SUBJ

When you select a particular attribute (for example, `SUBJ`) all the rules that mention that attribute will be displayed for editing in the window. Similarly, if you choose `Find Symbol`, a you will see a menu of symbols that are used as feature values, and the `Find Category` menu shows a menu of categories. When you choose an item from these menus, all the rules that include the chosen item in their right-hand sides will be displayed. The `Find Template` item gives access to all the rules that invoke a particular template. Templates are notational devices for abbreviating identical or similar schemata found on different rules or lexical items. They are introduced in Section I.5 and considered more fully in Section III.5. As described in Chapter III, they can give the effect of lexical rules of the sort that Kaplan and Bresnan (1982) discuss. The `Edit Template` menu item is provided as a convenient way of bringing up a window in which to define new templates or edit old ones. Finally, the top `TEdit Menu` item brings up the normal TEdit text editing menu for situations when its text formatting or file-manipulation commands may be of use (see the Appendix).

Visiting: Instead of using the menus, you can type in the name of the rules you want to edit, using the `CTRL-V` (for "visit") command. If you type `CTRL-V` while the caret is blinking in a rule window, you will be prompted for the name of a rule you want to see and edit. If you then type the name of a rule followed by a carriage return, GWB will print the rule in the current window. You may do this in a clear window, or one which already has rules in it. Uninstalled changes will vanish. `CTRL-V`, like sliding off the `Edit Rule` menu, also allows you to visit rules that are not currently in focus. (If you want a rule that isn't in focus, you must use the full name of the rule, e.g. `TEST ENGLISH S`. You only have access to rules that currently reside in GWB's internal database, either because they were edited and installed by `CTRL-X` or loaded from a file they were previous saved on (see Section 6)). Visiting a rule means being able to see and edit it, not necessarily to make it active for parsing. A rule is active only if it is mentioned in the current sentence analysis configuration (Chapter II).

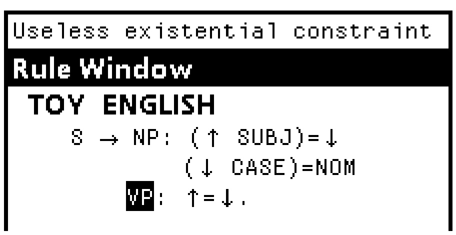
Showing: If you want to view or edit a currently active rule and its name is visible as part of another rule, there is a quicker way of getting access to it. Click with the mouse to make the type-in caret blink on the name of the item, and then type CTRL-S (for “show”). The current definition of the item will appear in a clear window if one is available, otherwise you will be prompted to give the region of a new window. CTRL-S will also cause lexical entries and templates to be shown.

Attached Menu: To get a persistent menu of the rules currently in focus, select the Edit Rule command from the command menu of a rule window, but without releasing the mouse. Then hold down the SHIFT key on the keyboard and release the mouse while the SHIFT is down. A menu will be attached to the top of the rule window with a list of the rules in focus:



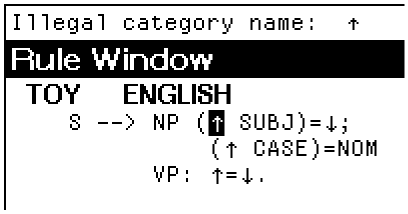
Clicking a category name with either the left or middle button lets you edit that rule. To remove the menu, click in the menu’s title-bar (the black part labeled “Rules”) with the right button to see the window-command menu, and move the cursor until it is over the Close command. Release the mouse button.

Format practice: Let’s play around a bit with the rule format. First, left-click the S item in the rule menu to bring the S rule back into the rule window (from now on we will assume that you have a permanent menu attached to the rule window.). Now remove the semicolon after the NOM symbol and type CTRL-X. This is the result:



Without the semicolon, the system assumes that what follows NOM is part of a functional schema, not a new category. So it treats each successive item as a designator in the f-description language. The VP can only be construed as an existential constraint on a symbol, and GWB, knowing that this does not make sense (because symbols *always* exist), notifies you that you have made an error.

Now, correct the rule by re-inserting the semicolon and then delete the colon after NP. This time GWB prints:



Without the colon in this case, GWB does not interpret what follows `NP` as a schema. It figures that ↑ is a category name, and it complains because ↑ cannot be used in that way. You can revert to the original correct form by re-insert the colon and type `CTRL-X`, but since GWB does not modify its internal database when it detects a notational error, you can also retrieve the original form by typing `CTRL-C` to inform GWB that you want to cancel the changes you have made and then selecting `S` again from the rule menu.

Our next rule, for `VP`, needs a verb, an optional object noun phrase, and the possibility of any number of prepositional phrases, including none at all. We express this by means of the Kleene star, *. Click `New` and type in

```
VP --> V (NP) PP*.
```

This gives the desired pattern of categories, but we also need some schemata to define the functional structures associated with these nodes. Since `V` is the head, and `NP` the object, the rule with the appropriate category-schemata pairs should read:

```
VP --> V: ↑=↓;
       (NP: (↑ OBJ)=↓  (↓ CASE)=ACC;)
       PP*: (↑ (↓ PCASE))=↓;.
```

Add the appropriate schemata to each category in the rule and install the rule. Note that you can omit the semi-colon after the schemata for the `NP` and `PP` since the parentheses and final period are sufficient for GWB to infer where the schemata end. Also note that the schemata after the `PP*` applies to each of the zero or more `PP`s that may occur, with ↓ instantiated differently for each `PP`.

Finally, type in and install a rule for simple noun phrases. The determiner and head both contribute all their features to the f-structure of the whole phrase, so they should each have an ↑=↓ schema. Relying on the convention that this schema is inferred by default if ↓ is not mentioned on a category, the rule can be written without any functional annotations at all:

```
NP --> (DET) N.
```

Caution: Be consistent in use of case when you type in categories and symbol designators. DET is not the same as Det to GWB and Case is not the same as CASE. If you have one of these in a rule and the other in a lexical entry, GWB will assume they are different categories and an analysis will fail for what may seem like a mysterious reason.

You can click `New` and enter the `NP` rule in a clear window, or, since editing windows can include several rules at the same time, you can enter the `NP` rule before or after the text of the `VP` rule currently displayed and type `CTRL-X`.

Now you have installed a few grammar rules. GWB has responded by updating the menu of rules, which makes it easy to access the current definition of rules and to try out new variations. Say you want to modify the `NP` rule to make the determiner obligatory. Click `NP` in the rule menu, and the current `NP` rule will be displayed in a rule window, available for editing. Delete the parentheses before and after `DET`. Type `CTRL-X`. Whatever changes you make in the right-hand side of the rule will be installed, and will constitute the rule that will be printed instead of the previous one when NP is selected from the menu.

2.4. Keeping alternative formulations

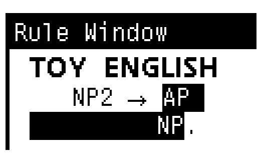
By now, you probably realize that when you use CTRL−X on successive edits of the same rule, the system replaces the previous version and saves only the current edit. The previous versions are simply gone without a trace.

If you want to keep alternative versions for later reference, one strategy is to designate a separate TEdit window (obtainable by clicking the TEdit item in the menu that pops up when you click the right button in the background region of the screen) for archival storage of old rules and make a habit of copying each version of a rule into the archive window before you re-use the edit window, or as soon as you have installed a rule. A rule saved in this way is easy to copy back into a Rule Window if you decide to resurrect it.

Instead of saving previous versions of a rule in a separate TEdit window, you might copy them to a different version of English, say TRINKET, before you edit the TOY version. To copy the NP rule to the TRINKET version, bring it up into an editing window, change TOY to TRINKET and type CTRL−X. You can then modify the TOY version but be able to retrieve its original formulation by visiting (CTRL−V) TRINKET ENGLISH NP or choosing it from the slide-off Edit Rule menu. You can restore TOY ENGLISH to its original form by changing TRINKET back to TOY and typing CTRL−X.

2.5. Deleting a rule

Suppose you install a rule (for example: NP2-->AP NP.) and later decide you don't want it after all. You can delete or erase a rule by bringing it up for editing, deleting its entire right side, and installing the empty rule with CTRL−X. For this example, having installed the undesired rule, you should click NP2 in the category menu with the left mouse button. In the Rule Window, delete what follows the arrow but leave the period which marks the end of the rule. Select the text to be deleted by clicking the left button before AP and the right button after the NP. The text to be deleted is highlighted in black:



Type the key marked DEL or DELETE to remove the highlighted text and then type CTRL−X. Voilà, the category menu is free of NP2. In other words, if you delete the entire right hand side of the rule, up to but not including the period, the rule itself is deleted from GWB's internal database and also from the active analysis environment. Leaving the period in is crucial when there is more than one rule in a window. This method for deletion applies to lexical entries as well: if the body of a lexical entry is completely empty, the word is taken out of the lexicon.

2.6. Which window?

If you summon a rule with a rule menu, which window will it be displayed in?

> If you click with the middle button on a category or attribute, the system will prompt you for a region on the screen to set up a brand new rule window, leaving existing editing processes untouched.

If you click with the left button, the system will try to re-use an existing window. Any Rule Window with text in it may be cleared and re-used as long as it does not contain an un-installed change. There is no risk of losing material in this way because whatever has been installed can be retrieved by clicking in the category menu. If the text in the window has been edited but you haven't installed the changes with `CTRL-X`, a left-click will get you a prompt for a new window, just like a middle click. A window will also be re-used if you have aborted the edit in it with `CTRL-C`. Only if all windows have uninstalled, uncanceled changes will the system prompt for a new one.

Try this out:

Left-click `NP`

Left-click `S`

Middle-click `VP`

Make a change in the `S` rule. Do not install the change.

Left-click `NP`

Install the change in the `S` rule

Left-click `VP`

Make a change in `VP`. Do not install the change.

Left-click `NP`

`CTRL-C` in `VP` window

Left-click `S`

If you try to `CLOSE` a window with uninstalled changes in it, GWB will ask you to confirm that you really want to flush the changes before it closes the window and forgets your edits. The cursor changes to a picture of a mouse with the left button dark. Clicking the left mouse button confirms that you wish to close. To prevent the window from closing, click the right or middle button. These conventions also apply to the `QUIT` command in standard Medley TEdit menu.

3. Lexical entries

Lexical entries are similar to rules in terms of punctuation, installation, visiting, and editing. There are also corresponding menus: each lexicon window has a command menu that includes `Find Attribute`, `Find Template`, `Find Category`, and `Edit Lex` items. Invoking the `Edit Lex` command will produce a menu of the lexical items that are currently in focus (as with rules, those that are active for sentence analysis, if any, or those in the version you are working on if other versions do not yet exist). As with rule editing, this menu will be permanently attached to a lexicon window if the `SHIFT` key is down when the `Edit Lex` command is selected.

3.1. Typing in

Before you can use your rules to analyze a sentence, you have to define a few words—a toy lexicon. Go to the lexicon menu and left-click the `New` item. The caret will start blinking in the lexicon window, which will contain a version/language line and another line with the word `New` followed by a period.

Here is the implicit template for the lexicon window:

```
VERSION LANGUAGE
     word   Category1 Morphcode1 Schemata1;
            Category2 Morphcode2 Schemata2;
                 Etc.
```

Morphcodes encode information about the morphological paradigm that the word belongs to, including inflectional information about irregular forms, and are described in Section 4. The schemata in lexical entries are identical in form and interpretation to the schemata that appear in c-structure rules, although they are not preceded by a colon.

Begin the `TOY ENGLISH` lexicon by entering the definition for `walk`. That is, enter `TOY ENGLISH` as the version/language. Then replace the word `New` with the word `walk` and type in the category `V`, the morphcode `*` (which you will later revise), and the schema

```
(↑ PRED)='WALK<(↑ SUBJ)>'
```

The quote marks enclose the semantic form, as in pencil-and-paper entries. This form indicates that the predicate `WALK` takes one grammatical argument, a subject. The predicate in a semantic form must be a designator in the f-description language, usually a symbol without internal spaces. So if you wish to wax more descriptive, you can separate the words in the predicate with hyphens (for example, you might use `KEEP-TABS` for the idiomatic sense of `keep`). Like rules, lexical entries end with a period. Install the entry using `CTRL-X`. The window will be cleared and the "pretty" form of the entry will be printed:

```
Lexicon Window
TOY ENGLISH

  walk    V * (↑ PRED)='WALK<(↑ SUBJ)>'.
----
```

Again as with rules, the dashes are supplied by the system to mark the end of a set (in this case only one) of lexical entries in a single version/language.

Notice that the word `walk` is entered in lower-case, as it would appear in ordinary text. When analyzing words in an input sentence, GWB first attempts to match lexicon headings against the characters just as they appear in the input. If that fails it tries converting an initial capital (for example, at the beginning of the sentence) to lower-case or an all upper-case string to lower-case with or without an initial capital. Because of this procedure, the entry above will be retrieved for the input words `Walk` and `WALK` as well as `walk`. If a word is acceptable only in all upper-case (e.g. `IBM`) or only with an initial capital (e.g. the English proper noun `Sam`), then it should be entered in the lexicon with the necessary upper-case letters. GWB takes the sequence of characters up to the next white-space as the spelling of the word. If for some reason you want to include a space in the spelling of a word, it must be "escaped" by preceding it with the backquote character (`` ` ``). If you want backquote itself to be literally in the name of a word, then it must also be escaped in a double-backquote sequence.

Of course `walk` can also be used as a noun. To express this alternative, insert the following before the final period:

```
;  N * (↑ PRED) = 'A-WALK'          CTRL-X
```

Here too, the semi-colon ends a category specification. In rules, a sequence of categories means that they all must appear in the c-structure in that order. In lexical entries, a sequence of categories may be thought of as a disjunction since each indicates a possible interpretation of a homophonous word. The parser will try out each interpretation independently. Note that this is the only way that you can express a disjunction of lexical categories in GWB; the following specification, with the disjunction brackets around the categories and morphcodes, does *not* work:

```
walk    { V * (↑ PRED)='WALK<(↑ SUBJ)>'
        | N * (↑ PRED) = 'A-WALK'}.
```

When you type `CTRL-X` to install a lexical entry, GWB checks the format and punctuation of your definition to see if it makes sense. If you have made, for example, a punctuation error that results in an ill-formed or nonsensical definition, GWB will inform you of that fact just as it does for errors in the format of rules. To see what happens, try deleting `TOY` and typing `CTRL-X`. This causes GWB to take `English` as a version name and `walk` as a language and attempt to make sense of the rest, resulting in an error:

```
Useless existential constraint
Lexicon Window
  ENGLISH

   walk    V * (↑ PRED)='WALK<(↑ SUBJ)>';
           N * (↑ PRED)='A-WALK'.
 ----
```

GWB attempted to fit the symbols it found in the lexical entry for `walk` into the fields of the template above, and failed. Now restore the lexical entry: either add `TOY` back as the version, select `walk` from the `Edit Lex` menu, or type `CTRL-V` and its name `(TOY ENGLISH walk)` to visit it.

3.2. Starting with an existing entry

It is easier to write lexical entries (and rules as well) if you use an existing entry as a starting point and "edit an old one into a new one" by replacing selected elements of the starting entry with something new of the same type. (Note that the old entry will not be changed by this process since you will always change the heading-word.) Bring up the entry for `walk` and edit it into one for `girl` by erasing the text from the `V` category to the semicolon, changing the word `walk` to `girl` and also changing the predicate `A-WALK` to `GIRL`. Then type `CTRL-X`. The result will look like this:

```
Lexicon Window
 TOY  ENGLISH

  girl    N * (↑ PRED)='GIRL'.
 ----
```

Now let's consider a transitive verb. Edit the verb definition of `walk` into `kick` by replacing

`walk`	with	`kick`
`'WALK`	with	`'KICK`
`(↑ SUBJ)`	with	`(↑ SUBJ)(↑ OBJ)`

and deleting the `N` part of the entry. Here's the result:

```
kick  V * (↑ PRED)='KICK<(↑ SUBJ)(↑ OBJ)>'.
```

This is a verb which can also be used without an object, as in “Horses sometimes kick”. It might seem natural to express this fact by surrounding (↑ OBJ) with brackets or parentheses to indicate optionality, but GWB does not accept this notation. Instead, you must disjoin two largely similar schemata as follows:

```
{(↑ PRED)='KICK<(↑ SUBJ)(↑ OBJ)>'
|(↑ PRED)='KICK<(↑ SUBJ)>'}.
```

Note also that disjunction only applies at the level of equations; you can’t say

```
(↑ PRED) = {'KICK<(↑ SUBJ)(↑ OBJ)>'
           |'KICK<(↑ SUBJ)>'}.
```

Often the definition of a new word will be exactly like an existing word, except for a systematic change to the semantic relation of the predicate. For example, the word `halt` belongs to the same inflectional paradigm as `kick` and is also optionally transitive. We could construct its definition by bringing the definition for `kick` into an edit window, changing `kick` to `halt` and replacing the two occurrences of `KICK` with `HALT`. GWB provides a short cut, the “like” feature, that automates some of these edits. Instead of bringing an existing word into the window to start editing, you can simply assert that a new word is like an already existing one:

```
halt like kick.
stroll like walk.
```

These are not proper definitions, but when you type `CTRL-X`, GWB will construct proper definitions by systematically modifying existing definitions of the indicated words. In particular, any string that matches the existing word in its definition will be replaced by the new word. The match is done in a case-insensitive way, so that `kick` matches `kick`, `Kick`, or `KICK`, and the corresponding replacements will preserve the case-pattern of the match, giving `halt`, `Halt`, or `HALT`. Thus, after typing `CTRL-X`, the newly constructed definitions will appear in the window:

```
halt    V * { (↑ PRED)='HALT<(↑ SUBJ)(↑ OBJ)>'
             |(↑ PRED)='HALT<(↑ SUBJ)>'}.

stroll  V * (↑ PRED)='STROLL<(↑ SUBJ)>'
        N * (↑ PRED) = 'A-WALK'.
```

These definitions are very nearly correct; all that is needed is to change `A-WALK` to `A-STROLL`. This was not done automatically because the matching routine operates on whole symbols (`A-WALK`), not on substrings they might contain.

Our inventory of words will need a determiner, so enter a definition for `the`:

```
the   DET * (↑ DEF)=+.
```

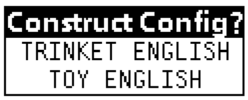
`the` is not a predicate in our toy lexicon, just an entity that has a positive value for the feature of definiteness, however that might be interpreted.

3.3. Analyzing sentences

You have perhaps entered enough rules and lexical entries to analyze (or parse) `The girl walks`. At the top of the screen is a window titled `Sentence Input Window` where you can use Tedit commands to enter and edit sentences to be analyzed. Click in this window and type

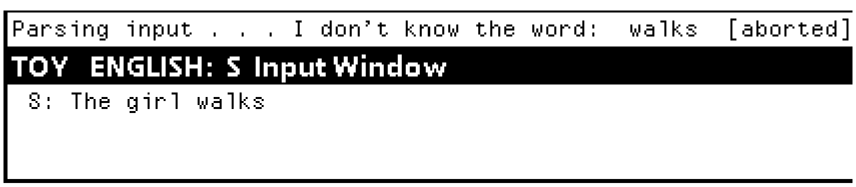
`The girl walks` [without a period, since your grammar doesn’t allow for punctuation]

Now type CTRL–X, which in this window has the effect of invoking the sentence analyzer or parser. GWB cannot operate at this point, however, because you have not yet told it which rules in its internal database you want to be active during the analysis of this sentence. Remember you have entered both TOY and TRINKET versions of ENGLISH, and it is not obvious which rules you want to parse with. You indicate which rules, lexical entries, and other linguistic specifications you intend to be active for a particular analysis by installing a "configuration". Configurations, as described in Chapter III, provide very detailed control over the information in the internal database that will be active in the current sentence analysis environment. GWB can help you get started by constructing a configuration that includes rules and lexical entries from a single version/language and default values for a number of other parameters (the root category of the grammar, the governable functions for coherence checking, and so on). Thus, when you type CTRL–X, the following menu appears on the screen:



This menu lists the version/languages of the rules you have previously entered. If you click the TOY ENGLISH item, a configuration will be created and installed for the rules and lexical entries in that version, with standard default values for the other configuration parameters. The TRINKET ENGLISH rules will remain in the internal database, perhaps to be activated when you create and install another configuration.

When the new configuration is installed, the title of the sentence input window changes to give the active configuration's name (configurations also are identified by version/language pairs) and the root category specified in that configuration (which has S as a default value). Then GWB proceeds with the analysis of your sentence. It inserts S: in front of the string to record what category it has taken to be the root for this analysis. It then attempts to analyze the words of the string and apply the grammar to them. In this case it cannot find a lexical entry for walks. It reports this in the prompt region above the Sentence Input Window, which now looks like:

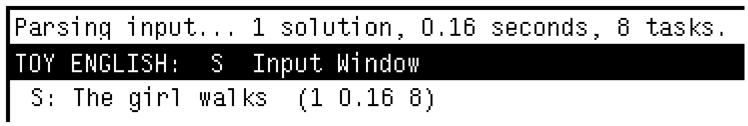


The caret resumes flashing at the end of the string. GWB doesn't know that walks is the plural of walk because we haven't specified any morphology for the current analysis environment.

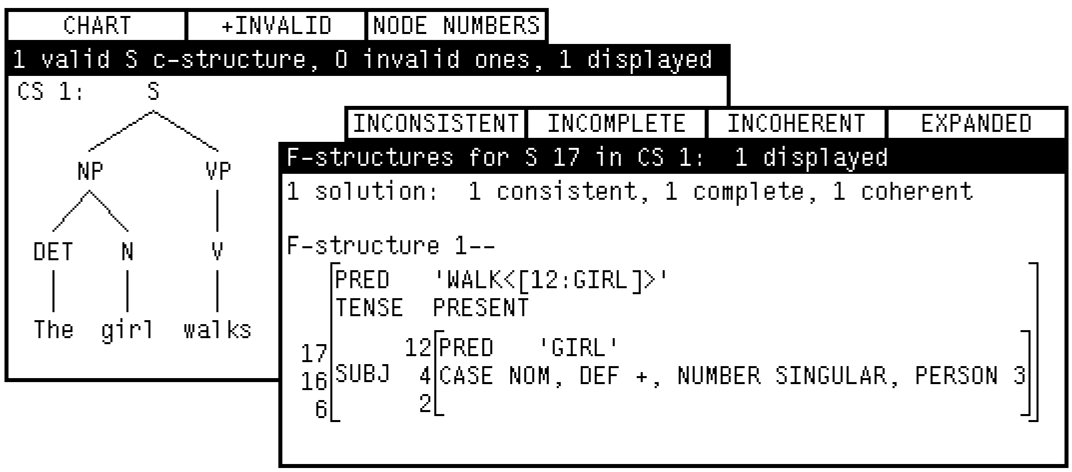
The next section describes a rudimentary mechanism for expressing systematic morphological variations, at least for simple suffixing languages like English. But for the moment, let's enter walks as a separate word. First use Edit Lex in the Lexicon Window menu to get to the definition for walk. Now append an s on walk and, at the same time, you might as well give the schemata for person, number, and tense:

```
walks    V * (↑ PRED)='WALK<(↑ SUBJ)>'
             (↑ SUBJ NUMBER)=SINGULAR
             (↑ SUBJ PERSON)=3
             (↑ TENSE)=PRESENT.
```

Type CTRL–X to install this entry and then try the parse again. To reparse a sentence that already exists in the sentence input window, make sure that the caret is blinking anywhere on the line with the sentence (perhaps by moving the mouse and clicking left) and type `CTRL-X`. When you try it with the newly installed definition for `walks`, the analysis succeeds. The number of solutions and the time it took to perform the analysis, are displayed in the prompt window and are also recorded for future reference in parentheses at the end of the sentence:



The system also records the number of tasks, a measure of the number of steps GWB required to perform all the constituent structure computations. The c-structure and f-structure for the sentence also appear in their windows:



The terminal nodes in the c-structure act like the items in the lexicon menu. Click `The` in the tree. Its lexical entry appears in a lexicon window. Which window it appears in is determined by whether you click with the left or middle mouse button, as with Rule Windows. You can also get easy access to the rules whose categories appear in the tree by clicking on the nonterminal nodes while holding down the `CTRL` key.

It is worthwhile examining the configuration that GWB constructed for you to get a preview of the information that these specifications contain. Press the left button in the title-bar of the sentence input window and release it over the `Edit Config` item in the menu that appears. This will cause a configuration-editing window to open up with the just-created `TOY ENGLISH` configuration displayed in it:

```
Configuration Window
TOY ENGLISH
 RULES (TOY ENGLISH).
 ROOTCAT S.
 LEXENTRIES (TOY ENGLISH).
 TEMPLATES (TOY ENGLISH).
 MORPHTABLES (TOY ENGLISH).
 GOVERNABLERELATIONS  SUBJ OBJ OBJ2 OBL-?* POSS COMP ?COMP.
 SEMANTICFUNCTIONS  ADJ XADJ.
 NONDISTRIBUTIVE  .
 PROJECTIONS (φ F-).
 EPSILON e._
 PARAMETERS  .
 INPUTKEYBOARD DEFAULT.
 LEXICALFONT DEFAULTFONT.
 TESTFILE  .
----
```

You can see that the rules and lexical entries will be drawn from `TOY ENGLISH`, that `S` is the root category of the grammar, and that `SUBJ`, `OBJ`, etc. are the governable grammatical relations. The details of these and other properties of a configuration are described in Chapter III.

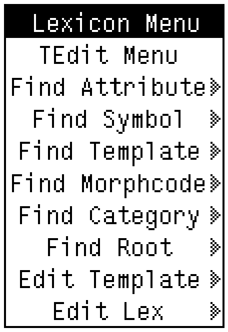
3.4. Accessing lexical entries

Lexical entries are examined, edited, and deleted according to essentially the same protocol that applies to rules. Like rules, any lexical entries in GWB's database can be visited by using `CTRL-V`. Go to an unused Lexicon Window and type `CTRL-V`. GWB will prompt you for entries to visit. Type `TOY ENGLISH girl kick the`, followed by a carriage return. Out comes a nicely formatted list of entries. But if you type in `TOY ENGLISH mouse`, which hasn't been defined in the lexicon, the system prompts you for a new entry by displaying

```
mouse .
-----
```

This is the way an undefined word appears, and you can enter its definition by simply adding the desired information between the word and the final period. On the other hand, just as with rules, you can delete an existing entry by leaving only white-space between the word and the terminating period. Note that you must be careful to leave at least one space before the period: a period before white space, as in `Mr.`, is interpreted as part of the word, perhaps indicating an abbreviation.

As you have seen, lexical entries are also accessible from the Lexicon Window menu, which pops up when you click the left button in the title-bar of a lexicon-editing window. In addition to `Edit Lex`, this menu has `Find Attribute`, `Find Symbol`, `Find Template`, `Find Morphcode`, `Find Category`, `Find Root`, and `Edit Template` items, each with a slide-off to give access to inactive version/languages:



The `Edit Lex` item brings up a menu of all lexical entries either currently active or defined in the selected version/language. If there are too many of them to fit easily in a single menu, the first word in an alphabetical group will appear with a slide-off triangle, and the other words can be reached by sliding the mouse to the right to bring up a secondary menu. This is illustrated by the following menus for a larger lexicon:

The word `kick` does not appear in the top-level menu for this lexicon, but it is ordered alphabetically between `keep` and `like`. Thus it can be found in the submenu obained by sliding to the right of `keep`. If there had been a larger number of words beginning with `k`, some of the items in this secondary menu would themselves have slide-off arrows to provide for further choices. In this way all the words in even a fairly large lexicon can be accessed through a relatively small number of menu selections.

The `Find Attribute` lexical menu item is very similar to that item in the rule menu. When it is selected a menu of all the attributes that appear in any lexical entry pops up, and clicking on one of them causes all the lexical entries that utilize that attribute to be displayed. `Find Symbol` selects the lexical items that use a particular symbol as a feature value, and `Find Template` selects items that mention a particular template. `Find Category` gives access to the lexical entries within a selected lexical category (`Prep`, `N`, etc.). `Find Morphcodes` gives access to the lexical entries that mention a particular morphology code as described in Section 4. `Find Root` shows you all the words that serve as a root or affix in the morphological analysis of other irregular forms (also described in Section 4); selecting a root or affix brings into the editing window all the entries that refer to it. Finally, clicking on the `Edit Template` item will open a window for editing or defining a template specification; a template window is not among the windows initially on the screen.

4. Morphological information

So far your grammar of English contains no indication that for each regular noun or verb defined in the lexicon there are several systematically related words. GWB provides a simple facility called a morphology table ("Morphtable") to relate these words and make it unnecessary to enter them each separately in the lexicon. Morphtables are limited in their power and are useful mainly for simple suffixing languages like English, but they may offer a little help even for morphologically more complex languages. Now you can type in a morphtable for `TOY ENGLISH` and see how it works.

4.1. Interpreting the morphology table

Go to the LFG Icon Window or the Sentence Input Window, and select the `Edit Morph` menu item. This will produce an empty version/language menu of morphology tables that can be edited. Select `New`. GWB will respond by prompting for a region for a morphology window. Since there is so far no morphtable for `TOY ENGLISH`, the window opens with only the language and version in it. A morphology table contains a list of rules of the form

```
(Takeoff Addon (Cat1 Morphcode Affix NewCategory)
               (Cat2 Morphcode Affix NewCategory)
                 ... )
```

(A few examples of morphology rules can be obtained by selecting the `Morphology tables` item in the menu that pops up when you slide off the `Examples` item in the Sentence Input Window menu.)

The `Takeoff` slot contains a string of characters (possibly the empty string "") and so does the `Addon`. Following the `Addon` is one or more category specifications consisting of these elements: `Cat1` stands for a category name. `Morphcode` specifies what inflectional paradigm a root must belong to for this rule to apply. `Affix` stands for the name of a special lexical entry containing the resulting features associated with the inflection that this rule identifies; these entries are described below. `NewCategory` allows for a category change in derivational morphology (e.g., adding -ly changes an adjective to an adverb).

In the lexical entries above you typed in * where a morphcode was called for. Go back to the entry for `girl` and replace the asterisk with the morphcode `S`, to indicate that the plural form takes -s. Then in the morphology window type in the following morphtable entry, according to the template above.

```
(s "" (N S -NPL))
```

The `NewCategory` indicator, which is optional, has been omitted. This specification is an instruction to GWB which says:

If `Takeoff` appears at the end of a word	`girls`
take the `Takeoff` off	`girl`
and put the `Addon` on to get a potential root	`girl` + "" = `girl`
Look up the root in the lexicon.	`girl N S...`
If it has a definition for `Cat1` (`N`)	yes

and that definition has the morphcode S yes

Then `girls` can be analyzed as
the morphological combination `girl + -NPL`

In the same spirit you can introduce rules for the other common spelling variations of English noun inflections. The rule

```
(es "" (N ES -NPL))
```

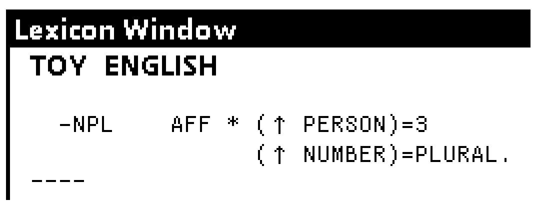
provides an appropriate analysis for words like `churches` and the rule

```
(ies y (N ES -NPL))
```

handles the morphographemic variation in words like `entries`. Note that the characters in the `Takeoff` and `Addon` are all in lower-case, as they normally appear in text and in the headings of lexical entries. As with rules and lexical entries, GWB reads and internalizes the information in the Morphology Window only when you type the `CTRL-X` command in that window. A previously installed morphtable can be retrieved for inspection or further editing by choosing its version/language from the menu that pops up after selecting the `Edit Morph` item from the menus associated with the LFG Window, the Sentence Input Window, or an existing Morphology Window.

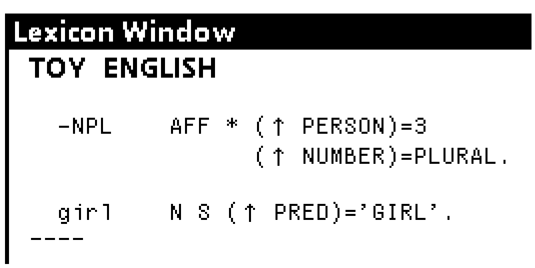
4.2. Affixes in the lexicon

What is `-NPL`? It is an item to be defined in the lexicon, like `girl`. Its category is `AFF` to indicate that it is an affix. Type into a lexicon window and install the entry for this affix:



The person/number schemata in this lexical entry will be added to the schemata for the `N` sense of `girl` to make up the lexical entry for `girls`. Thus the plural features do not have to be specified redundantly throughout the lexicon. The `AFF` morphcode * says that this item does not itself undergo any morphological process, i.e., it has no affixes. It is conventional to precede the names of suffixes with a dash, to distinguish them from ordinary words.

Now if you type `The girls walk` in the sentence input window, the system will be able to correctly analyze the sentence. Left-click on `girls` in the c-structure window to inspect its lexical definition. GWB shows its morphological decomposition in the lexicon window by printing**:**



You can also examine the morphological analysis of particular words without having to create and analyze a sentence that contains them. If you click in a lexicon window and type `CTRL-V`, you will be prompted to enter a word to visit. If you type in a single word (for example, `girls`), you will see the same morphological decomposition that would appear if you clicked on that word at the bottom of a c-structure tree.

4.3. More morphology rules

We can continue filling out the morphtable, for example, by adding a rule for the `-ed` past-tense suffix of regular verbs like `walk`:

```
(ed "" (V S-ED -VPAST))
```

Changing the `V` morphcode for `walk` from * to `S-ED` should enable the system to analyze `The girls walked` However, when you install the modified morphtable and lexical entry by typing `CTRL-X`'s and then type this sentence, the system responds with the message

```
I don't know the word:   -VPAST  [aborted]
```

If a word is decomposed by the morphtable into a pair of morphemes, the system requires that the affix as well as the root be defined in the current lexicon. It does not assume from the absence of an affix definition that you intended the affix to supply no inflectional features. In this case, the following disjunctive definition for `-VPAST` is appropriate:

```
-VPAST  AFF * {(↑ TENSE)=PAST | (↑ PARTICIPLE)=PAST}.
```

Again ask for a parse of the same sentence, and this time it works.

You must provide another morphology rule to interpret a final `-s` on a verb as a marker of a third person singular verb. The following rule encodes this information:

```
(s "" (V S-ED -V3SG))
```

However, this can be collapsed with the other specification for the takeoff `S`:

```
 (s "" (N S -NPL)
       (V S-ED -V3SG))
```

Of course, the inflectional schemata for `-V3SG` must be entered as a separate item:

```
-V3SG  AFF  *      (↑ TENSE)=PRESENT
                   (↑ PERSON)=3
                   (↑ NUMBER)=SINGULAR.
```

Now you can change the verbal morphcode for `walk` from * to `S-ED`, and remove `walks` from the lexicon (by deleting all the material between the word and the final period and type `CTRL-X`). When you attempt to analyze `The girl walks`, the appropriate definition for `walks` will be constructed automatically from the definition for `walk`, the `s` morphology rule, and the entry for `-V3SG`.

This mini-grammar also requires morphological rules and affix lexical entries for the other elements of the regular noun and verb paradigms, for example, for the `-ING` verbal suffix. Rules for the zero suffixes for nouns and verbs with empty strings for both the takeoff and addon should also be added:

```
 ("" "" (N S -NSG)
        (V S-ED -VINF))
```

The affix entries for the zero morphemes (e.g. `-NSG`, `-VINF`) permit the inflectional features for singular nouns and infinitive verbs to be factored out of individual lexical entries:

```
-NSG   AFF * (↑ PERSON)=3
             (↑ NUMBER)=SINGULAR.

-VINF   AFF * {  (↑ INF)=+
                |(↑ TENSE)=PRESENT
                 ~[(↑ SUBJ NUMBER)=SINGULAR
                   (↑ SUBJ PERSON)=3]}.
```

According to the -VINF entry, a root verb is either an infinitive or, if the subject is not third-person singular, a present tense form. The tilde is GWB's negation symbol, and square-brackets are used to group a set of conjoined schemata together. Of course, the same zero-morpheme rule applies for the morphcode classes representing all the English inflectional paradigms. The rules for all the different morphcodes can be collapsed into a single rule that includes a parenthesized list of morphcodes instead of just a single one:

```
("" ""  (N (S ES) -NSG)
        (V (S-ED ES-ED) -VINF))
```

GWB's current morphological analyzer has one other capability that makes it easy to describe words where the final character of the root is doubled in forming some of the inflectional variants. The gemination pattern for p in the paradigm stop, stopped, stopping could be handled by explicit rules such as

```
(pped  p (V S-ED -VPAST))
(pping p (V S-ED -VING))
```

but this strategy would, redundantly, require one ed rule for each of the consonants p, d, b and so on that enter into this kind of pattern. Instead, this collection of rules can be collapsed into the following specification:

```
(#ed "" (V S-ED -VPAST))
```

GWB gives a special interpretation to a number sign (#) at the beginning of a takeoff. The # matches any character provided that it is the same as the character in the word that precedes it. Thus, #ed in this rule matches the ped suffix in stopped but not in hoped. When this rule applies to stopped, the ped will be taken off and replaced by the empty string to produce the desired root form.

As a final illustration, we provide a category-changing rule of derivational morphology, the rule that relates adverbs ending in -ly to a corresponding adjective (suddenly to sudden). The NewCategory component of a rule can be used to encode this rudimentary kind of derivation:

```
(ly "" (ADJ * -ADV ADV))
```

This indicates that an ADV results when an ADJ with a * morphcode is obtained by removing ly from a word, and that the schemata in the -ADV lexical entry are to be conjoined with those of the adjectival stem to form the schemata for the derived form.

4.4. Irregular forms

The rules in the morphology table thus permit the redundancy of systematic inflectional suffixes to be removed from individual lexical entries. GWB also allows redundancies to be factored out of irregular inflections, forms whose morphographemic variations do not conform to any of the common patterns. For example, the verb go has irregular past tense and past participle forms (went and gone). The inflectional features of these forms cannot be supplied by general rules, but they do share root features with go and with the more regular third-person singular goes and present participle going. All forms of go have at least the same PRED semantic form and perhaps other features as well.

Schemata can be shared among the various forms of a word by placing them in a separate, distinct lexical entry under a special morphcode ROOT. You can enter the root schemata common to the various forms of go can be entered in the lexicon under the stem go-:

```
go-  V ROOT (↑ PRED)='GO<(↑ SUBJ)>'.
```

This is in some sense a pseudo-lexical entry, much like the entries for the various affix schemata, since the heading of the entry is not something that is expected to appear by itself in a sentence. Spelling the heading with a hyphen at the end is a useful convention for marking this fact and also for indicating that these are features for a root that is to be completed by the addition of features typically associated with some suffix.

The individual forms of go can then refer to these root features by specifying go- in a special morphcode format. The entries for gone, for example, can be entered as

```
gone  V @go- (↑ PARTICIPLE)=PAST.
```

The @ in the morphcode indicates that gone is a verb whose schemata are the result of conjoining the V-ROOT schemata of go- with the idiosyncratic participle schema for this inflectional form. The past-tense form can be defined in a similar manner:

```
went  V @go- (↑ TENSE)=PAST.
```

The infinitive and third-singular forms could also be defined in this way, with their inflectional schemata listed explicitly. But these schemata are not idiosyncratic at all—they are identical to the schemata of the normal inflectional affixes -VINF and -V3SG that were defined above. This commonality can be represented by including the names of the inflectional affixes in the @ morphcode without listing any additional distinctive schemata:

```
go    V @(go- -VINF).

goes  V @(go- -V3SG).
```

The parenthetic morphcode permits the morphological analysis of these words to be given explicitly, indicating that their schemata come from the go- root and the normal inflectional affixes.

Entries of this type can be used to relate all the forms of a given stem to common lexical material defined in a special stem entry, conventionally spelled with a trailing hyphen, and to the schemata associated with independently specified inflectional affixes. Each of the forms requires a separate lexical entry, and clearly this cannot be avoided for a verb like be that has completely idiosyncratic inflectional realizations. But many stems have only a few forms that do not conform to a regular paradigm. In English, for example, the third-person singular and present-participle forms are almost always regular; only the past and past participles show idiosyncratic variation. The morphtable mechanism can be used to express this sort of subregularity and thus avoid the need for explicit specification of forms like goes and going (and also the citation form go, given that it is related to the stem go- in a conventionally systematic way). This is accomplished by providing a more discriminating morphcode in the stem and also morphtable entries to interpret the subregularity represented by that morphcode. Thus, the fact that go and do take the standard -es and -ing suffixes can be indicated by changing the ROOT morphcode in their stem entries to a new code ES-ING:

```
go-  V ES-ING (↑ PRED)='GO<(↑ SUBJ)>'.

do-  V ES-ING (↑ PRED)='DO<(↑ SUBJ)>(↑ XCOMP)'.
```

The interpretation of this morphcode is then provided by additional morphtable entries:

```
(es  - (V ES-ING -V3SG))
(ing - (V ES-ING -VING))
(""  - (V ES-ING -VINF))
```

The first line allows the `es` suffixes in `goes` and `does` to be replaced with hyphens to produce the stem forms `go-` and `do-`. Since these are marked with the morphcode called for in the table entry, this analysis results in the appropriate third-singular schemata. The second entry identifies `going` and `doing` as the present participles of `go-` and `do-`, respectively, and the third line accounts for the base forms `go` and `do`. Rules such as these make it unnecessary to have separate entries for regular forms of irregular verbs. Only the stem and its truly irregular variants must be specified. The truly irregular forms are defined just as before:

```
gone  V @go- (↑ PARTICIPLE)=PAST.

went  V @go- (↑ TENSE)=PAST.
```

When a `ROOT` morphcode is not present in the stem entry to identify the intended root schemata, the system extracts the schemata associated with any single morphcode for the indicated category that does exist in the root entry (`ES-ING` in the `go-` example).

This mechanism for expressing subregularities can be used even when an irregular form has the same shape as a separate regular form, as is the case for the past and past-participle forms of `hit`. The infinitive can be derived from the stem `hit-` by virtue of a morphtable entry for the subregular gemination morphcode `S-#ING`:

```
(#ing  - (V S-#ING -VING))
```

But `hit` would also have its own lexical entry, providing explicitly only for the past/past-participle inflection:

```
hit-  V S-#ING (↑ PRED)='HIT<(↑ SUBJ)(↑ OBJ)>'.

hit   V @(hit- -VPAST).
```

The result of the morphtable analysis will be taken as an alternative to the explicit entry to provide an ambiguous representation for `hit`.

The several ways of specifying the morphological decomposition of a single word can be combined to characterize complex homophonous forms. The word `saw`, for example, can be a verb in two senses, either as the irregular past of `see` or the regular root of `saw`. It can also be a common noun. All these possibilities are expressed in the following lexical entries:

```
saw   V      {@see- (↑ TENSE)=PAST;
              S-ED (↑ PRED)='SAW-WOOD<(↑ SUBJ)(↑ OBJ)>'}
      N      S (↑ PRED)='SAW'.

see-  V      ROOT (↑ PRED)='SEE<(↑ SUBJ)(↑ OBJ)>'.
```

These entries illustrate how curly-brackets with a semicolon separator can be used to represent alternative morphcode-schemata combinations.

As you can see, the combination of `ROOT` and `@` references can define complicated dependencies between the definitions of various items. The `Find Root` item in the Lexicon window menu helps you keep track of those dependencies.

5. Templates

Putting schemata in the lexical entry for an affix factors those specifications out from all the word-forms that take that affix, and thus enables morphologically-related generalizations to be stated across broad classes of lexical items. GWB provides a further means for expressing functional generalizations that are not obviously correlated with inflectional morphology. You can incorporate a collection of schemata into the definition of a named “template”, and then use the name of the template in place of those schemata in lexical entries or in c-structure rules. The specifications of individual lexical entries or rules can thus be streamlined, with the template definition representing the common properties that are shared by all the items that mention (or “invoke”) it.

5.1. Defining and invoking templates

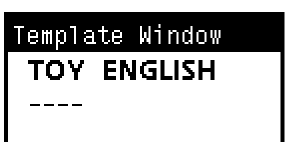
To see how a template can express a generalization, consider the requirement shared by all Englih common nouns that their singular forms must appear with an explicit determiner. This requirement could be imposed in the lexical entry for `girl` by adding the following disjunction:

```
{(↑ NUM)=SG (↑ DEF)|(↑ NUM)=PL}
```

to produce the extended entry:

```
girl N S (↑ PRED)='GIRL'
         { (↑ NUM)=SG
           (↑ DEF)
          |(↑ NUM)=PL}.
```

The new disjunction asserts that either the number is singular and some value exists for the feature `DEF`, or the number is plural. This same disjunction could also appear in the definitions of all other common nouns, but instead we incorporate it into the definition of a `CN` (for Common Noun) template. A template is defined and edited in a Template Edit window. Although template windows are not visible in GWB’s initial screen configuration, the title-bar menus of most of the other windows contain an `Edit Template` item that can be used to make a template window appear. If you select the `Edit Template` item from, say, the Sentence Input Window, another menu will pop up with the single item `New`. If you release the mouse over that item, you will be asked for the region of a new template window. GWB will initialize that window by inserting `TOY ENGLISH` as the default version and language for the template you are about to define:



As usual, use the left button to position the input caret before the line of hyphens and type in the following definition for the `CN` template:

```
CN = {(↑ NUM)=SG (↑ DEF)|(↑ NUM)=PL}.
```

When you type `CTRL-X` to install the template, the definition will be reprinted in the usual indented format:

```
Template Window
TOY ENGLISH
   CN = { (↑ NUM)=SG
          (↑ DEF)
         |(↑ NUM)=PL}.
```

This definition of the CN template has now been entered into GWB's internal database, and it has also been activated so that it can be invoked in various lexical entries. The template was activated because when GWB earlier constructed the default TOY ENGLISH configuration, it anticipated that you would eventually define and want to make use of some (TOY ENGLISH) templates. As you can see by looking at the configuration display above in Section 3.3, (TOY ENGLISH) is specified in the TEMPLATES part of the configuration.

The lexical entry for girl now can be simplified by making use of the CN template. Bring the entry into a Lexicon Window and add @CN in place of the explicit disjunction. The entry will now be shortened to

```
girl N S (↑ PRED)='GIRL'
         @CN.
```

This shows that a template can be invoked in a position where an ordinary schema might otherwise appear simply by writing the name of the template preceded by the character @. You can see the schemata that this invocation is equivalent to by clicking the mouse over the name of the template and typing CTRL-F (for Functional template). A new "Template expansion" window will appear on the screen to show the schemata that result from expanding that invocation of CN:

```
Template expansion
@CN:
 { (↑ NUM)=SG
   (↑ DEF)
  |(↑ NUM)=PL}
```

Other common nouns can also share the schemata included in the template definition. By defining boy like girl, for example, you will obtain the following lexical entry:

```
boy N S  (↑ PRED)='BOY'
         @CN.
```

If at some point in the future you want to extend or modify the common noun requirements, you can edit just the definition of the template and the change will take effect for all the lexical entries that invoke it.

The template mechanism can be used to simplify somewhat further the entries for all common nouns. In addition to the determiner/number interaction already expressed in the CN template, all common nouns also include a schema defining their particular PRED. By providing the semantic predicate as a "parameter" of the common-noun template, you can also factor from individual entries the details of even the PRED specification. Go back to the Template Window containing the CN definition. (If it is no longer on the screen, you can bring it up by choosing CN from the menu that now appears when you select an `Edit Template` item; alternatively, you can click to place the type-in caret over the `CN` characters in one of the lexicon windows and then type `CTRL-S` (for S _how)).

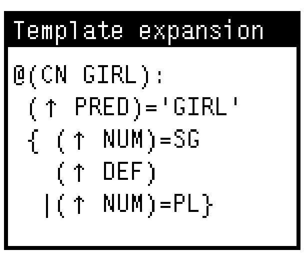
You need to indicate that the predicate is going to be provided as a parameter at each invocation of the template, and that the given predicate is then to appear in the semantic-form of a `PRED` schema. You must specify a parameter-name, say `P`, to stand for the predicate that will be supplied each time, and use that in a generalized `PRED` schema. You specify that `P` is the parameter name by including it in parentheses between the template name and the following equal sign. You then add the generalized `PRED` schema to the body of the template definition. The result is the following definition:

```
CN(P) = (↑ PRED)='P'
        {(↑ NUM)=SG
         (↑ DEF)
        |(↑ NUM)=PL}.
```

Having installed this definition, you must then change all of the common-noun entries so that they no longer specify their idiosyncratic `PRED` schema and instead supply their particular predicate as an argument to the `CN` template. The entries for `girl` and `boy` simplify to

```
girl N S  @(CN GIRL).
boy  N S  @(CN BOY).
```

When a parameterized template is invoked, the template name and the values for its parameters are enclosed in parentheses after the `@`. You can see the effect of this parameterization by again clicking on one of the template invocations and typing `CTRL-F`. The template expansion window will show how GWB interprets the invocation:



Section 5 of Chapter III provides a complete description of how parameterized templates are defined and invoked, including a discussion of how they can implement the lexical redundancy rules originally proposed by Kaplan and Bresnan (1982). Section III.4.1 describes a special designator, %stem, that can be used to simplify template invocations in lexical entries.

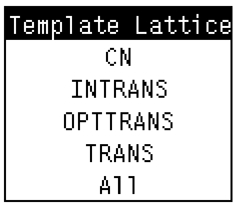
5.2. Displaying the template lattice

A template can be invoked in any position where an ordinary schema might otherwise appear. Templates can thus be invoked in lexical entries, as you have just seen, as well as from the functional annotations on c-structure rules. It is also possible for the invocation of one template to appear in the definition of another one, so that the effect of the first one is included whenever the second one is invoked. The pattern of inter-template references typically forms a hierarchy that can encode and organize families of linguistic generalizations. Templates in GWB can thus play the same explanatory role that inheritance of typed feature structures plays in Head-Driven Phrase Structure Grammar (Pollard & Sag, 1994). This is true even though templates are purely abbreviatory devices and do not require the deep mathematical analysis that type inheritance seems to call for. To assist in understanding the template-reference relations in a complicated system of definitions, GWB provides a simple facility for displaying the lattice of template invocations.

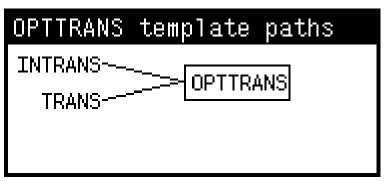
You should define a few more templates with more interesting reference relationships in order to experiment with the lattice display. Begin by defining templates just for the PRED schemata of intransitive and transitive verbs and another template that invokes these to provide for verbs that are optionally transitive:

```
INTRANS(P)   = (↑ PRED)='P<(↑ SUBJ)>'.
TRANS(P)     = (↑ PRED)='P<(↑ SUBJ)(↑ OBJ)'.
OPTTRANS(P)  = {@(INTRANS P)|@(TRANS P)}.
```

After typing CTRL-X to install these, click in the title-bar of either the Sentence Input Window or a Template Window and select the Template Lattice menu item. A menu of the currently active templates will pop up:

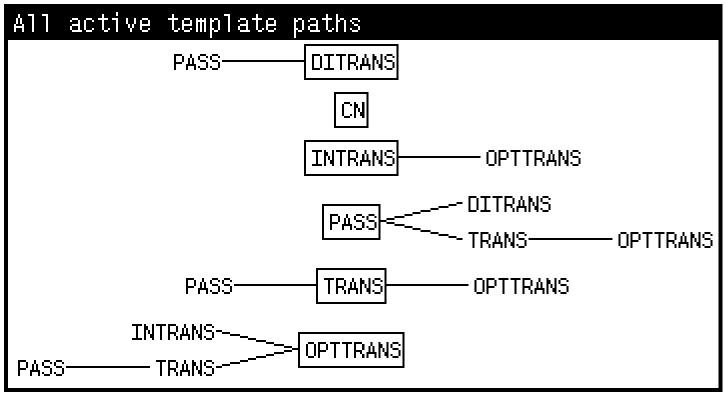


If you choose the OPTTRANS item, you will be shown the pattern of references as seen from that template's point of view:



Each node of the graph represents one of the active templates. Templates to the right in the graph invoke (and thus inherit information) from templates that they are linked to and that appear further to the left. Thus, this graph shows that OPTTRANS invokes both INTRANS and TRANS and, since there are no nodes to its right, that OPTRANS is currently not invoked by any other template. The OPTTRANS node has a black border to indicate that it is the focus of this particular graph. Active templates that are not related to OPTRANS, such as CN, are not shown in this display.

Selecting the All item at the bottom of the menu results in a larger graph showing the relationships for all the active templates. This is illustrated in the window below, which shows a more complicated configuration of template references:



This shows the lattices for all active templates arranged vertically in alphabetical order. At the bottom you see the simple lattice for `OPTTRANS`, with the further elaboration that `TRANS` now invokes the `PASS` template, providing for the passive alternation. The `DITRANS` template also invokes `PASS`, as can be seen from the `DITRANS` lattice. The lattice focused on `PASS` shows in a single place that `PASS` has two consumers. Finally, the isolated node for the `CN` template indicates that it neither invokes nor is invoked by any other template—it is currently referenced only by lexical entries.

Of course, the graphs for still more complicated template systems will not all be visible at the same time in the window. You can use the window's left and bottom scroll bars to navigate among the different regions of the graph, or you can reshape the window to make it bigger. You can also use the mouse to pick out particular lattices and have them displayed in separate windows. The nodes in a template lattice are mouse-sensitive: clicking on a node will display in a separate window the lattice that has the new selected node as its focus. Thus if you click middle or left on one of the `OPTTRANS` nodes in the `All` graph, you will be prompted for a new window region and then see in the new window just the graph showing paths from the `OPTTRANS` perspective. You can also click on nodes in such a single-lattice window, but in this case the effect depends on which mouse button you use to make your selection. If you select a node, say `TRANS`, with the left button, the lattice in the window will be replaced by the `TRANS` graph. But if you click with the middle button, the `TRANS` graph will be shown in a new window. This arrangement permits you to view several different graphs at the same time, and a sequence of node selections enables you to quickly explore various parts of the lattice without scrolling. Finally, if you come to a template whose definition you want to see or edit, you can bring it into a template editing window by holding down the `CTRL` key while clicking on one of its representative nodes. The `CTRL`-select convention is also used in the c-structure window to bring up a rule editor for a particular category.

6. Saving grammars on permanent files

The rules, lexical entries, morphology table, and templates that you have defined have been installed into GWB's internal database. They can be retrieved from this database for editing or activated for sentence analysis. However, this database exists only in your workstation's current memory. If you turn off your machine or leave GWB (for example, rebooting your machine or invoking Medley's `LOGOUT` command), all of your specifications will be forgotten. To preserve your grammar for future use, you must transfer it to a file on some permanent storage device—your workstation's hard disk, or perhaps a floppy disk or file server. The naming conventions for files may depend on the operating system or network environment you are running in, and for the most part you can use those conventions inside Medley. Medley also supports its own internal conventions and attempts to translate automatically between different file-name patterns. The examples in this section will use the internal convention whereby a file-name is an atom or string of the form

{device/host}<directory>subdirectory>...>name.extension;version

Thus, the name {DSK}<LFG>ENGLISH>TOYGRAM;5 denotes a file on the ENGLISH subdirectory of the LFG directory of the DSK device, the workstation's disk. The CORE device holds files internal to Medley's virtual memory and are thus not suitable for long term storage. Files created on the LPT device will be sent to a printer rather than saved; printing can also be accomplished by selecting the Hardcopy item in window and background command menus.

On Unix and MS-DOS systems Medley allows for a few more file-naming options. The device DSK denotes the workstation's root-directory. On Unix, other hosts on the network can then be referenced as if they were separate directories; on MS-DOS systems other devices are accessed using the standard device-letter conventions. Directories and subdirectories can also be separated by / instead of < and >. From Medley's point of view, file names on DSK are case-insensitive and have version numbers. If you search for a file created by Medley using Unix tools outside of Medley, you must be sure that case and version numbers match appropriately. The device {UNIX} is also provided as a variant of DSK with the property that its filenames conform exactly to the standard Unix conventions.

6.1. Tedit PUT and GET

The most straightforward way to create files containing your grammatical specifications is to use TEdit's PUT command. To save some rules, for instance, first bring them into a Rule Window, as if you were going to edit them. Then click the mouse in the window's title-bar to bring up the rule-editing menu and select the TEdit Menu to get the TEdit command menu. This has the normal PUT, GET, and other commands described in the TEdit documention. Select PUT and type in a name (e.g. TOYRULES) for the file that you want the rules to be saved in. The file will be created on the directory you are current connected to and your rules will be transferred into it. Later, perhaps in a subsequent session, you can use the TEdit GET command (again, bring up the normal TEdit command menu by selecting the TEdit Menu item) to bring those rules back into an empty rule window and reinstall them into GWB's internal database by type CTRL−X. This procedure can be applied to lexical items, configurations, or other specifications by invoking the PUT and GET commands on windows of the appropriate type.

While this is the simplest way of saving your grammar, it is not the best method to use as you develop more complex arrangements of versions and languages. One difficulty is that the TEdit files created in this way can only contain one kind of specification. Your TOY ENGLISH rules and lexical entries must be saved on separate files, since they are displayed in windows with different CTRL−X behaviors. It is all too easy to lose track of where you have saved compatible sets of rules and lexical entries. A second difficulty with this method is that it requires you to remember exactly which files need to be updated as you introduce new rules or modify old ones. If you forget about one of the improvements you have made and do not explicitly PUT it back to a permanent file, the results of your efforts will be lost.

These difficulties are not unique to the problem of keeping track of and saving grammatical specifications. The same issues arise in any complex computing environment where the various programs and data structures modified during the course of a session must be preserved in permanent file storage. Medley comes with an elaborate facility for automatically doing the bookkeeping that protects programmers from making these kinds of mistakes. GWB extends the Medley facility, called the File Manager (or File Package in earlier releases) so that it offers the same degree of protection to grammar writers. The File Manager is described in great detail in the Medley Reference Manual. The following paragraphs outline the simple File Manager scenarios that will typically come up during grammar development.

6.2. The File Manager: Cleanup

GWB notifies the File Manager whenever you install a new rule, lexical entry, etc. or modify and reinstall a previously existing one. The File Manager keeps track of this information in its own special data structures. You can issue commands to the File Manager either to see a list of the changes you have made but not yet transferred to permanent files or to cause the necessary transfers to take place nearly automatically. GWB makes the most common File Manager commands available as sub-items in the LFG Logo menu. File Manager commands can also be typed in to a Lisp Executive Window, obtained, if one is not already on the screen, by selecting the EXEC item in the right-button background menu.

Click left in the LFG Logo window to bring up its menu, move the cursor to the File Manager item, and slide the mouse to the right. A submenu will appear with items labeled Cleanup, Files?, and Load. Releasing the button over the Cleanup item starts the procedures necessary to move your recent internal changes to permanent storage locations. The Files? item informs you of the current state of the changes you have made, indicating what actions would be needed to get to the “clean” state in which your permanent files are consistent with the information in the internal database. The Load command is used to bring previously saved grammars back into the internal database.

Your goal now is to save the rules and lexical entries you have been working on, so Cleanup is the appropriate choice. When you release the mouse button over Cleanup, a new window opens in the middle of the screen. This is where your interactions with the File Manager will take place. The File Manager first enumerates the names of all the rules, lexical entries, morphology tables, and configurations that you have installed but not yet associated with a permanent file. With the TOY ENGLISH and TRINKET ENGLISH grammars you have defined, the window will look like this:

```
File Manager Window
The following are not contained on any file:
  the morphology tables: (TOY ENGLISH)
  the lexical entries:
         (TOY ENGLISH -NPL -NSG -V3SG -VINF -VPAST girl go goes
 gone kick saw see- the walks went)
  the rules: (TRINKET ENGLISH NP), (TOY ENGLISH NP S VP)
  the configurations: (TRINKET ENGLISH), (TOY ENGLISH)
want to say where the above go ?
```

After listing the various items, the File Manager has asked you whether you want to make those associations, that is, to assign a file name to each of the items. If you answer N (for No) to this question, it will leave these items in their unassigned condition to be asked about again the next time you select Cleanup or Files?. If you answer Y (Yes), the File Manager will print each item in turn and blink the input caret while it waits for you to specify an assignment. Let’s suppose that you want to store all the TOY ENGLISH items to the file named TOYGRAM on your currently directory, {DSK}<LFG>ENGLISH>, but that the TRINKET ENGLISH NP rule is to be stored in TRINKETGRAM on another directory named {DSK}<MYFILES>ENGLISH>. After typing Y to the question, GWB will identify the kind of item it will first ask about (morphology tables) and then print its name:

```
(morphology tables)
(TOY ENGLISH)
```

It then waits for you to indicate the name of the file you want the TOY ENGLISH morphology table to be stored on. The most straightforward response you can give is simply to type the name of the desired file, TOYGRAM, terminated by a carriage return. Since this is a file previously unknown to GWB, it asks you to confirm that you really do intend a new file to be created, that you haven't simply mistyped another name. The dialog, with your responses underlined, looks like this:

```
want to say where the above go ? Yes
(morphology tables)
(TOY ENGLISH)  File name:  TOYGRAM
create new file TOYGRAM ?
```

You confirm by typing Y again, and GWB proceeds to the next item. Notice that the name you enter is the so-called "root" name of the file, a single string without spaces or other punctuation (unlike a version/language pairs that GWB uses in its internal database). Also, the root name is without any identification of the storage device or directory that you want the file eventually to reside on. At this point you are merely setting up the associations between items and file names. The storage device is determined a little bit later, when the file is actually written out.

The next items to be processed are the lexical entries. Again, a line identifying the kind of item is displayed, followed by the name of the first one:

```
want to say where the above go ? Yes
(morphology tables)
(TOY ENGLISH)  File name:  TOYGRAM
create new file TOYGRAM ?  Yes
(lexical entries)
(TOY ENGLISH went)
```

At this point you can again type TOYGRAM, or you can type the LINEFEED key (LF on some keyboards or CTRL-J for keyboards without a separate line-feed key) to indicate that you want this item to be assigned to the same file as the last one. If you type LF, GWB will print TOYGRAM to indicate its interpretation of your response. This is the first TOY ENGLISH lexical entry you have been asked about, and since it is almost always the case that you will want all entries in a given version and language to be stored on the same file, GWB prints another line to inform you that it is making that assumption:

```
want to say where the above go ? Yes
(morphology tables)
(TOY ENGLISH)  File name:  TOYGRAM
create new file TOYGRAM ?  Yes
(lexical entries)
(TOY ENGLISH went)  <LF>  TOYGRAM
Note:  All TOY ENGLISH lexentries will go on TOYGRAM
```

This indicates that GWB is associating all existing TOY ENGLISH lexical entries with the TOYGRAM file and also any TOY ENGLISH lexical entries that you might define at a later time. It then begins to ask you about c-structure rules:

```
(lexical entries)
(TOY ENGLISH went)  TOYGRAM
Note:  All TOY ENGLISH lexentries will go on TOYGRAM
(rules)
(TOY ENGLISH VP)
```

Here again you can type LINEFEED to continue with TOYGRAM, and you are again informed that all TOY ENGLISH rules will be assigned to that file. You will next be asked about the TRINKET ENGLISH NP rule. Since this one is supposed to go on a different file, TRINKETGRAM, you must spell out its name completely and then confirm that you also want it to be created as a new file. Finally, you will be asked about the TOY ENGLISH configuration. You cannot use the LINEFEED abbreviation to assign this to TOYGRAM, because TRINKETGRAM has become the most recently entered file name. You must retype TOYGRAM in its entirety:

```
(rules)
(TOY ENGLISH VP)  <LF> TOYGRAM
Do you want all TOY ENGLISH rules to go on TOYGRAM?  Yes
(TRINKET ENGLISH NP)  TRINKETGRAM
create new file TRINKETGRAM ? Yes
Note:  All TRINKET ENGLISH rules will go on TRINKETGRAM
(configurations)
(TOY ENGLISH)  File name:  TOYGRAM
```

This completes the process of associating your newly defined items with file names. During this process, GWB did not actually transfer your grammar to the files that you indicated. What it did was to construct and modify a "table of contents" for each of those files, a set of instructions or commands to be executed now, when the assignments are complete. At this point the cleanup continues by actually transferring the information, or, in Medley parlance, by "dumping" the files. It will make files with the names you specified, placing them on your default storage device and directory, the one you are currently "connected" to.

A connected directory was established for you when you started the session based on the value of the Interlisp variable LOGINHOST/DIR. This will be {DSK} unless you constructed a personal initialization file (or "Greet-file") that changes the start-up value of this variable. It may be somewhat confusing, but note that LOGINHOST/DIR is not necessarily the same as the Unix or DOS working directory from which you started GWB. While you are using GWB, you can at any time change the currently connected directory by means of the CD or CONN commands in the Lisp Executive Window: if you are connected to {DSK} or one of its subdirectories, for example, typing CD <LFG>ENGLISH> will change your connected directory to {DSK}<LFG>ENGLISH>. In general the host or device of the new connected directory is the same as the previous one unless you explicitly provide a new host in the command (CD {CORE}<DIR1>). The CD and CONN commands interpret many of the relative directory addressing conventions of the Unix and DOS operating systems. If you are connected to {DSK}<LFG>ENGLISH>, then CD ../FRENCH will leave you connected to {DSK}<LFG>FRENCH>.

Suppose now that you are currently connected to the <LFG>ENGLISH> directory on your workstation's local disk {DSK}. GWB will assume that this is where you want the new files made, but will ask you for confirmation. It chooses TRINKETGRAM to ask about first:

```
(configurations)
(TOY ENGLISH)  File name:  TOYGRAM
Make TOYGRAM on {DSK}<LFG>ENGLISH> ?
```

Type `Y` as your confirmation, and GWB will make the file for you, printing out a number of messages about the file as it is constructed. When it is finished with `TOYGRAM`, it prints the full name of the file it has created (including a version number). It then moves on to the next file, asking whether `TRINKETGRAM` should also go on `{DSK}<LFG>ENGLISH>`. Since you want `TRINKETGRAM` to be stored on the directory `<MYDIR>ENGLISH>` instead, don't confirm with `Yes` but type `C` and then `<MYDIR>ENGLISH>`:

```
Make TOYGRAM on {DSK}<LFG>ENGLISH> ?  Connect to directory <MYDIR>ENGLISH>
```

This changes your connected directory conveniently, without requiring a separate `CD` command in the Executive Window. The file will be made on the new connected directory, and GWB will report back in the window when it has finished. At this point your cleanup is complete—all the grammars in the internal database now also reside on permanent storage. If you want to remove the File Manager window from the screen, right-click in it and select the `Close` command from the window command menu.

We have described one straightforward path through the `Cleanup` dialog, ignoring some other possible scenarios. For example, suppose that you want the File Manager to simply forget about some item that you defined either accidentally or for temporary experimentation, without assigning it to a permanent file. When it asks about that item, you can respond with a closing square bracket `]`, which is interpreted as an assignment to "Nowhere". The square-bracket is used in this arcane notation because it is non-alphabetic and therefore cannot be confused with a character in a real file name. If you want the File Manager to skip an item, leaving it unassigned until a later dialog, you can respond by typing a space character to indicate "no action". If you want to assign an item to the same file that another item of the same type belongs to but you don't remember that file's name, you can type `@` plus the name of the other item, and the File Manager will determine the name of the file for you. You can also respond with `?` to see a helpful summary of the complete list of possible responses, including some that are described fully in the Medley Reference Manual:

```
[ -- prettyprint the item to terminal and then ask again
{line-feed} - same as previous response
{space} - no action
] - nowhere, item is marked as a dummy
(list name)
@ other-item   -- put the item near the other item
a file name
```

If you change your mind while you are making an assignment, you can type `CTRL-Q` to restart the response to that item. At any time you can type `CTRL-D` to terminate the dialog entirely (but the assignments you have already made will remain in effect).

When all the assignments are done and the File Manager asks you to confirm the location that a file will be dumped on, you have another option besides confirming the default location or connecting to another directory. You can enter `No`, which indicates to the File Manager that you do not want the file to be dumped at this time. The File Manager will skip over that file but remember that modified definitions are associated with it and have not yet been dumped. You will be told that the file needs dumping whenever you use `Files?` to interrogate the File Manager, and you will be asked again where it should be dumped every time you execute the `Cleanup` command, until it is finally written out to some storage device.

This example has also dealt only with newly defined items and newly created files. If you edit an item that was previously assigned and dumped to a file, the File Manager notes that the information assigned to the file has been modified so the file needs to be dumped again.

Cleanup in this case does not request a new assignment, asking only for confirmation of the directory that the file is to be made on. If there are no newly defined items and no changes for a file, Cleanup will take no action on it. Thus, it is always safe (and wise) to invoke Cleanup at the end of every session, to avoid losing any stray, unsaved rules, lexical items, or templates.

6.3. Files? and Load

The major function of the Cleanup command is to move all recently modified information to permanent files. It enters the interactive assignment dialog described above only if there are items unassigned to files, and then it proceeds to dump all necessary files. The Files? command, also available by sliding the mouse to the right of the File Manager menu item, allows you merely to interrogate the current state of your modifications and their file-assignments, without making any new assignments or transferring any information to files. In effect, it executes only the first part of the Cleanup sequence. If you select Files? instead of Cleanup, you will be told the names of already existing files that contain items that you have recently modified and which therefore need to be—and would be by Cleanup—dumped. If there are also unassigned items, you will be asked whether you want to begin the assignment dialog. This gives you the opportunity to associate items with files while your filing strategy is fresh in your mind, but postpones actually creating the files until later, when your modifications have reached a consistent state.

The Load command is the opposite of Cleanup: it is used to bring previously saved grammars back into GWB's internal database. When you slide off the File Manager item and select Load, you will be asked in the File Manager window to supply the name of the file you want loaded:

```
Name of file to load:
```

If, in a subsequent session with GWB, you want to restore the grammar you saved on {DSK}<LFG>ENGLISH>TOYGRAM, you would respond by typing that file-name and a carriage-return. If you are currently connected to {DSK}<LFG>ENGLISH>, then you need only type TOYGRAM to uniquely identify the file. Either way, the loading process will begin to copy the grammar on the file into the internal database. During this process, messages will be printed in the File Manager window, telling you the name of the file, the time it was created, and perhaps a brief indication of what it contains. If the messages reach the bottom of the window, the system may turn the window from white to black and suspend the loading process. It is waiting for you to confirm that you have seen the messages already in the window before it scrolls them away to make room for the remaining messages to be displayed. You can give this confirmation by typing any key (e.g., the space bar), and the loading process will resume. When the file has been loaded, the message Done will appear in the File Manager window. The information in the file is now available through the standard menus for editting. If the names of any loaded items are included in the active configuration, those items will become part of the active sentence analysis environment. And any configurations that were loaded are now available for activation, through the Change Config menu item described in Chapter II.

6.4. Manipulating managed files

The files created by the File Manager are ordinary Lisp source files of the sort that typically contain the definitions of Lisp functions, variables, and other program data structures. Medley provides a large number of tools besides Cleanup, Files?, and Load for manipulating these files and the items that are associated with them. These tools are accessed primarily by invoking particular Interlisp functions in an executive window (usually a thin window overlapping with the Sentence Input Window). For example, entering

```
(WHEREIS '(TOY ENGLISH S) 'RULES)
```

at the Executive will print `TOYGRAM`, the name of the file that that particular rule has been assigned to. The atom `RULES` in this example is the file-manager “type” of the item you are interested in; other GWB-specific types are `LEXENTRIES`, `CONFIGS`, `TEMPLATES`, and `MORPHTABLES`, and these must be given to direct the operation of the various functions. Thus, to find out what file the lexical entry `(TOY ENGLISH walk)` is assigned to, you can enter `(WHEREIS '(TOY ENGLISH walk) 'LEXENTRIES)`.

A few other File Manager functions are worth mentioning. You can associate items with a file directly and then causes it to be dumped, without going through the `FILES?` and `CLEANUP` dialogues. This is accomplished by the Interlisp function `FILE`. `FILE` takes the root name you want to give to a new file and then a sequence of commands that describe what it is supposed to contain. A command is a parenthesized expression beginning with the type of the items (`RULES` etc.) and followed by the names of the particular items. Try entering the following into the Executive Window:

```
(FILE DISCUSS (RULES (TOY ENGLISH S VP) (TRINKET ENGLISH NP))
        (LEXENTRIES (TOY ENGLISH  girl)))
```

This will create a new file named `DISCUSS` that will contain the `S` and `VP` rules from `TOY ENGLISH`, the `NP` rule from `TRINKET ENGLISH` and the `TOY ENGLISH` lexical entry for `girl`. These items will still belong to the files you originally assigned them to, but this selection of items will also belong to this additional file. When you make changes to these items, you will be told that the original files `TOYGRAM` and `TRINKETGRAM` need to be dumped, as well as the new file `DISCUSS`. If you provide `FILE` with only the name of an existing file and no other commands, it will merely dump the items you have already associated with that file.

The Interlisp function `DELFROMFILE` can be used to break the association between a particular rule, lexical entry, etc. and the files that it is assigned to, `DELDEF` acts to delete the item in addition to deleting its file assignment (this effect is also achieved by erasing in an editing window the characters that make up the items definition). `MOVETOFILE` can be used to remove an item from one file and assign it to another in a single step. Each of these functions takes two arguments, the name of the item (or items) plus the type of the item . Thus, `(DELDEF '(TOY ENGLISH NP) 'RULES)` will remove the definition of that rule from GWB’s internal database. The functions that operate on files can also take as an optional third argument the root name of the file that you want to remove the item from. This is useful if an item has become associated with two different files and you only want to eliminate it from one of them. Remember that these functions do not actually change the disk file; they change the specification of its contents. The file itself reflects these changes only after it has subsequently been dumped (e.g. via `CLEANUP`).

The Medley structure editor (SEdit) can also be applied to make idiosyncratic changes to the data structure that records the items assigned to a particular file. This is accomplished by invoking the `DC` command (for D_isplay-edit the C_ommands) on the root name of the file: Typing `DC TOYGRAM` will bring up an SEdit window showing the list of commands describing what information is contained in that file (provided that the file has first been loaded so that its contents exist in GWB’s memory):

```
SEdit TOYGRAMCOMS  Package: INTERLISP
((LEXENTRIES (TOY ENGLISH))
 (RULES (TOY ENGLISH))
 (CONFIGS (TOY ENGLISH))
 (MORPHTABLES (TOY ENGLISH)))
```

SEdit provides a simple mouse and menu editor for changing these structures, using many of the same conventions as TEdit. Unlike TEdit, the mouse is used to select symbols and structures (parenthesized lists), not characters, words, and lines, and double-clicking causes the selection to move up to the next enclosing level of structure. Clicking `LEFT` or `MIDDLE` in the title bar of the window will bring up a menu listing a number of basic SEdit commands. As one application, you can use SEdit to merge the contents of two files together: `DC` both files to get their commands in separate editing windows, then add the commands of the first one to the end of the commands of the second one. Point and click `LEFT` where you want the additional commands to show up in the second file's command list, then hold the `SHIFT` or `COPY` key while you select commands from the first window. Release the `SHIFT` key when your selection is complete. At the end of the edit, type `CTRL-X` to exit the SEditwindows or simply use the `Close` item in the window's right-button title-bar menu. Changes to the file's contents will appear the next time you dump the file.

The many other capabilities of the File Manager and of SEdit will probably not be needed for straightforward linguistic uses of GWB, so they will not be further described here. The Medley Reference Manual should be consulted for a full discussion of these facilities.

7. Encoding and typing of special characters

The formal notations of rules, lexical entries, and other specifications include special characters that are not found on standard keyboards, and the words for languages other than English typically include accents, characters, or entire alphabets that are not usually available. *Internal* to GWB, these characters are currently represented by the unique 16-bit codes defined in the Xerox Character Encoding Standard [Xerox Systems Institute, 1987]. The first 256 codes of this specification are identical to the ISO 646 character standard, a superset of conventional ASCII that includes most of the characters used for Western European languages. The Xerox encoding is honored by the Classic, Modern, and Terminal font families for both display purposes, although some character images may not exist for some fonts and sizes. The internal character encoding is likely to be of little interest to you, unless you are writing your own Lisp programs to operate on GWB's internal data structures.

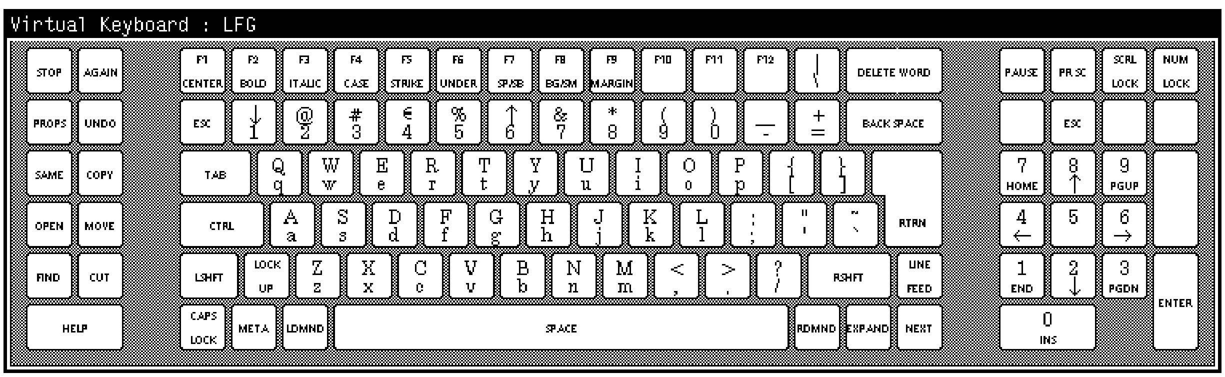
The correspondence between the internal encoding and the *external* encoding of characters for printers and on files is probably much more important, since those representations may be interpreted by other devices and by other programming systems. When GWB formats information for transmission to a printer, it converts from the internal encoding to the encoding used by that particular printing device. Xerox Interpress printers also use the Xerox encoding standard, so their conversion is transparent. Conversion software for PostScript printing is provided by a Medley Library package that is included in GWB, and the library also contains converters for a number of less popular printers. However, not all characters and alphabets in the Xerox character set are available for Postscript and other printers, and you may have to do some tailoring to your local printing conventions.

When GWB reads and writes information to character-oriented files, it also uses the Xerox encoding standard as a default. Methods for converting to and from other commonly used standards (ISO8859/1, Unicode, IBM-PC, etc.) are under development; these will permit files created by non-Xerox software to be converted to the Xerox standard as GWB reads them and converted from the Xerox standard as GWB writes them.

7.1. Virtual keyboards

You have already seen that some special characters in rules and lexical entries can be entered by typing substitute characters that appear on standard keyboards; these are converted to their proper internal encodings when `CTRL-X` is typed to cause the items to be read and redisplayed. Thus, ! and ^ may be typed for ↓ and ↑, respectively, and -> may be used instead of the right-arrow → in rewriting rules. GWB also includes the Virtual Keyboards package from the Medley Library to provide a more systematic and convenient way of entering unconventional characters. The features of the Virtual Keyboards package most important to GWB are outlined below; a full technical description can be found in the Medley Library documentation.

The Virtual Keyboards package allows you to change the behavior of the keyboard that is physically attached to your computer so that it simulates a keyboard with different key-labels (hence making yours a “virtual” version of that other keyboard). It also allows you to display images of other keyboards on your screen and use them as menus for “typing” occasional special characters. For example, one of the virtual keyboards installed in GWB is the `LFG` keyboard. This defines a slightly modified default keyboard behavior that governs typing in to an LFG rule or lexicon window. This keyboard has the following image as laid out for the Sun Type 4 physical keyboard:



This indicates that, when this keyboard is active, typing the shift-4 combination transmits the set-membership character ∈ rather than the usual dollar-sign, and shift-1 transmits the down-arrow. When this keyboard is displayed on the screen, whether or not it is active, the set-membership can be "typed" in by holding the shift-key down on the physical keyboard and clicking the mouse over the image of the 4 key.

Whether or not the location of keys and shifts on a virtual keyboard conforms to the layout of your physical keyboard depends on whether or not a "keyboard configuration" file that describes your physical keyboard is available to GWB. Keyboard configuration files currently exist for the various Xerox work-stations, for the Sun Type 3 and Type 4 keyboards,and for conventional IBM-PC keyboards; other configuration files can be created by using the Keyboard Editor package in the Medley library. When a file describing your actual keyboard is not available, the Virtual Keyboard package is not able to modify the behavior of your physical keyboard, and it displays keyboards according to the Sun 4 layout pictured above.

The display and activation of virtual keyboards is controlled by the `Keyboard` item that appears in both the window and background right-button menus. Clicking on this item in either menu brings up a menu of currently defined keyboards, for example:

```
Quit
DEFAULT
DVORAK
EUROPEAN
FRENCH
GERMAN
GREEK
ITALIAN
LFG
LOGIC
MATH
OFFICE
SPANISH
STANDARD-RUSSIAN
```

Selecting one of these keyboard-names causes the behavior of that virtual keyboard to be "switched" in to the physical keyboard, thus modifying the characters that are transmitted when keys are pressed. The European keyboard has the various accented Roman characters found in many Western European languages, while the French, German, and Spanish keyboards have layouts that are found on typical keyboards in the corresponding countries. The Greek keyboard has the full Greek alphabet and is useful not only for typing strings in that language but also for entering LFG projection designators. The Logic, Math, and Office keyboards have a variety of special technical symbols. The Default keyboard reestablishes the behavior shown on the physical key caps, and of course, the `Quit` menu item (or releasing the mouse button outside the menu) allows you to exit without making a keyboard selection.

When the `Keyboard` item is chosen from the right-button menu in a window, the effect is to change the keyboard behavior for any typing directed towards that window. Different windows can have different keyboards installed, and whenever you click to change type-in from one window to another, the keyboard behavior will automatically change. When the `Keyboard` item is chosen from the right-button menu in the background, the effect is to change the systems default keyboard, the one that is installed automatically when a window is created that does not have an otherwise specified keyboard. In particular, new Executive, TEdit, and graphics windows will be initialized with this keyboard. LFG rule- and lexicon-editing windows are always initialized with the LFG keyboard, however, and the keyboard for the Sentence Input window is determined by the `INPUTKEYBOARD` specified in the currently installed parsing configuration (the default keyboard is used if the configuration does not include such a specification).

7.2. TEdit abbreviations

The TEdit text editor provides an abbreviation facility that offers another method for entering certain predefined special characters. These characters have been assigned names that are made up of ordinary Ascii characters. They can be entered in a TEdit window (which includes LFG rule and lexicon windows, the Sentence Input window. and any of your own text-editing windows, but not the Lisp Executive window) by typing the name of the character and then pressing the `Expand` key on the keyboard. The name you typed will be removed and replaced by the desired character. The LFG characters and the names by which they can be entered are as follows:

∈	`mem` *or* `$`
→	`ra` *or* `->`
↓	`da` *or* `!`
↑	`ua` *or* `^`
←	`la` *or* `<-`

Thus, the set-membership symbol can be entered by typing the sequence `mem` and pressing `Expand`. The location of the key that represents the `Expand` function varies among different keyboard types but can be discovered either from Medley system documentation or by experimentation. On a Sun Type 3 keyboard, it is the key labeled `Right` that is immediately to the right of the space bar. The abbreviation facility is implemented by means of the Lisp variable `TEDIT.ABBREVS`, which is described in the TEdit documentation (and included in the Appendix). As you have seen, the character-names in the right-hand column may be used in the appropriate rule and lexical entry contexts without using the `Expand` key; the characters are expanded automatically when `CTRL-X is` pressed (Note that in Text as opposed to LFG-editing TEdit windows, `CTRL-X` is also just a synonym for `Expand`; in LFG windows it instead initiates the process of internalizing your linguistic specifications).

7.3. Projection designators

As discussed in Chapter III, projections or structural correspondences are named by characters in the Greek alphabet, which the Xerox Character Encoding Standard assigns to 16-bit codes in the range 9728 to 9983 (decimal). Although these can be entered by means of the Greek virtual keyboard as indicated above, the system provides a more direct way of specifying them in contexts where a projection designator makes sense. Any Greek character can be entered simply by typing its common Ascii name. Thus, you can enter lower-case σ by typing `sigma` or `SIGMA`, τ by `tau` or `TAU`. You can enter an upper-case Greek letter by capitalizing only the first letter of its name (`Sigma` for Σ). There are also one-letter abbreviations for most of the characters, but only when a double-colon is provided to explicitly mark that letter as designating a projection. Thus the sequence `t::s::`↑ would be interpreted as τσ ↑. Each of the following lower-case letters can be entered by the first character of its name: alpha, beta, gamma, delta, epsilon, zeta, iota, kappa, lambda, mu, nu, xi, omicron, pi, rho, sigma, tau, upsilon, and chi. In addition, eta, phi, and omega can be abbreviated by h, f, and w respectively (the letters e, p, and o are used for other characters). The upper-case version of a character can be entered by upper-casing its lower-case abbreviation.

8. Exiting: How to leave Medley

When you have reached the end of your GWB session, you can leave Medley by invoking the Lisp function `EXIT`. Click in the Executive window and type in `(EXIT)`. After a few seconds, GWB will disappear from your screen, and your system will return to the point at which you originally entered Medley. Before exiting, GWB first checks to see if you have any unsaved changes, and if so, asks you whether you want to first dump those items or whether you want to exit anyway. If you exit without saving, your changes will be lost forever. If you are sure in advance that you have no interest in your changes, you can invoke `EXIT` with a parameter `FAST`, as in `(EXIT FAST)`, and Medley will not ask you about changes.

If you are running Medley under another window system (e.g. X-Windows), you can also terminate your session by using the “quit” or “kill” command usually provided as a menu option for the window containing the GWB display. This is equivalent to a “fast” exit–-there will be no checking for unsaved changes.

Chapter II

Grammar Testing and Debugging

You have now learned how to enter a grammar into the system and how to apply it in the analysis of strings typed into the Sentence Input window. It is likely for a simple grammar that all the rules are correct and that they are compatible with the lexical entries and morphological rules you have specified. But that is less likely to be true as your grammars become more complex and cover a larger fragment of a natural language. GWB's grammar-debugging facilities can help you explore the linguistic consequences of your grammatical formulations so that you can detect and correct both conceptual errors and errors of specification. You will be better able to see how to do this by using a grammar slightly less limited than `TOY ENGLISH`. The file `DEMOENGLISH`, which is supplied along with the GWB software, contains a grammar more suitable for experimenting with the system's testing and debugging facilities. This grammar contains rules, lexical entries. a morphology table, and some templates. It also contains several configurations, including a `DEMO ENGLISH` configuration that collects all these specifications together into a single active analysis environment.

You can load the grammar in this file into GWB's internal database by using the File Manager `LOAD` command. As described in Section 6.3 of Chapter I, click in the LFG Logo window to bring up the LFG command menu, and slide the mouse to the right of the `File Manager` item. Release the button over the `LOAD` command and when the File Manager window appears, type `{DSK}<LFG>ENGLISH>DEMOENGLISH` (or some other name if `DEMOENGLISH` resides elsewhere in your file system) to indicate the name of the file to load into the internal database. Confirm with a carriage return to start the loading process. When the file is loaded, the system will print `Done` in the File Manager window.

Once the file is loaded, you must activate the `DEMO ENGLISH` configuration before you can experiment with the new grammar. Left-click in the title-bar of the Sentence Input window, select the `Change Config` menu item, and then click on `DEMO ENGLISH` in the list of choices that are displayed to you. The new configuration will replace the `TOY ENGLISH` configuration that was previously active, and the title of the Sentence Input window changes accordingly. The strings you then type in the Sentence Input window will be analyzed with respect to the new grammar.

Request a parse by typing the following string into the Sentence Input window:

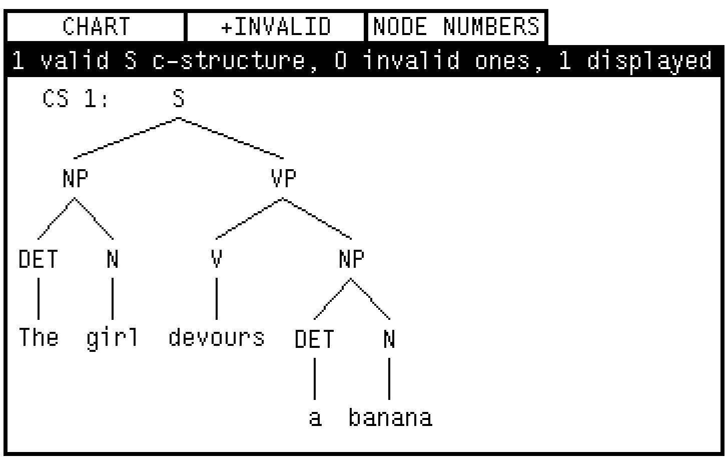
```
The girl devours a banana  [CTRL-X]
```

At this point the parser uses the rules, lexicon, and morphology of the `DEMO ENGLISH` configuration to analyze the sentence. Before it begins the analysis, it first modifies the input window by prefixing the characters `S:` at the front of the sentence. This marks the fact that the top-level parsing category for this analysis defaults to the category S, which is defined in the DEMO ENGLISH configuration as the root category of this grammar. The parsing category determines which rules to apply in a particular analysis (only those necessary to build constituents of that type) and what kind of structures to show in the c-structure window. You will see below how to override the default root category when you temporarily want to focus on the rules for other types of constituents.

1. Display of linguistic representations

The analysis of the sentence is completed in a few seconds, and summary information indicating the number of f-structure solutions, the computing time, and the size of the chart in tasks is printed in the prompt region above the Sentence Input window and, for future reference, is appended to the sentence inside the window. Grammatical structures discovered for the sentence appear in the C-structure and F-structure windows.

The title bar of the C-Structure window tells you how many c-structures the current grammar yields and the number of c-structures that are displayed. For this simple sentence there is one c-structure whose f-structure contraints have at least one valid solution, and no c-structures with only invalid solutions. The single valid c-structure is displayed in the window:



There is also only a single f-structure for this c-structure, shown in the F-structure window:

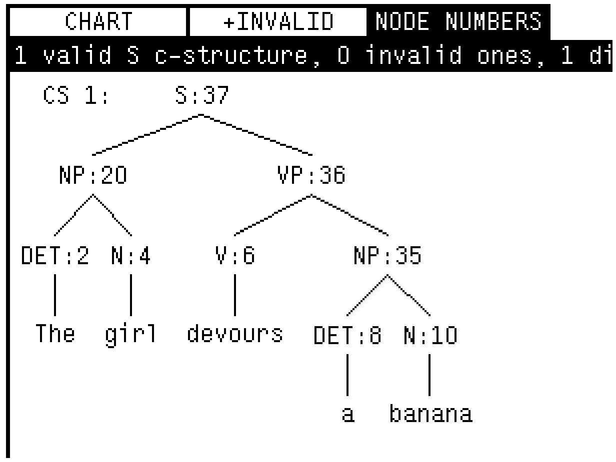
```
        INCONSISTENT  INCOMPLETE  INCOHERENT  EXPANDED
F-structures for S 37 in CS 1:  1 displayed
1 solution:  1 consistent, 1 complete, 1 coherent

F-structure 1--
   [PRED   'DEVOUR<[20:GIRL], [35:BANANA]>'                 ]
   [TENSE  PRESENT                                          ]
   [       20[PRED 'GIRL'                                 ] ]
   [SUBJ    4[CASE NOM, DEF +, NUM SG, PERSON 3, SPEC THE ] ]
   [        2[                                            ] ]
 37[       35[PRED 'BANANA'                     ]           ]
 36[OBJ    10[CASE ACC, NUM SG, PERSON 3, SPEC A]           ]
  6[        8[                                  ]           ]
```

The title-bar of the F-structure window also indicates the number of solutions displayed in the window, and the first line gives some summary statistics indicating the number of solutions that satisfy LFG's consistency, completeness, and coherence conditions. For this example, the single solution meets all the grammaticality conditions and is displayed immediately below the summary line.

The f-structure is displayed as an attribute-value matrix in standard LFG format, with the brackets annotated by identifying indices. The arguments in the semantic-form value of the `PRED` attribute are abbreviated representations for the full f-structures with the same index number shown below. Thus, `[20:GIRL]` shows that the first argument of `DEVOUR` is f-structure 20, which is also the value of the `SUBJ` attribute. `GIRL`, the relation in the semantic-form `PRED` of f-structure 20, is included in the abbreviation to make it easier to interpret. In this display the symbol-valued features are shown on a single line separated by commas so that more information can be shown in a given amount of screen space; the parameter `SYMBOLSINLINE` can be changed, as described in Section III.6, to obtain the conventional arrangement in which each symbol-valued feature appears on its own line.

The numeric indices play the role of the variables f_1, f_2, etc. that are sometimes used to annotate f-structures in published presentations of LFG. These indices serve to correlate the units of the f-structure with the c-structure nodes they correspond to. Each c-structure node that the parser discovers is assigned a unique node-number, whether or not it forms part of a valid S c-structure. Initially, these numbers are not shown in the c-structure display, in order to avoid cluttering the tree. But clicking on the `NODE NUMBERS` menu item attached to the top of the c-structure window will cause the tree to be redrawn with visible node-numbers:



Comparing these node-numbers with the indices on the f-structure, it is now easy to see that the outermost f-structure corresponds not only to the root `S` node, but also to the `VP` and `V` nodes. This reflects the many-to-one nature of the c-structure to f-structure correspondence, and also formalizes the intuition that the `V` is head of the `VP` and the `VP` is head of the sentence. You can click again on the `NODE NUMBERS` item to turn off the node-number display.

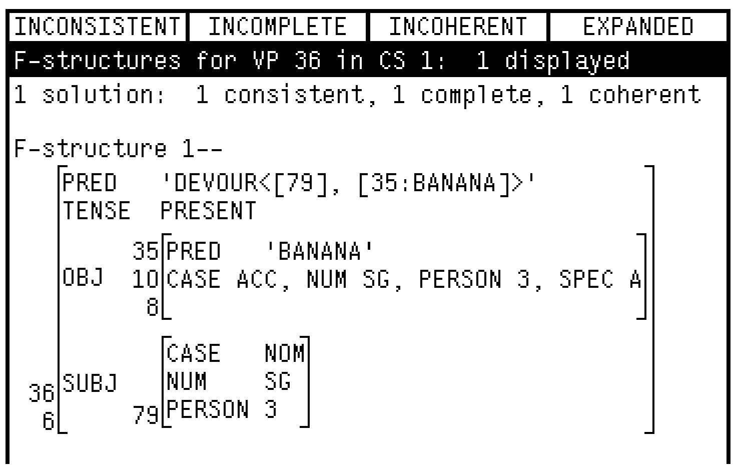
The displays in the C-Structure and F-Structure windows give access to additional information about the analysis. The nodes of the c-structure act as menu items. When you click the left button on the `S` node, for example, another window opens in the middle of the screen and shows the functional description associated with that node:

```
Description from S:37 in CS
(f37 SUBJ)=f20
(f20 CASE)=NOM
(f2 SPEC)=THE
(f2 DEF)=+
(f4 PERSON)=3
(f4 NUM)=SG
@(CN GIRL):
 (f4 PRED)='GIRL'
 { (f4 NUM)=SG
   (f4 SPEC)
  |(f4 NUM)=PL}
(f37 TENSE)
f37=f36
(f6 TENSE)=PRESENT
(f6 SUBJ CASE)=NOM
(f6 SUBJ NUM)=SG
(f6 SUBJ PERSON)=3
@(TRANS DEVOUR):
```

The functional description is the union of all the equations instantiated in the subtree that that node dominates. If all the equations are not visible in the window, you can reshape the window or use its left or bottom scroll-bars to move the equations around. The metavariables ↑ and ↓ from the rule and lexicon schemata are replaced by the appropriate f-structure variables (displayed as an `f` followed by a corresponding node-number). This display also indicates what elements of the f-descriptions are derived by expansion of functional templates: `@(CN GIRL)` shows that `GIRL` is a parameter to the `CN` (common-noun) template whose expansion results in the equations indented after the colon (see Chapter III, Section 4).

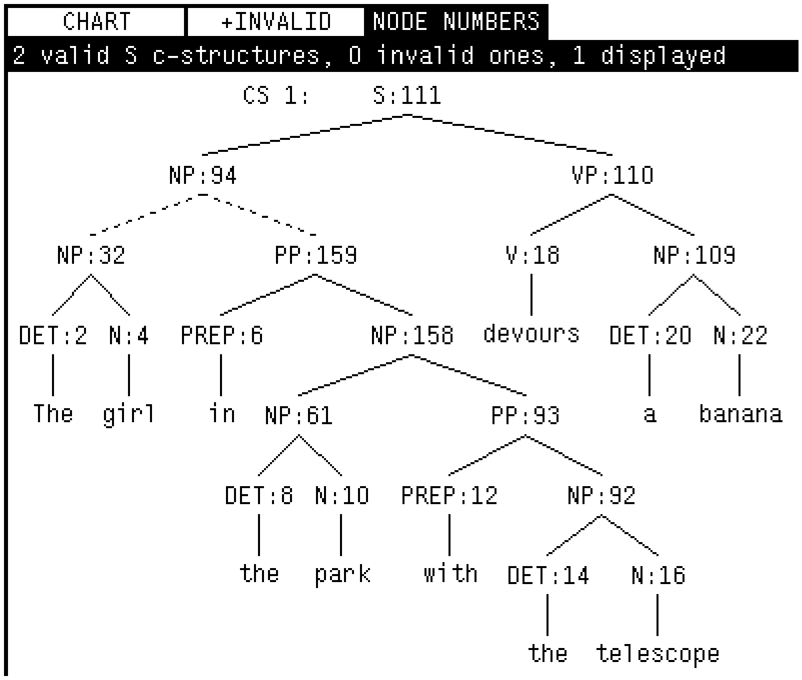
By selecting nonterminal nodes lower in the tree, you can examine information from smaller parts of the sentence. If you select a preterminal lexical category, for example, you can focus just on lexical template expansions. Selecting a terminal node (a word) does not show the instantiated lexical schemata, since instantiating them only trivially modifies the lexical information. Instead, clicking on a word brings the stem and affixes of that word into a lexicon window, thus indicating its morphological analysis and also permitting the relevant entries to be examined and perhaps edited. The currently active lexical entries for words in the sentence are thus immediately available, without requiring further specification of language and version. Similarly, the rules for the categories in the tree can also be accessed by holding the `CTRL` (`EDIT` or `PROPS` on some keyboards) key down when selecting the node. For both lexical entries and rules, GWB will try to reuse an existing editing window if you click with the left button; if you click with the middle button, it will always prompt for a new window.

Selecting a node with the left button (without `CTRL`) shows the functional description for a subtree. Clicking with the middle button has another useful effect: it causes the f-structure solutions for that node's description to be displayed in the F-structure window. If you middle-click the `VP` node, for example, you will see the f-structure determined by the schemata in the rules and lexical entries just for that subtree:

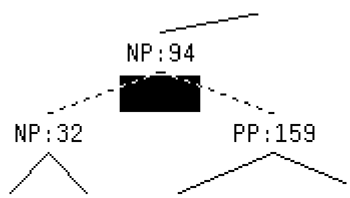


This resembles the f-structure for the dominating `S` node but some details of the `SUBJ` f-structure are missing. The `PERSON` and `NUM` are included, because they are supplied from the verbal inflection, but the `PRED` does not appear because it is not defined within the `VP` subtree. The solution is not considered incomplete, however, because it is still possible for the predicate to be filled higher up. If you middle-click on the root `S`, the original outermost f-structure will be displayed, with the `SUBJ` completed.

In general, there can be a large number of alternative trees over a given string, some of which are valid and some of which are invalid. In an attempt to make the linguistic representations easier to comprehend and navigate, GWB does not display automatically the complete collection of alternative trees. Instead, it chooses from among the set of valid trees, if any, one particular tree to show as the initial c-structure display, and that tree's f-structure solutions are shown in the f-structure window. Nodes at which there are alternative subtree expansions are indicated by dashed lines leading to their daughters. You can use the mouse to indicate that you want to examine those alternatives as well. Enter the sentence `The girl in the park with the telescope devours a banana.` This sentence has two different c-structures, depending on how the prepositional phrases are attached. After typing `CTRL-X`, you will the following c-structure display:

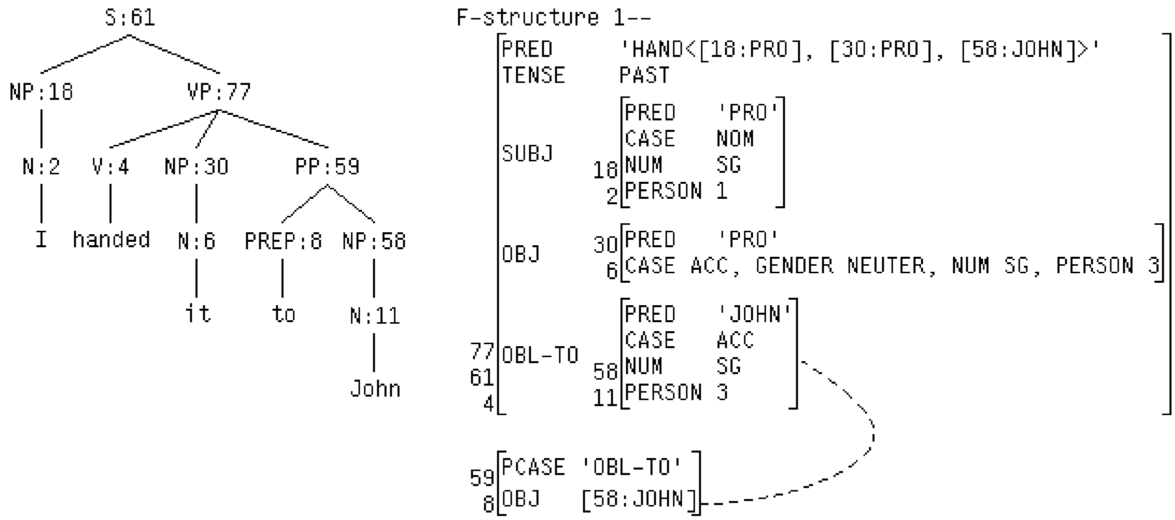


The title reports that there are two valid c-structures with only one currently displayed. The dashed lines indicate the location of the alternative subtree expansion. If you want to examine the other tree, you can click the left or right buttons in the region of the dashed lines, between the `NP:94` node and its line of daughters. A black rectangle will appear; releasing the button over that rectangle will cause the alternative tree to be added to the c-structure window.



You may need to scroll the window horizontally, using the bottom scroll-bar, to see all of the alternative tree. You can click on its nodes with left or middle to see its associated f-structure displays. If a sentence has both valid and invalid trees (this one does not), clicking `+INVALID` at the top of the window may cause some lines from a node to be converted from solid to dashed. This signals that only invalid trees have alternative subtrees concealed at that node and only those will be displayed by clicking and releasing the mouse in the black rectangle.

The solutions displayed in the F-structure window represent not only the f-structures themselves but also, by virtue of the node numbers, the c-structure to f-structure mapping ϕ for all the nodes in the subtree whose root you have selected. Typically that mapping is presented in its entirety by the numbers annotated to a single f-structure, but that is not the case when the f-structures corresponding to some nodes are not connected to the f-structure of the root. In that case, the single root f-structure will be followed in the display by the disconnected f-structures corresponding to other nodes. You can see this display by analyzing the sentence `I handed it to John`. You will see the following c-structure and f-structure displays:



The `PP` category in the `VP` rule has the annotation `(↑ (↓ PCASE))=(↓ OBJ)` for the oblique argument. This makes the `OBJ` of the `PP` f-structure serve as the `OBL-TO` of the `S` f-structure, but asserts no connection for the f-structure corresponding to the `PP` node itself. This f-structure is therefore shown as a separate item in the solution display, with the appropriate identity link for its object.

The solutions in the F-structure window also act as menu items. If you select a solution with the left button, its particular f-description is shown in the description window. In general, the description for a specific solution is derived from but is not the same as the description associated with the node it corresponds to. The difference shows up when the node's f-description (obtained by left-clicking the node) contains disjunctions of equations, for example, the active/passive alternatives marked by the vertical bars you can see by scrolling the f-description of the `S` node (left-click on the node). Conceptually, the solutions at a node are found by multiplying the different branches of its disjunctive f-description into an equivalent disjunctive normal form. This consists of a single outer disjunction whose alternatives are sets of simply conjoined equations. Potentially, a separate solution will appear in the f-structure window for each of these alternative conjunction-sets. The description you see when you left-click on a particular solution is the set of conjoined equations that that solution satisfies, ignoring the alternative equations that are displayed when the node is selected but which correspond to other solutions. Thus, if you left-click on the single f-structure for the `S` node, only the equations for the active interpretation will appear in the description window:

```
Description of F-structure 1
(f61 SUBJ)=f18
(f18 CASE)=NOM
(f2 PRED)='PRO'
(f2 NUM)=SG
(f2 PERSON)=1
(f2 CASE)=NOM
(f61 TENSE)
f61=f77
(f4 TENSE)=PAST
(f4 SUBJ CASE)=NOM
~(f4 PARTICIPLE)
(f4 PRED)='HAND<(f4 SUBJ)(f4 OBJ)(f4 OBL-TO)>
(f77 OBJ)=f30
(f30 CASE)=ACC
(f6 PRED)='PRO'
(f6 NUM)=SG
(f6 PERSON)=3
(f6 GENDER)=NEUTER
(f77 f59)=(f59 OBJ)
(f8 PCASE)='OBL-TO'
(f59 OBJ)=f58
(f58 CASE)=ACC
```

GWB uses special techniques (see, for example, Maxwell and Kaplan, 1993) to identify the nodes in a tree at which particular conjunction-sets in the disjunctive normal form become unsatisfiable, and it tries to confine the presentation of unsatisfactory solutions just to the nodes where they first go bad. These strategies also help in making the linguistic representations more comprehensible, but sometimes the results are surprising. To illustrate one effect, enter the sentence `A sheep devoured a banana`. This produces a single c-structure with a single f-structure solution. If you click on `sheep` in the c-structure window, you will see a disjunctive specification (as opposed to simply an underspecification) for the `NUM` attribute:

```
sheep   N   IRR   @(CN SHEEP)
                  { (↑ NUM)=SG
                   |(↑ NUM)=PL}
                  (↑ PERSON)=3.
```

And if you middle-click on the N dominating `sheep`, you will see the disjunction reflected in two f-structure solutions:

```
F-structure 1--             F-structure 2--
  [PRED   'SHEEP']            [PRED   'SHEEP']
  [NUM    PL     ]            [NUM    SG     ]
 4[PERSON 3      ]           4[PERSON 3      ]
```

However, when you click on the higher NP node to see the solutions for the f-description at that node, only the `SG` solution appears. The plural specification on the noun is incompatible with the singular feature of the indefinite article. That inconsistency is detected as the article and noun are being combined, and solutions for all conjunction-sets containing that inconsistency are not presented at higher nodes. Thus, you may need to middle-click on several nodes in the lower part of the tree, not just the root, in order to examine the effects of all disjunctive possibilities.

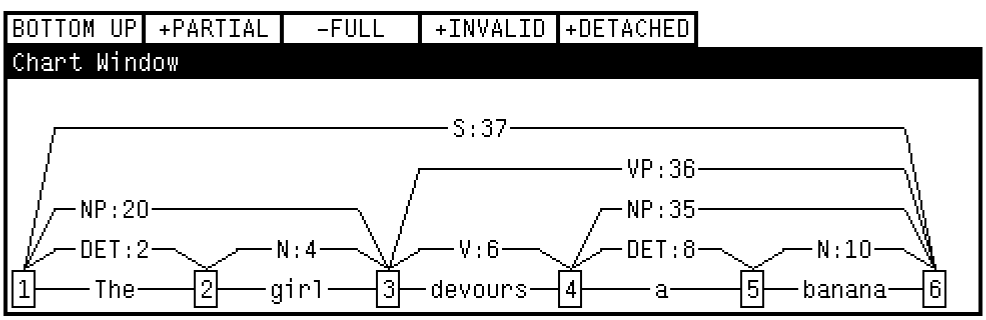
F-structure solutions also act as menu items for middle-button selection. If the grammar defines additional projection structures to be in correspondence with the f-structure, then the middle button can be used to inspect these, as discussed in Section 4 below. The `DEMO ENGLISH` grammar defines no further projections, so middle clicking on its f-structures has no effect.

2. Debugging with c-structure and chart

If you want to see c-structures for which no corresponding functional structure is valid, click the button `+INVALID` above the c-structure window. If the window is not large enough for all the trees to be visible at once, you can use the window-command menu (click the right button in the window's title) to enlarge the window or click the mouse in the horizontal or vertical scroll-bars to slide the sequence of trees across the window. If you want to compare two trees, you can either shift-select one of the trees into a TEdit window (this is also useful for writing papers), or you can use the background menu to make a snapshot of one of the trees (see Chapter IV, Section 13 for details).

Above the C-Structure Window, left-click `CHART` to see the various constituents (both complete and partial) that are compatible with the grammar, whether or not they are successfully included in a complete S spanning the string. The parser looks up the words in the lexicon and assigns to each word the names of the categories it belongs to. Then it finds all possible constituents that could be constructed out of each string of categories given the currently active rules. Many of these will be ruled out at the c-structure level because they are not part of any root constituent spanning the whole string. Constituents that are ruled out because they are not part of any root can be viewed by clicking `+DETACHED` on the menu at the top of the chart window.

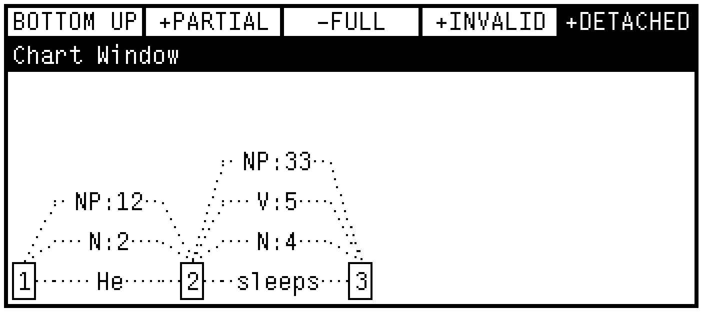
The chart is a parse forest, so each constituent displayed in the chart represents all of the different subtree structures that can be used to construct that constituent. When you left-click a node in the chart, all of its subtrees with valid functional structures are displayed in the c-structure window.



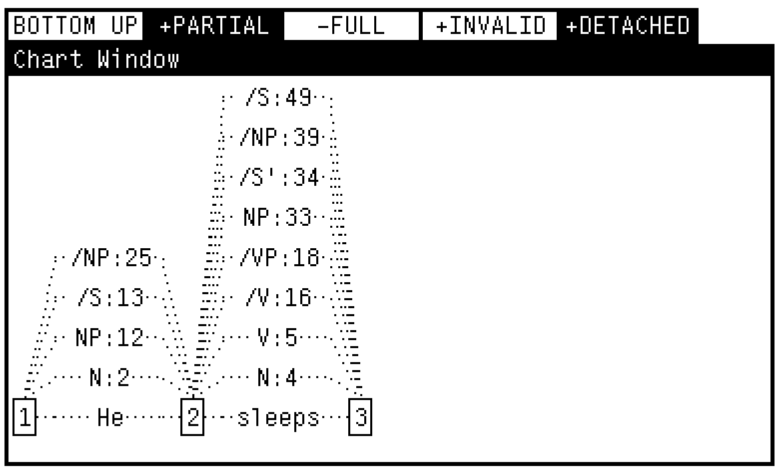
Now type into the Sentence Input Window

```
He sleeps [CTRL-X]
```

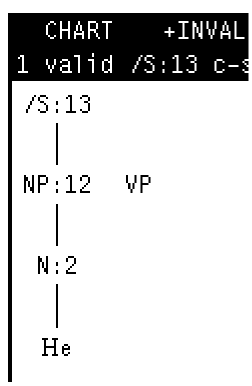
The title bar of the C-Structure Window says "No S c-structures", indicating that something is wrong at the c-structure level, that there is no way of constructing a root S that completely spans the string. Click the `CHART` button if it is not already black.



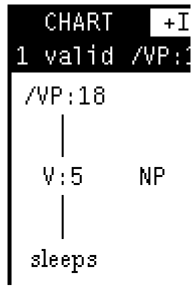
Notice first that in this chart all the lines are dotted. Dotted lines are used in chart displays to represent detached constituents, those that are not components of any spanning root. Usually, detached constituents are not shown unless you specifically request them by selecting `+DETACHED` in the chart window menu. But GWB automatically selects this option when there are no spanning roots; otherwise, the chart display would initially be empty and uninformative. Next notice that there are no S or VP constituents—the words in this string did not completely satisfy the requirements of either the S or VP rules. Click the `+PARTIAL` menu item at the top of the chart to see the partial constituents:



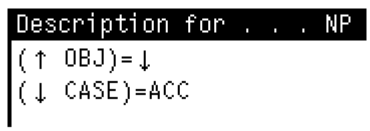
The partial constituents, with labels prefixed by "/", represent partial matches of the c-structure patterns for the various rules. The `/S:13` constituent indicates that the word `He` between string positions 1 and 2 provides evidence to satisfy the beginning of a path through the S rule. If you left-click on the `/S:13`, the C-Structure Window will show the subtrees that enable that partial match:



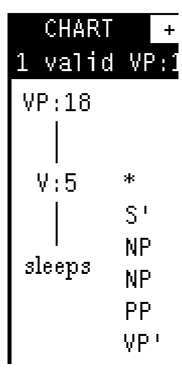
Here the `/S:13` dominates a complete NP but needs a fully satisfactory VP immediately to the right of the NP to extend the match; such a VP does not exist. By middle-clicking on the `/S:13` node in the C-Structure Window you will see that the NP corresponds to the `SUBJ` of the partial S's f-structure. Even though no complete (unslashed) VP is present, the `/VP:18` constituent indicates that a VP was started over the word `sleeps`. If you click on the `/VP:18` constituent, you will see a tree for `/VP:18` plus the categories required to extend it further:



The NP indicates that the `/VP:18` constituent requires an NP to its right. If you left-click on that NP symbol, the F-Description Window will show the schemata from the VP grammar rule that introduces that particular NP:



From this we can see that the required NP contributes an object to the sentence, and thus, that our grammar erroneously allows only for transitive verbs. If you look at the VP rule in the Rule Window (click on the `/VP` node with the `CTRL` key down), you will see that this is the case: the grammar requires that every VP have an NP object. Add parentheses around the NP and its constraints, type `CTRL-X` in the Rule Window to install the change, and try parsing the sentence again (simply type another `CTRL-X`, since the type-in cursor automatically moved back to the Sentence Input Window). Now you will see a valid c-structure with the proper f-structure. A complete spanning S now appears in the chart, and a slash no longer marks the `VP:18` as partial. Indeed, if you left-click on `VP:18`, you will see that it now has many more extension possibilities:

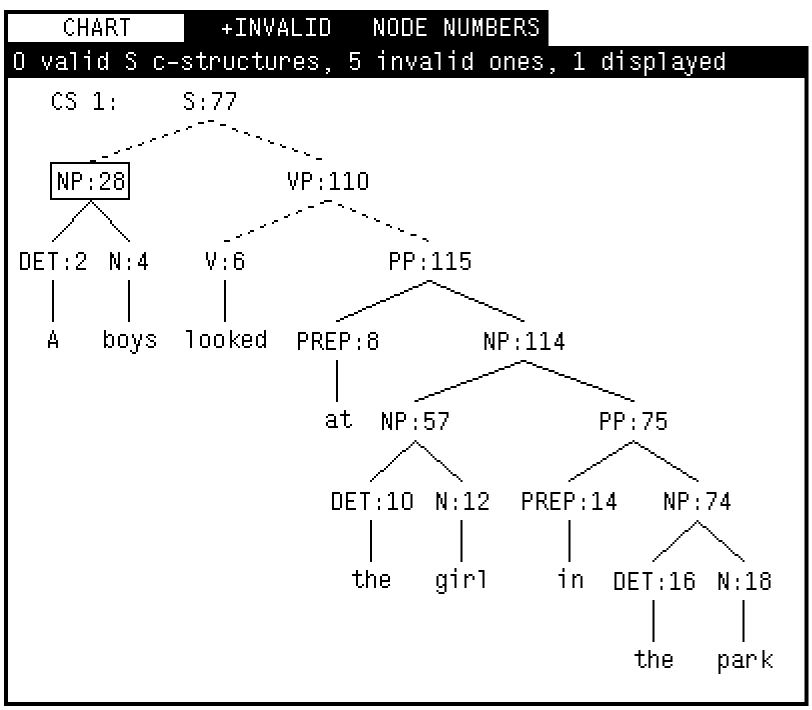


With the object NP no longer obligatory, the other constituents to the right of the object are now permitted. There are two NP's in the column of next categories; by left-clicking you can see that they differ in their functional annotations. Also, the * at the top of the column marks the fact that the VP can be considered complete as it now stands: the rule permits but does not require the incorporation of other nodes.

The menu above the Chart Window offers several other selections to control exactly which constituents are displayed. The usual mode of operation is to display only complete or full constituents that have validly satisfied f-descriptions and which are attached to the subtree of some spanning root node—when at least one such constituent exists. If there is no spanning root, then the full valid detached constituents are shown. You can ask for partial constituents to be included, as you have seen, by selecting `+PARTIAL` and constituents without valid f-structures by selecting `+INVALID`. If the chart becomes too cluttered and you want to focus on only partial constituents, you can cause the full constituents to be suppressed by selecting `−FULL` (the label is `−FULL` here instead of `−COMPLETE` to avoid confusion with the functional notion of completeness). Finally, you can select `BOTTOM UP` to cause GWB to display constituents that would be recognized by a bottom-up parsing algorithm, that is, that satisfy the requirements of some c-structure rule even though they are not discovered in the usual search for constituents that can be part of a root category starting at the left-end of the string. This permits you to examine constituents towards the right end of the string that may otherwise be concealed by unsatisfactory sequences further to the left.

3. F-structure debugging

Well-formed constituent structures are filtered through the functional well-formedness conditions. By default, the system automatically displays one c-structure tree for which the functional description of the root node is consistent and is satisfied by at least one complete and coherent f-structure. If none of the c-structures are valid in this way, then the `+INVALID` button at the top of the c-structure is automatically selected, and one of the invalid trees is chosen for display. Thus, if you analyze the string `A boys looked at the girl in the park`, you will see the following c-structure window:

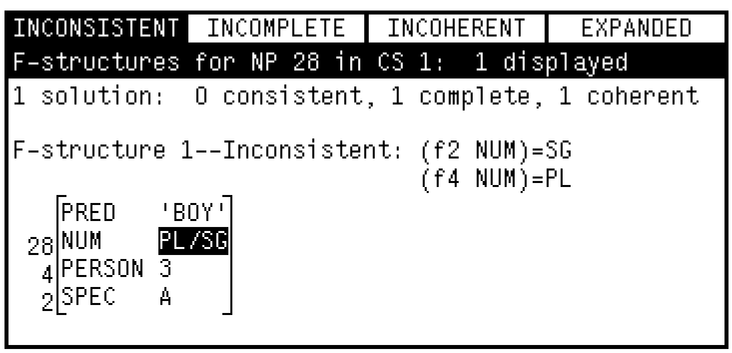


The box around the NP node indicates that it is the lowest node in the tree where the f-description becomes unsatisfactory. That is, the DET and N subtrees both have satisfiable f-descriptions, but when the f-descriptions of those two nodes are combined together according to the assertions in the NP rule, the resulting formula no longer has well-formed solutions. This being the case, GWB does not bother to compute or solve the f-description for the higher S node, and the f-structure window is left completely empty to signal that nothing has been computed. As before, the dashed lines in the tree indicate that there is at least one other way (also invalid) of constructing a VP over that interval of the string, and you can click under under the VP to ask that the alternatives there be displayed.

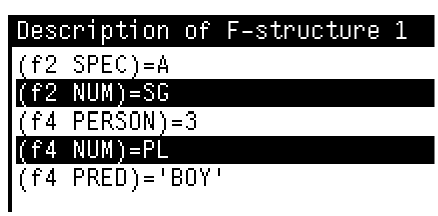
The boxed nodes in the tree along with the empty f-structure window may be enough information to confirm that your grammar correctly classifies unacceptable strings as ungrammatical. In some cases, however, it may be important to examine the analysis results in more detail to make sure that the bad strings are bad for the right reasons. A detailed inspection of invalid results may also be useful in locating errors in your grammar or lexicon that mark as ungrammatical a string that you expect to be accepted. Although it usually does not display them, GWB allows you to make specific requests to see structures that are invalid for different reasons.

3.1. Inconsistency

For example, suppose you wanted to see more of the details of the boxed NP's f-structure analysis. If you click on that NP with the middle button, the F-structure Window changes to indicate that there is one solution to the f-description at that node (that is, the disjunctive normal form of the f-description has only one conjunctive subformula) but that it is inconsistent. Above the f-structure window, left-click INCONSISTENT to see the f-structure. The button reverses and the faulty structure is displayed with the inconsistency highlighted white-on-black and the mutually unsatisfiable constraints printed at the top.

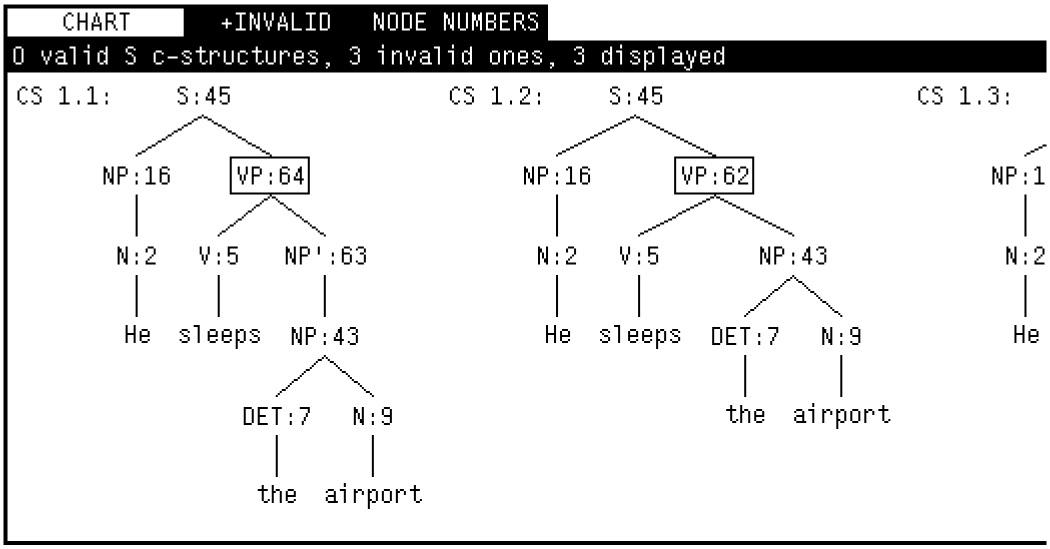


Left-click anywhere inside that f-structure to see the complete f-description, now with any inconsistent equations highlighted.

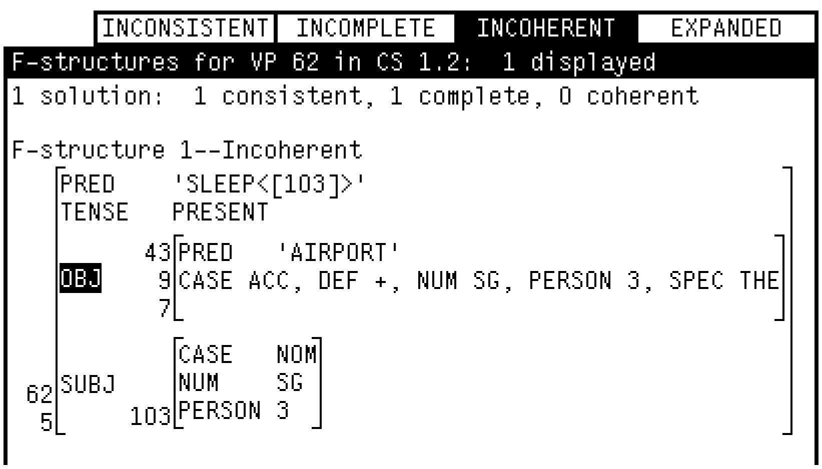


3.2. Incoherence

If you enter `He sleeps the airport [CTRL-X]`, you will see only an invalid tree in the C-structure Window with dashes marking that the top node has two alternative subtrees. If you click under the `S`, those trees will be shown:



Again, the boxes around the `VP`'s indicate indicate where the solutions have gone bad: The functional requirements for all nodes below the `VP`'s are satisfactory, but there was no way of combining the solutions for their daughters to make valid solutions for the VP's. Middle-click on `VP:62` to display its f-structures. The f-structure window indicates that there is one solution, but it is incoherent. Click on the `INCOHERENT` button in the f-structure window to see the incoherent solution. Notice that `OBJ2` has been highlighted to show that it is the ungoverned attribute. You can middle-click on `VP:64` also to find out what is wrong with its f-structure.

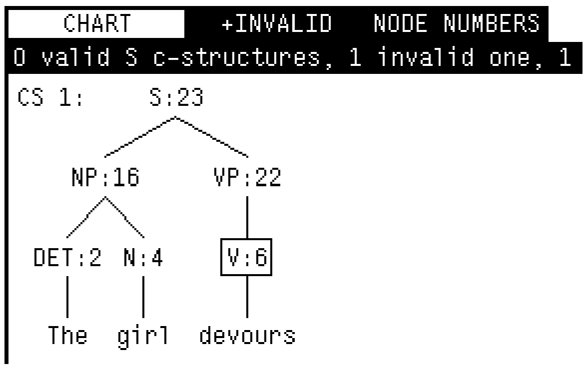


Middle-clicking on the `S` nodes provides no interesting functional information. Those are above the boxed `VP`'s, and these are already known to be bad in irredeemable ways. GWB does not bother to propagate the incorrect results into higher structures, to avoid excessive computation, although it is sometimes that an expected structure is not displayed.

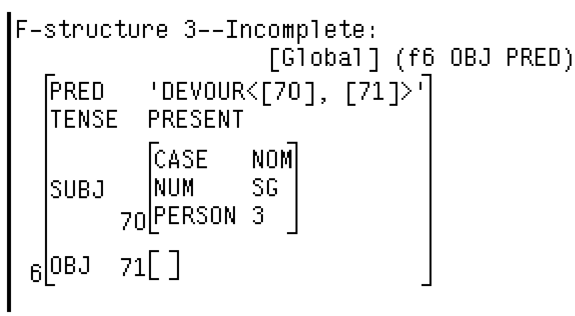
The Coherence Condition as enforced by GWB is slightly different than the condition defined originally by Kaplan and Bresnan (1982). According to the original definition, an f-structure is regarded as incoherent if a governable grammatical function is not sanctioned by a local predicate. Experience has suggested a refinement to this definition, and GWB marks a governable function as incoherent if *a local predicate exists and* it does not sanction the function. For example, this allows an f-structure corresponding to a preposition-phrase to have an `OBJ` even though the preposition itself serves as a semantically vacuous case-marker.

3.3. Incompleteness and determinacy

Now type `The girl devours [CTRL-X]` into the Sentence Input Window.



This sentence has no solution because it is incomplete, i.e. it is missing an object. Usually incompleteness is detected at the top-level `S`, when it is known that there is no more information to be had. However, in this case the sentence is marked bad at the `V:6` constituent. GWB conducted a global analysis and determined that the attribute `OBJ` is not assigned by any of the schemata instantiated from the grammar rules or lexical entries for this tree, implying that the verb's subcategorization frame can never be satisfied, and that it is therefore pointless to solve any equations at the `VP` level or above. This global analysis permits better localization of the source of the difficulty—it enables the box to be drawn at the `V` instead of the `S`. It also allows GWB to prune incompletenesses early, producing a substantial improvement in performance. If we click on `V:6` and then select the `INCOMPLETE` menu item, we can see the bad solutions. In this case there are three, but we will only look at the third one:



The solution is marked incomplete on the first line. The second line gives more detail, showing that the `PRED` of the `OBJ`, required by `DEVOUR`, is missing. The `[Global]` tag indicates that this was determined in a global analysis. The `SUBJ PRED` is also missing at this level but is not noted as a source of incompleteness. The global analysis recognized that it might be supplied from information outside of the V subtree, just as for the VP f-structure we examined in Section 1.

The f-structure ontology as proposed by Kaplan and Bresnan (1982) did not include a "bottom" element to represent the complete absence of information about the value of an attribute. F-descriptions containing statements of the form (↑ SUBJ)=(↑ OBJ) would give rise to unacceptable solutions if there were no other specification for either the SUBJ or the OBJ. The f-description would be indeterminate, since there was no single smallest element to serve as the common value. However, GWB does include a bottom element in the f-structure subsumption lattice, and that element provides for unique minimal solutions for f-descriptions of this sort. If an attribute has bottom as its value, then neither the attribute nor the value is shown in the f-structure display, just as if no assertion about the attribute had been made. This arrangement is particularly convenient when relating elements of different projections: a general statement can be made to map f-structure adjuncts to semantic-structure modifiers, for example, and the f-description will be acceptable even when the particular f-structure has no adjunct.

3.4. Resolution of functional uncertainties

Expressions of functional uncertainty can be used in GWB to encode constraints on various sorts of nonlocal f-structure dependencies. The following NP rule, for example, uses this mechanism to specify the possible within-clause function of a relative clause head:

```
NP → { (DET) AP*:↓∈(↑ ADJ); N
      |NP  S':↑=(↓ TOPIC POSS*)
              (↓ TOPIC)=(↓ {XCOMP|COMP}* {SUBJ|OBJ})
              ↓∈(↑ RELMOD)}.
```

The schema ↑=(↓ TOPIC POSS*) identifies the complete NP f-structure with the bottom of a (possibly empty) chain of possessives inside the TOPIC, and (↓ TOPIC) = (↓ {XCOMP|COMP}* {SUBJ|OBJ}) then indicates that the entire TOPIC may serve as a SUBJ or OBJ possibly in some internal complement structure. Looking at an f-structure with many attributes and values at many different levels of embedding, it is not always obvious that a given uncertainty has been resolved in the way you intend. The EXPANDED item in the menu above the F-structure Window changes the display of the f-structure to show you how uncertainties have been instantiated. Type the following into the Sentence Input Window:

```
The girl who John telephoned slept [CTRL-X]
```

The c-structure and f-structure will appear, and you will see that GIRL shows up not only as the subject of SLEEP but also as the object of TELEPHONE. This is the result we expect, since we believe that OBJ is the only string in the language {XCOMP|COMP}* {SUBJ|OBJ} that permits all other functional requirements to be satisfied. We can confirm that this is what happened by clicking on EXPANDED. The f-structure is redisplayed with the important difference that it now shows how functional uncertainties were expanded:

```
F-structure 1--
   [PRED   'SLEEP<[52:GIRL]>'                                                  ]
   |TENSE  PAST                                                                |
   |       [PRED      'GIRL'                                                  ]|
   |       |         { [                                                    ] }|
   |       |         { |           [PRED  'RELPRO']                         | }|
   |       |         { |HEAD     35|BOUND +       |                         | }|
   |       |         { |          6[WH    +       ]                         | }|
   |       |         { |                                                    | }|
   |       |         { |TOPIC    [35:RELPRO]                                | }|
   |       |         { |PRED     'TELEPHONE<[43:JOHN], [35:RELPRO]>'        | }|
   |SUBJ   |RELMOD   { |TENSE    PAST                                       | }|
   |       |         { |OBJ      [35:RELPRO]                                | }|
   |       |         { |         [PRED   'JOHN']                            | }|
   |       |         {51|        |CASE   NOM   |                            | }|
   |       |         {50|SUBJ    |NUM    SG    |                            | }|
   |     52|         {49|      43|PERSON 3     |                            | }|
 59|     22|         {10|       8[             ]                            ] }|
 58|      4|      161                                                         |
 12[      2[CASE NOM, DEF +, NUM SG, PERSON 3, SPEC THE                       ]]
```

In an expanded display attributes such as `OBJ` are highlighted to indicate that they lie on the path of a functional uncertainty.

4. Projections

The Grammar Writer's Workbench allows the grammar writer to define projections other than the ϕ (c-structure to f-structure) correspondence. New projections are usually used to describe major sub-systems of linguistic theory. For instance, there could be a σ projection to handle the semantics, a ρ projection to encode selectional restrictions, and a δ projection for a discourse structure. Other uses for projections have been proposed, such as a τ projection that defines the translation of an f-structure into some other language (Kaplan et al., 1989; Kaplan and Wedekind, 1993).

To see an example of how projections can be used, go to the Sentence Input Window and click in the title bar with the left mouse button to get its menu. Select the `Change Config` menu item and then select `SIGDEMO ENGLISH` from the menu that pops up. This will install a new configuration, activating new rules and templates. The title bar of the Sentence Input Window reflects the change, indicating that the default root category is different for this configuration and also that the σ (semantic) projection is defined as well as the usual c-structure-to-f-structure projection:

`SIGDEMO ENGLISH:  ROOT input window with σ φ projections`

You can examine this configuration using the `Edit Config` menu item:

```
Configuration Window
SIGDEMO ENGLISH
 RULES (DEMO ENGLISH)
         (SIGDEMO ENGLISH)
         (XSIGDEMO ENGLISH).
 ROOTCAT ROOT.
 LEXENTRIES (DEMO ENGLISH)
              (DMORPH ENGLISH)
              (XSIGDEMO ENGLISH).
 TEMPLATES (DEMO ENGLISH)
             (SIGDEMO ENGLISH)
             (SELECTION ENGLISH)
             (XSIGDEMO ENGLISH).
 MORPHTABLES (DEMO ENGLISH).
 GOVERNABLERELATIONS  SUBJ OBJ OBJ2 COMP ?COMP OBL-?* POSS.
 SEMANTICFUNCTIONS  ADJ RELMOD TOPIC FOCUS.
 NONDISTRIBUTIVE  ADJ XADJ CONJ.
 PROJECTIONS (φ F- (HEAD TOPIC))
               (σ Semantic (REL TIME ARG1 ARG2 ARG3)).
 EPSILON e.
 PARAMETERS .
 INPUTKEYBOARD DEFAULT.
 LEXICALFONT DEFAULTFONT.
 TESTFILE .
----
```

This shares its lexical entries with the DEMO ENGLISH configuration, and it also inherits its rules and templates from the DEMO ENGLISH versions. These are overriden, however, by any specifications belonging to SIGDEMO ENGLISH and SELECTION ENGLISH. SIGDEMO ENGLISH contains a VP rule corrected to allow the object to be optional, and an NP rule that establishes a semantic relation between the NP head and the relative pronoun. The major differences are to be found in the SIGDEMO ENGLISH templates. They override almost all the original ones to indicate how the σ projection maps f-structures associated with preterminal nodes into their semantic structures. This configuration also gives highest priority to the XSIGDEMO ENGLISH versions of rules, lexeical entries, and templates. No such items exist at the start, so these are vacuous specifications. But including them in the configuration provides a convenient home for experimental versions of various items that can be tested and evaluated without confusing them with the more stable versions in SIGDEMO. Finally, this configuration introduces a new root category ROOT that permits punctuation marks optionally to follow the top-level S.

Now analyze the sentence The girl devours a banana [CTRL-X]. The c-structure window and the f-structure window have essentially the same information as before. However, if you now click in the f-structure window with the middle button, a new window will open with the semantic structure for the sentence:

```
Semantic structures for F-structure 1:  1 displayed
1 solution:  1 consistent, 1 complete, 1 coherent

Semantic structure 1--
  [REL     DEVOUR        ]
  [ARG1   4[REL GIRL]    ]
 6[ARG2  10[REL BANANA]  ]
```

The semantic structure window is very similar to the f-structure window. Clicking in it with the left button produces the semantic structure's description, just as with f-structures.

```
Description of Semantic structure 1
(σf4 REL)=GIRL
(σf6 REL)=DEVOUR
(σf6 ARG1)=σ(f6 SUBJ)
(σf6 ARG2)=σ(f6 OBJ)
(σf10 REL)=BANANA
```

If there are projections defined on semantic structures, then middle-clicking in the semantic structure window will let you access them, just like in the f-structure window.

If there is more than one projection defined for f-structures, then clicking on an f-structure with the middle button will produce a menu that lets you choose which of its projections you want to view. The numerical indexes in this display identify the f-structure elements that the σ projection maps to the associated semantic structure units.

Currently, one thing makes debugging with other projections tricky, however: An f-structure solution will sometimes be marked incomplete or inconsistent when there is nothing wrong with the f-structure itself. Instead, one of its projections is incomplete or inconsistent, and the Grammar Writer's Workbench does not distinguish between the different projections when determining that a solution is bad.

5. Changing default analysis modes

We usually think in terms of parsing sentences, and S is thus commonly specified as the `ROOT` component of the configuration. Even though S is the root category for the grammar as a whole, you may wish to focus on some other category, say NP, while you are developing and debugging its rules and lexical entries. This means you can test the NP subgrammar without typing in complete sentences, and GWB will not spend as much time exploring grammatical possibilities that are irrelevant to your current concerns. Also, GWB will automatically display in the C-Structure Window any trees spanning the string that are labeled with the current parse category.

You can use any category in the active grammar as the current parsing category by selecting the `Parse Cat` item in either the left-button pop-up menu for the Sentence Input Window or the left or middle menu in the LFG Logo Window. This will give you a menu of categories that can be designated as the current parsing category. If you make a change, the title of the LFG Input Window will be modified to show the new parse category. From now on, any new strings will have the new parse category prepended to them. (Sentences that already have a parse category prepended will continue to parse according to that parse category.) It is also possible to change the parse category for a single input by explicitly typing the desired category in the input window. Thus, if NP is the current parse category, you can tell GWB to parse a particular string as an S by typing `S:` at the front:

```
S: The girl devours the banana [CTRL-X]
```

will parse as an S no matter what the current parse category is set to.

Chapter III

Grammatical Notations

The examples in Chapter I illustrate how the Grammar Writer's Workbench faithfully implements the original LFG formalism as presented by Kaplan and Bresnan (1982). The right-hand sides of c-structure rules denote arbitrary regular languages by means of expressions using the basic operations of concatenation, disjunction or union, and Kleene closure. The primitive terms of these expressions are either simple categories or categories annotated with Boolean combinations of functional schemata. The individual schemata involve the equality, set-membership, and existential predicates, applying them to designators constructed from the metavariables $\uparrow$ and $\downarrow$ and simple function-applications. GWB also implements a number of extensions to the original LFG formalism. It provides a notation much richer than standard regular expressions for specifying the regular languages of c-structure rule expansions. Functional specifications can make use of new formal devices such as functional uncertainty (Kaplan and Maxwell, 1988a; Kaplan and Zaenen, 1989b), functional precedence (Kaplan and Zaenen, 1989a), and multiple projections that relate f-structure to other levels of linguistic representation (Kaplan, 1987; Halvorsen and Kaplan, 1988). There are also a number of abbreviatory devices to help in stating both c- and f-structure generalizations. This chapter supplements the tutorial material in Chapter I by providing a technical description of the full set of notational conventions that are supported by the Grammar Writer's Workbench.

1. Regular predicates for c-structure rules

The daughter string of a c-structure node in LFG must belong to the regular language specified in the right-hand side of the relevant c-structure rule. Since every regular language can be specified by a regular expression applying only the operations of concatenation, union, and Kleene closure to primitive terms, a simple regular expression notation would be sufficient to characterize all and only the node expansions that are permitted by lexical-functional theory. But the rich mathematical properties of the regular languages offer many other possibilities for expressing linguistic generalizations without any increase in formal power or change to the set of acceptable trees. Many of these expressive possibilities are made available in GWB's c-structure rule notation. GWB allows c-structure expansions to be specified by means of Boolean combinations of "regular predicates" over primitive terms. Since the regular languages are closed under the operations of union, intersection, and complementation, Boolean combinations will be regular if each of the predicates itself only denotes a regular language. Despite their overall expressiveness, the predicates described below all have this characteristic.

An LFG c-structure rule in GWB is a specification of the form

$M \rightarrow p.$

where M is the mother category whose expansion is defined by this rule and p is a regular predicate that determines the (regular) set of possible daughter strings for M. In the rule `S` $\rightarrow$ `NP VP`, for example, `S` is the mother category and `NP VP` is a predicate of concatenation. The set of regular predicates is constructed recursively from primitive terms as described below. A primitive term is a string-element predicate that may or may not be annotated with functional schemata.

category symbol [string elements]
e
?

A category symbol is a sequence of characters that matches nodes labeled with that sequence. Any alphanumeric character is allowed in the sequence, and punctuation marks (such as {, |, &, space...) that have any of the special meanings described below may also be included provided they are preceded by the back-quote character ‘. Back-quote thus serves as an escape character for the regular predicate notation, making it possible to mention ordinary punctuation marks in a grammar.

The symbol e is the default "epsilon" symbol for an LFG grammar: the predicate e matches against the empty string of categories. In combination with the union operator described below, this offers one way of expressing optionality. For certain kinds of grammars, such as a grammar over phonological segments, it may be convenient to let the letter e stand for the ordinary category with that label, and to use some other symbol (for example, eps or 0) as the epsilon symbol. The epsilon symbol can be changed by means of the EPSILON component of a configuration.

The symbol ? denotes the disjunction of all categories that appear anywhere else in a rule. Thus, the following two (nonsense) rules are equivalent:

```
NP  → DET ? A N.
NP  → DET {DET | A | N} A N.
```

The ability to denote the complete set of categories may not seem particularly useful in isolation, but it is a powerful notational tool when intersected with other predicates. For example, the intersection

```
[?* NP ?*] & [?* VP ?*]
```

is an intuitive way of denoting the set of strings containing at least one NP and one VP in either order and separated by any number of categories drawn from NP, VP, and other ones mentioned elsewhere in the rule.

element [primitive terms]
element : *schemata* ;

The unadorned string elements just introduced are primitive terms of the regular predicate notation. Every string element can also be annotated with a set of functional schemata. These are attached with a colon and terminated by a semi-colon (the semi-colon may be omitted when a punctuation mark that terminates some enclosing predicate is present). When a term with schemata is matched in a daughter string, the schemata are instantiated and added to the f-description.

Attaching schemata to an e that is disjunctive with some other predicate permits functional requirements to be imposed in trees that do not satisfy that other predicate. For example, the disjunctive predicate

```
{NP:(↑ SUBJ)=↓; | e: (↑ SUBJ PRED)='PRO';}
```

defines the subject to be a dummy pronoun if an overt subject NP does not appear. Since there is no node beneath the epsilon and hence no lexical information, ↓ has no meaning in a schema attached to e and such a schema is therefore disallowed.

GWB supports the special abbreviatory convention that Kaplan and Bresnan proposed for non-epsilon terms that either have no schemata attached or whose schemata do not contain the daughter metavariable ↓. If such a term appears in a positive context, that is, not in the scope of any of the complementation predicates described below, then it is interpreted as if the schema ↑=↓were attached to it. The term `VP:(↑ TENSE) ↑=↓;` is thus equivalent to the abbreviated `VP:(↑ TENSE);`. In the unmarked case mother and daughter f-structures are simply identified with each other. This permits the specification of head relations to be suppressed so that the assignment of particular grammatical functions can stand out.

The full set of regular predicates is constructed from this base of primitive terms. In the following recursive description the symbols *p* and p_i denote arbitrary regular predicates.

$p_1\ p_2\ \cdots\ p_n$ [concatenation]

The concatenation predicate is expressed by a sequence of predicates separated by white-space (spaces, tabs, carriage returns). This predicate is satisfied by an ordered sequence of substrings satisfying each of the predicates p_1, p_2, ... p_n in turn. You may interchangeably use spaces or carriage returns to separate the concatenated predicates as you type them in, but as you have seen, GWB always arranges the items vertically when it displays them for editting.

p * [iteration or repetition]
p +
p#n
p#n#m

p * denotes the Kleene closure of *p* : it is satisfied by *zero* or more substrings each of which satisfies *p*. *p*+ is satisfied by a sequence of *one* or more substrings satisfying *p*.

The predicates with # involve a specified number of repetitions. *p#n* is satisfied by the concatenation of exactly *n* substrings satisfying *p,* where *n* is a natural number. The predicate `NP#2`, for example, requires exactly two adjacent NP's to appear. *p#n#m* is satisfied by an expression that has at least *n* but not more than *m* repetitions of *p*. *m* is either a natural number greater than *n* or else it is the symbol *, indicating an unspecified upper bound. Thus, `NP#3#5` requires 3, 4, or 5 NP's in a row, while `NP#3#*` requires 3 or more NP's. Note that NP* and `NP#0#*` are equivalent as are `NP+` and `NP#1#*`. As a special convenience, repetition factors are allowed to appear between a categorial symbol and the colon that introduces its schemata. For example, `PP*:(↑ (↓ PCASE))=(↓ OBJ);` is accepted as a variant form of `PP:(↑ (↓ PCASE))=(↓ OBJ);*`.

[*p*] [grouping]

Square brackets are used merely to explicitly mark that the components of *p* are to be treated as a unit with respect to other enclosing predicates. This is useful when the normal precedence conventions would associate these components in an unintended way or when you are unsure exactly what the normal precedence conventions would do. It is always safe to include brackets; if the brackets turn out to be unnecessary the system will delete them when the rule is printed out for editing or when it is printed on a permanent file.

$\{ p_1 | p_2 | ... | p_n \}$ [disjunction or union]

This predicate is satisfied by any string meeting the conditions of at least one of the individual p_i predicates.

(p) [optionality]

This predicate indicates that a substring satisfying p may or may not be present. The predicate (p) is equivalent to the disjunction $\{ p | e \}$ and the repetition *p*#0#1.

$p_1 \& p_2$ [conjunction or intersection]

This predicate is satisfied by any string meeting the conditions of *both* p_1 and p_2.

~p [negation or complementation]

This predicate is satisfied by any string that does not satisfy the predicate *p*. For example, the predicate `~[NP:(↑ SUBJ)=↓; VP]` will match any string of nodes provided that it does not consist of a subject NP followed by a VP. The meaning of complementation is quite obvious when the predicate *p* contains no functional annotations: it denotes just the set of all strings not in the regular language specified by *p*. When schemata are attached, the predicate denotes the complement of the set of strings whose categorial requirements and associated schemata are both satisfied. This includes all the node-sequences that belong to the unannotated complement (such as VP NP, NP NP VP, VP NP NP) but also includes all strings that satisfy

```
NP:(↑ SUBJ)~=↓;  VP
```

The f-structures defined by an annotated complementation predicate may have unexpected properties because of the nonconstructive meaning that LFG assigns to negative functional schemata. Also to avoid confusion, in the scope of a complementation predicate terms without ↓-containing schemata are *not* interpreted as containing ↑=↓.

$p_1 - p_2$ [relative complementation]

This predicate is satisfied by any string that satisfies p_1 but does not satisfy p_2. It is exactly equivalent to p_1 & ~p_2. As an equivalence going the other way, note that ~*p* denotes the same strings as ?*–*p*. Note that the schemata in p_2 may be assigned negative, nonconstructive meanings and any intended ↑=↓ interpretations must be stated explicitly.

p [term complementation]

The term complement is any single category-schemata combination that does not satisfy the requirements of *p*. This is the same as ?–*p* in contrast to the ?*– *p* of the usual complement. The term-complement of an unannotated category is simply the disjunction of all the other categories that appear in the same rule: `\NP` is the disjunction `{VP|AP}` if `VP` and `AP` are mentioned elsewhere in the rule. Given De Morgan's laws, the predicate `\NP:(↑ SUBJ)=↓;` is equivalent to the disjunction `{ NP:(↑ SUBJ)~=↓| VP | AP }`. Note also that *p** is equivalent to ~[?* *p* ?*] in the special case where p is a primitive term. As for the other complementation predicates, any ↑=↓ interpretations intended in p must be stated explicitly.

p_1 / p_2 [ignore or insert]

This predicate denotes strings that satisfy p_1 when substrings satisfying p_2 are freely removed or ignored. Alternatively, it denotes strings formed from the language of p_1 by arbitrarily inserting strings from the language of p_2. For example, the following rule expresses the generalization that adverbs can be freely inserted after the object of English VP's:

```
VP → V  (NP:(↑ OBJ)=↓)
        [(NP:(↑ OBJ2)=↓) PP*:(↑ XCOMP)=↓] / ADV:↓∈(↑ ADJ).
```

Without this special notation, an `ADV*` with its schemata would have to be repeated several times and the generalization would be lost.

$p_1 < p_2$ [linear precedence]
$p_1 > p_2$

These predicates impose a linear ordering on strings. $p_1 < p_2$ is satisfied by any string in which there is no substring satisfying p_1 in front of a substring satisfying p_2. This is equivalent to ~[?* p_2 ?* p_1 ?*], which reduces to [\ p_2]*[\ p_1]* if p_1 are p_2 are primitive terms. The opposite ordering restriction is expressed by the $p_1 > p_2$ predicate. Thus the rule

```
S →  [NP:(↑ SUBJ)=↓, VP] & NP<VP.
```

is equivalent to the more familiar rule

```
S →NP:(↑ SUBJ)=↓; VP.
```

but expresses a different set of generalizations. Since the linear precedence predicates are defined in terms of complementation, the general caveats about schemata in negative contexts apply.

p_1 , p_2 [shuffle]

This predicate is satisfied by any string that can be formed by "shuffling" together the elements of strings satisfying p_1 and p_2. That is, the elements that satisfy p_1 must come in their p_1-determined relative order and similarly for the elements satisfying p_2, but elements satisfying those two predicates can be freely intermixed (provided they maintain their prescribed relative order). Thus, `[A B],[X Y]` is satisfied by any of the strings

```
A B X Y    A X B Y
A X Y B    X A B Y
X A Y B    X Y A B
```

As other examples, `NP:(↑ SUBJ)=↓, VP` is satisfied by strings containing one subject NP and one VP in either order, and `NP#2#4,VP#3` would be satisfied by strings containing three VP's intermixed with between two and four NP's. GWB's shuffle predicate provides a regular generalization of the immediate-dominance specifications of other syntactic theories, such as GPSG (Gazdar *et al.*, 1985).

" *any string of characters* " [comment]

Character strings enclosed in double-quotes are treated as invisible comments and assigned no other interpretation. They may appear anywhere inside a regular predicate expression except between the characters of what you intend to be an atomic category symbol. This allows you to embed justifications, explanations, or other annotations at the most appropriate positions.

When these predicates are put together in a complicated formula, the meaning of the combined expression depends on the relative scope that is assigned to the different operators. If you are ever in doubt about the relative scope of operators, you can always include grouping brackets to make your intended meaning clear; unnecessary brackets will be eliminated when GWB prints or displays the expression. Without explicit grouping brackets, the precedence of operators is determined by a simple set of conventions. With one exception, a potential scope ambiguity between most of the operators (* + # ~ – \ / , < >) is resolved so that the left-hand operator has wider scope than the right-hand one. Thus the expressions in the first column receive the interpretation notated more explicitly by the entries in the second:

```
A<B–C      A<[B–C]
A–B<C      A–[B<C]
A/B*       A/[B*]
~A, B      ~[A, B]
```

The one exception to this general principle involves combinations of the term complement and iteration operators. Since the term complement of an iteration makes no sense, an expression such as `\A*` is more conveniently interpreted as `[\A]*` instead of `\[A*]`. The operators listed above all bind more tightly than the intersection and concatenation operators, so that A<B&C is interpreted as `[A<B]&C` and `A/B C` is interpreted as `[A/B] C`. Potential ambiguities between intersection and concatenation are resolved in favor of a tighter binding for the intersection operator. Thus, `A&B C` is interpreted as `[A&B] C`. The notations used for the remaining operators (optionality, grouping, and union) is such that their interpretation relative to each other and to other operators is always unambiguous.

When a c-structure rule is activated, GWB translates its right-side regular predicate into a deterministic, minimal-state finite-state automaton. In gaining an understanding of the significance of various predicate combinations, it may sometimes be helpful to examine the structure of this underlying automaton. If the Interlisp function `PPRN` (for <u>P</u>retty-<u>P</u>rint <u>R</u>ule <u>N</u>etwork) is applied to the name of the category, GWB will print a description of the states and arcs of the corresponding automaton. For example, if the currently active sentence rule is

```
S → (NP: (↑ SUBJ)=↓)
     VP: (↑ TENSE).
```

Typing `(PPRN 'S)` in an Interlisp Executive window will result in the following display:

```
[NAME = S
 Closed sigma = {NP VP}
 Deterministic Minimized]

START:  State S.0
  NP to S.1
        (↑ SUBJ)=↓
  VP to S.2
        (↑ TENSE)
```

```
State S.1
  VP to S.2
       (↑ TENSE)

State S.2 [final]

----
```

This indicates that state `S.0` is the initial state of the automaton and has two transitions, a subject `NP` leading to state `S.1` and a `VP` leading to the final state `S.2`. The `VP` transition appears at both states `S.0` and `S.1` because the `NP` is optional in the rule. `PPRN` can be given the name of an open file as a second argument, in which case the text will be printed to that file instead of to the Executive Window. If instead of an open file, the atom `TEDIT` is provided (e.g. `(PPRN 'S 'TEDIT)`), you will be prompted for a TEdit window and the text will be printed there. This option permits you to scroll back and forth and to add your own annotations before saving the text of the rule-network on a file.

2. Empty strings and empty nodes

Apart from their use in expressing various kinds of generalizations, the epsilon and optionality regular predicates in certain arrangements permit the right-side of a c-structure rule to be satisfied by an empty string of daughters. Such empty nodes or traces are used in many syntactic theories (particularly Government Binding theory) to code in phrase-structure trees the positions of null anaphors, the extraction sites of long-distance dependencies, and other kinds of coindexing conspiracies. The original LFG theory of long-distance dependencies (Kaplan and Bresnan, 1982) was also based on c-structure epsilons appearing in certain control domains. Kaplan and Zaenen (1989b) criticized the motivation for the c-structure approach to extraction phenomena. They proposed the formal device of functional uncertainty as the basis of a linguistically superior account that is much more compatible with LFG's general functional orientation. This approach deals directly with functional assignments and does not rely on empty nodes or traces as surrogate phrasal coding devices for implicit functional identities. The functional approach thus permits empty nodes to be banished from c-structure trees without losing any linguistic generalizations. In the absence of empty nodes the possible phrase-structure trees for a sentence are much more strongly constrained by the actual words of the string.

GWB implements the modern, functional uncertainty approach to long-distance dependencies (see Section 4 below), and by default it also maintains a prohibition against empty c-structure nodes. Thus, the regular predicate notation allows you to use optionality or unannotated epsilons (e's) to factor phrasal generalizations in the right sides of rules, and the normal interpretation of the notation might therefore allow a rule to be satisfied by the empty string. GWB disallows that possibility by imposing a special interpretation: any rule of the form $M \rightarrow p$ is interpreted by default as the rule $M \rightarrow [p - \mathrm{e}]$. For example, suppose that a language permits NP's to be realized with or without noun heads as described by the rule `NP` → `(DET) A* (N)` and that sentences are defined by the rule `S` → `NP VP`. GWB would not assign a c-structure to the string `Walks the dog`, since it treats the rule as requiring that at least one of `DET`, `A`, or `N` be present.

The default interpretation is more subtle when the predicate contains an e annotated with functional schemata, as in the rule

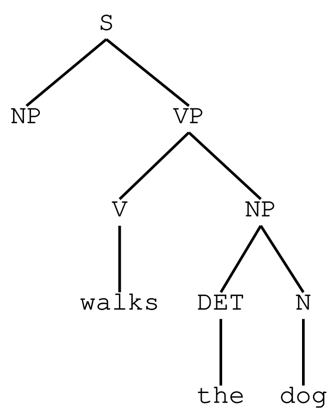
```
S  → {NP:(↑ SUBJ)=↓| e: (↑ SUBJ PRED)='PRO'}  VP.
```

This installs a pronominal subject in the case where an NP is not present in the string. GWB achieves the intended effect of this specification without producing a c-structure containing an empty e node by shifting the schemata to a non-empty node either to the left or right of the e. In essence, GWB treats this rule as the equivalent but more redundant

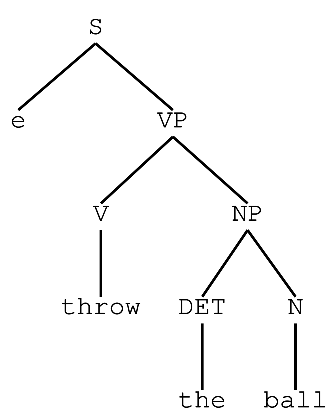
`S → {NP:(↑ SUBJ)=↓; VP| VP: (↑ SUBJ PRED)='PRO'}.`

Given the convention of subtracting e's from every right-side predicate, there is always at least one non-empty node to the left or the right of an e that its schemata can be transferred to. Since schemata containing ↓ cannot be associated with e predicates, these schemata can be shifted to non-empty categories without changing their meaning.

GWB also makes it possible to explore alternative theories in which empty nodes play a more explicit role in c-structure representations, even though these are not in good style according to current LFG standards. The default behavior with respect to empty nodes is controlled by two parameters that can be set in the `PARAMETERS` section of an analysis configuration (see Section 6 below). Setting the parameter `ALLOWEMPTYNODES` to `TRUE` suppresses the default behavior of subtracting the empty string from the right hand side of every rule. If this is done, GWB will interpret the NP rule above as allowing empty NP nodes and the string `Walks the dog` would be assigned the following c-structure:



The second parameter, `PRESERVEEPSILONS`, affects the treatment of e's that appear explicitly in rules. If this parameter is set to `TRUE`, then schemata will not be shifted from e's, e transitions will be preserved in the underlying finite-state automaton, and the resulting c-structures will be displayed with explicit e nodes. Given the S rule above, the string `Throw the ball` would be assigned the following c-structure:



The f-structure corresponding to this c-structure will have a pronominal subject.

3. Regular abbreviations

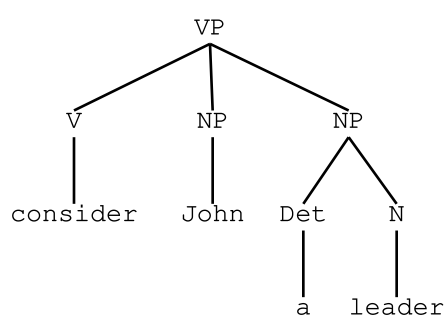
There may be several reasons for introducing a category and a c-structure rule that defines the realization of its daughters. Depending on your theoretical stance, the existence of a node in c-structure tree may be motivated by phonological or intonational evidence that you intend c-structure arrangements to account for, or by a universal theory of category/function correspondences as suggested by Bresnan (1982a). But it may also be the case that a category and c-structure rule is introduced for grammar-internal reasons, as a means of collecting in one place a specification of complex category sequences that would otherwise be replicated in several different rules. The purpose of such a category is to help in expressing generalizations that hold across several different grammatical constructions, but this may have the undesired side-effect of cluttering up the c-structure trees with nodes that have no extrinsic justification. GWB provides separate notational devices, meta-categories and c-structure macros, that permit the factoring of recurring grammatical patterns without forcing an unmotivated elaboration of linguistic representations.

3.1. Meta-categories

A meta-category permits certain kinds of cross-categorial generalizations to be expressed. In English, for example, the open-complement function XCOMP may be associated with any of the constituents AP, PP, NP, or VP', and we might write the VP rule as

```
VP   →   V
         (NP:(↑ OBJ)=↓)
         (NP:(↑ OBJ2)=↓)
         ({ AP:(↑ XCOMP)=↓
           |PP:(↑ XCOMP)=↓
           |NP:(↑ XCOMP)=↓
           |VP':(↑ XCOMP)=↓}).
```

and expect to see trees such as

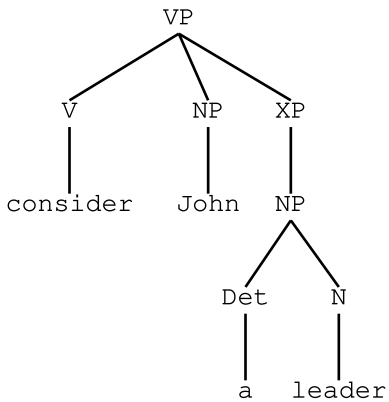


We could bring out the common functional association by introducing a new ordinary category XP:

```
VP   → V
        (NP:(↑ OBJ)=↓)
        (NP:(↑ OBJ2)=↓)
        (XP:(↑ XCOMP)=↓).

XP   → {AP|PP|NP|VP'}.
```

but then the XP would show up as an extra, presumably unwanted and otherwise unmotivated, node in the tree:



If instead we specify XP as a meta-category, the abbreviated version of the VP rule would still produce the original tree. A meta-category is distinguished from an ordinary category by a very small change in the rule that defines its regular-predicate expansion: we separate the left and right sides by an equal-sign instead of a rewriting arrow:

```
XP   = {AP|PP|NP|VP'}.
```

Rules that reference the category symbol XP, such as the VP rule, are not modified when, by virtue of this simple change to its expansion, XP is converted from an ordinary category to a meta-category.

As another meta-category example we might consider the VP category itself. The status of VP as a constituent in all languages has been the subject of much linguistic debate. The arguments for the existence of such a constituent, at least in languages like English, are mostly transformational in nature, having to do with movements, copies, and deletions of common constituent patterns under the assumption that transformational rules only operate on single nodes. The same facts can be accounted for in LFG by letting VP be an ordinary category that appears in the predicates that define several other rules, as in these S and VP' rules:

```
S   → NP:(↑ SUBJ)=↓; VP.

VP'   → (to) VP.
```

But the VP does not correspond to a distinct functional unit, and the generalizations about surface constituent patterns can equally well be expressed by defining VP as a meta-category:

```
VP   = V
        (NP:(↑ OBJ)=↓)
        (NP:(↑ OBJ2)=↓)
        (XP:(↑ XCOMP)=↓).
```

This will give rise to flatter c-structures than the ordinary VP rule would provide, but the surface strings and f-structures will be exactly the same.

In general, a symbol *M* is defined to be a meta-category by a statement of the form

$$M = p.$$

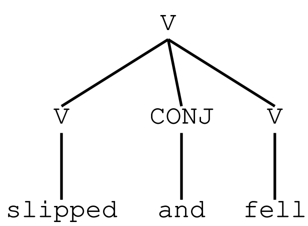
where *p* is an arbitrary regular predicate. Any other rule that mentions *M* is interpreted as requiring the appearance of substrings satisyfing *p* in all positions where a node labeled *M* would otherwise be required. Unless `ALLOWEMPTYNODES` is `TRUE`, the empty string is not included as a substitution for *M*, even if *p* provides for it.

Any schemata attached to occurrences of *M* (such as `(↑ XCOMP)=↓` in the VP rule) are attached instead to every node required by *p*. Thus, converting a category from ordinary to meta (merely by switching → to =) does not normally change the sentences recognized by the grammar nor the f-structures assigned to them. However, distributing the schemata across all the nodes may produce unexpected results if the schemata contain ↓ and *p* contains sequences instead of unit-length terms: the ↓ will denote a different f-structure at every node, and this may lead to unintended inconsistencies. For example, if VP is defined as a meta-category, the term `VP:↑=↓;` appearing in the S rule will have the presumably undesired effect of identifying the S f-structure with the `OBJ`, `OBJ2`, and `XCOMP` f-structures.

Meta-categories are implemented by expanding the rules in which they appear, and this process would not terminate for meta-categories that are self-recursive either directly or indirectly. When GWB encounters such a recursive expansion, it prints a warning message in the Prompt Window and substitutes the empty regular language `~?*` instead of *p* for the recursive appearance of *M*. This terminates the expansion.

3.2. Regular-expression macros

Meta-categories stand for what may be thought of as single units of c-structure; regular macros, on the other hand, expand to arbitrary regular predicates with no intuition that they correspond to singleton nodes. The cross-categorial treatment of constitutent coordination offers a good example. As described by Kaplan and Maxwell (1988b), the f-structure element corresponding to a coordination construction is the set of f-structures corresponding to the coordinated nodes. This holds independent of the categories involved. Thus, the verb coordination *slipped and fell* is assigned the following c-structure and f-structure:




```
{ [PRED   slip<(↑ SUBJ)>]
  [TENSE  PAST          ]
  [PRED   fall<(↑ SUBJ)>]
  [TENSE  PAST          ] }
```

This configuration can be specified by introducing a rule that provides an alternative to lexical specification for characterizing a V:

```
V  → V: ↓∈↑;  CONJ  V: ↓∈↑.
```

But adjectives, nouns, prepositions, and phrasal categories such as VP and AP can also be coordinated in just the same way. We can factor out the general pattern by defining a regular macro:

```
COORD(CAT) = CAT: ↓∈↑;  CONJ  CAT: ↓∈↑.
```

This defines `COORD` as the name of a regular macro with one formal parameter, `CAT`. We can now write

```
V  → @(COORD V).
```

as the formal equivalent of the rule above. The @ marks the invocation of the regular macro `COORD` and specifies that in this instance its parameter `CAT` is to be realized as `V`. The effect is first to substitute `V` for `CAT` wherever it appears in the regular expression that expands `COORD`, and then to substitute the result in place of `@(COORD V)` in the V rule. But we can now also invoke the `COORD` macro in other rules:

```
A → @(COORD A).
P → @(COORD P).
```

```
VP   → { V
          (NP:(↑ OBJ)=↓)
          (NP:(↑ OBJ2)=↓)
          (XP:(↑ XCOMP)=↓)
         |@(COORD VP)}.
```

etc.

The expansions will be different in each case, because of the differing realizations of the formal parameter. This example illustrates two of the features that distinguish regular macros from meta-categories: Their invocations are explicitly marked with the special character @, and they may have formal parameters whose realizations are specified in the invocation. In an invocation the name of the macro and the realizations of any formal parameters are grouped together in parentheses. In the definition of the macro, the names of the formal parameters are given in parentheses between the macro name and the equal-sign.

As another macro example, consider a strategy for creating an ID/LP-style c-structure grammar in GWB. You might start by factoring the S rule into an ID component and an LP component:

```
S   → [NP:(↑ SUBJ)=↓, VP] & NP<VP.
```

But you might then want to view the linear-precedence predicate of this rule as an instance of a more general set of conditions it shares with a number of other rules. You could define this condition as a regular macro and then reference it by name in all the appropriate rules:

```
LP   =  NP<{VP|AP|PP|VP'} & {V|A|N|P}<{VP|AP|NP|PP|VP'}.
```

The first clause of the LP predicate asserts that NP's come before other phrasal categories, and the second indicates that the language is head-initial. This particular macro has no formal parameters, so there are no parentheses in its definition. Also, parentheses can therefore be omitted from its invocations, as in the following rules:

```
S   →  [NP:(↑ SUBJ)=↓, VP] & @LP.
VP  →  [V, NP:(↑ OBJ)=↓, NP:(↑ OBJ2)=↓, XP:(↑ XCOMP)=↓] & @LP.
```

etc.

The @ at each invocation is thus the only indication that LP is a regular macro and not a meta-category. The macro invocations enable the linear-precedence constraints named by LP to be shared by these and other rules. The constraints are imposed (by intersection) as further conditions on the immediate-dominance specifications. In this example, clearly, the macro expansion has no interpretation as a single node.

A regular macro M is defined by a statements of either of the forms

$$M = p.$$

$$M\ (param_1\ param_2 \ldots param_n) = p.$$

Where p is a regular predicate. The first form is exactly the same as for the definition of a meta-category; the interpretation as either a macro or meta-category is determined at each invocation by whether or not it is marked with an @. The second form provides for some number n of formal parameters, where each $param_i$ is a symbol that presumably appears somewhere in p. The i^{th} realization at a particular invocation is systematically substituted for the symbol $param_i$ everywhere in p. Any $param_i$ symbol in the parameter list can be followed by a question-mark (for example, `CAT?`). This indicates your intention to omit a realization for that particular parameter (and presumably all other parameters to its right) in some invocations, and to have the predicate *p* be treated for that invocation as if every appearance of that parameter name had simply been treated as a comment. The question-mark suppresses the printing of the warning message that is normally provided when there is a mismatch between the number of realizations and the number of parameters.

An invocation of the macro *M* appearing in another rule (or macro or meta-category definition) is a regular-predicate term of either of the forms

@*M*

@($M\ real_1\ real_2\ ...\ real_n$)

The first is appropriate for a macro with no realizations. The realization expressions $real_i$ in the second form are themselves arbitrary regular predicates, perhaps with attached schemata. Those expressions will be substituted in place of the corresponding $param_i$ symbols in the predicate defining *M*. As with meta-categories, a warning message will be printed and the empty regular language `~?*` will be substituted if a cyclic expansion of *M* is detected. Furthermore, unless a parameter name in the macro definition is followed by a question-mark, a warning message will also be printed when a rule containing an invocation is activated and the number of realizations is not the same as the number of parameters specified in the definition for *M*.

Because regular macros do not correspond to categories, their expansions are treated somewhat differently. First, the empty-string expansion of a regular macro is retained whether or not `ALLOWEMPTYNODES` is `TRUE`. Second, given the intuition that a regular macro expands to an arbitrary regular predicate and not a singleton node, it makes no sense to distribute schemata attached to the macro invocation to all the terms in the expansion. Indeed, it makes little sense to attach schemata to the macro invocation at all, but for convenience, we allow them to be attached but give them the following distinctive interpretation: Any macro invocation of the forms

@*M*: *schemata*;

@($M\ real_1\ real_2\ ...\ real_n$): *schemata*;

are interpreted, respectively, as the sequences

@*M* e: *schemata*;

@($M\ real_1\ real_2\ ...\ real_n$) e: *schemata*;

That is, the schemata are interpreted as if they were attached to an epsilon symbol that is concatenated with the macro-expansion predicate. On this interpretation, schemata containing ↓ have no meaning and are therefore disallowed. Since this e is implicit in the interpretation of the macro invocation and does not actually appear in the grammar, it is not preserved even when the parameter `PRESERVEEPSILONS` is `TRUE`.

3.3. Phantom nodes

In some circumstances there may be motivation to treat a given pattern of c-structure nodes as the daughter sequence of a single mother node, while in other situations there may be no reason to isolate that pattern of nodes as a separate subtree. In the first case the pattern would appear as the right-side of an ordinary rewrite rule, while for the second case the pattern might appear as a meta-category definition. We used the English `VP` above to illustrate how a meta-category definition enables a pattern of nodes to occur in several c-structure positions without necessarily giving rise to a distinct constituent. The meta-category enables the c-structure generalizations to be factored out of other rules without introducing otherwise uninteresting nodes in the tree. Returning to that example, we also note that English `VP`'s can be coordinated, and a separate node does seem necessary for the coordination construction so that the f-structure of each conjunct can be referred to as a separate unit. GWB allows a single `VP` specification to be used in these two different ways. The `VP` is defined as an ordinary rewrite rule, not as a meta-category, as in the following example:

```
VP   → { V
          (NP:(↑ OBJ)=↓)
          (NP:(↑ OBJ2)=↓)
          (XP:(↑ XCOMP)=↓)
        |@(COORD VP VP)}.
```

The `VP` references in the `COORD` macro invocation will result in the appearance of explicit `VP` constituents in this construction. In the situations where there is no need or desire to have a separate `VP` node and the `VP` would be better treated as a meta-category, that effect can be achieved by referring to the `VP` as if it were a macro without parameters. Thus the `VP` references in the following rules will not result in separate constituents in the c-structure:

```
S   → NP:(↑ SUBJ)=↓; @VP.

VP'   → (to) @VP.
```

In general, when a category defined by a rewriting rule is referenced as a macro, GWB treats that invocation as if it were a reference to a meta-category defined by the right-side of the rule. The conventions for interpreting schemata attached to meta-categories are applied to each such invocation. We say that such an invocation of a rewriting rule gives rise to a *phantom node*, to distinguish this kind of reference from its use as an ordinary phrasal expansion. Since it is sometimes difficult to debug a grammar when the phantom nodes do not appear in the tree, GWB allows you to indicate that you prefer temporarily for phantom nodes be included in the c-structure. This is done by selecting the *Show phantom nodes* item in the menu of the Sentence Input Window, as discussed in Section IV.6.

4. The functional description language

The schemata that appear in c-structure rules and in lexical entries are written in GWB's functional description language. Except for the absence of bounded-domination metavariables, this is an extension of the notation presented by Kaplan and Bresnan (1982) and is used to express conditions that the f-structures and other projections corresponding to particular nodes must satisfy. Explicit projection symbols may be used in codescriptive statements that characterize structures representing different aspects of linguistic organization. The f-description language includes designators that denote symbols, semantic forms, f-structures, and other projection structures. Properties of those entities are asserted by propositions constructed from a small number of predicates and relations applied to those designators, and those propositions are combined together by the usual Boolean operators.

4.1. Designators

Designators are terms in the description language that stand for elements in the structures representing different projections. They are defined recursively in the following way:

symbol [attribute and value symbols]

A symbol is a sequence of characters that are not separated by white-space (spaces, carriage returns, tabs) or by other delimiters in the f-description language (parentheses, brackets, predicate and relation symbols). A symbol character-sequence denotes the unique symbol with that name. The symbol may be an attribute in the f-structure (a feature or grammatical-function name) or in some other attribute-value structure, it may be the value of some attribute, or, in some cases, it may be both. The symbol `OBL-GOAL`, for example, may be both a grammatical function and the value of an attribute such as `PCASE`. LFG differs from many "unification-based" grammatical formalisms in that attributes and values are not necessarily disjoint. Symbols may also appear in special structures such as semantic forms, where they also retain their uniqueness property.

*symbol_* [instantiated symbols]

Kaplan and Bresnan (1982) used the single-quote notation to denote semantic forms, as described below. These entities were used to formalize several different aspects of the interaction between the syntactic and semantic components, including a notion of individuation or unique instantiation. Further experience and understanding has suggested that unique instantiation is a formal device with motivation for purely syntactic features, and GWB therefore provides for instantiated feature values that carry no semantic entailments. An instantiated syntactic feature value is indicated by suffixing the underscore character _ to the name of an ordinary symbol. The notation is intended to suggest the attachment of an indexing subscript.

Equalities of two ordinary symbols with the same spelling, such as `NOM=NOM`, are always true. In constrast, instantiated symbols, even those with identical spellings, are taken to denote distinct objects, so that equations such as `OBL-ON_=OBL-ON_` are never satisfied. For feature values that are ordinary symbols, the existential constraints in the f-description language can be used to discriminate between the absence of any defining equation and the presence of *at least one* but *possibly several* such equations. If instantiated symbols are involved, the truth value of a Boolean formula can further discriminate between the appearance of *exactly one* defining equation as opposed to more than one.

As an illustration, suppose that the case-marking feature of a preposition is specified as an ordinary symbol, for example, by the schema (↑ `PCASE`)=`OBL-ON` in the lexical entry for `on`. This will give rise to the proper functional configuration for a sentence like *John relied on Bill*. If the within-clause role of a fronted prepositional phrase is determined by functional uncertainty, the proper result will also emerge for the question *On whom did John rely?* The ungrammatical *On whom did John rely on Bill?* will be rejected because the conflicting nominal predicates will both be assigned to the oblique-object grammatical function. But the equally ungrammatical *On whom did John rely on?* will not be disallowed because it involves doubling of only the semantically empty features of the preposition. Changing the `PCASE` value to an instantiated symbol via (↑ `PCASE`)=`OBL-ON_` will rule this out because of a clash between the two instantiations of `OBL-ON_`.

An instantiated symbol is interpreted as the corresponding ordinary symbol when it appears as an f-structure attribute, so that (↑ `ON-OBJ`) and (↑ `ON-OBJ_`) are exactly equivalent. The change of `PCASE` to an instantiated value thus still allows the proper assembly of an oblique object according to the schema (↑ (↓ `PCASE`))=(↓ `OBJ`). An instantiated symbol is also treated as the corresponding ordinary symbol when it is used as a value in constraining as opposed to defining schemata. The existential constraint (↓ `PCASE`) will be satisfied if the `PCASE` value is `ON-OBJ_`, and so will the constraining equation (↓ `PCASE`) =c `OBL-ON`.

Note that an equality between two designators for the *same* instantiation of an instantiated symbol will be true, as in the conjunction of (↑ `PCASE`)=`ON-OBJ_` and (↑ `PCASE`)=(↑ `PCASE`).

When an instantiated symbol is presented in an f-structure display, the underscore is replaced by a numeric subscript so that different references to the same instantiated symbol are easy to recognize.

'*function*' [semantic forms]
'*function*<a_1... a_k>'
'*function*<a_1... a_k>n_1 ... n_l'

A semantic form is an expression enclosed in single-quotes. It makes visible in the syntax only those aspects of semantic representation that interact in some way with syntactic properties. In particular, it defines the mapping between a predicate's governed grammatical relations and the particular argument positions of the semantic relation denoted by the *function* designator. Also, as discussed by Kaplan and Bresnan (1982), a semantic form acts like an instantiated symbol, carrying in this case a notion of semantic individuation: two semantic forms are taken to denote distinct semantic entities (and hence equality never holds between them) even if they are specified by identical expressions. The entities denoted by a semantic form may have other significant semantic properties, such as various kinds of scoping and entailment relations, but those have no syntactic consequences.

Included within the quotes of a semantic form is a designator for a semantic *function*, optional designators for one or more thematic argument functions a_1... a_k enclosed in angle brackets, and optional designators for one or more nonthematic grammatical functions n_1 ... n_l appearing after the brackets. The *function* is typically a symbol designating an atomic semantic relation (e.g. '`GIRL`', '`WALK`<(↑ `SUBJ`)>'), but other types of designators are also allowed. For example, '(↓ `PRED`)<(↑ `SUBJ`)>' might denote the semantic entity corresponding to a predicate nominal, constructed by applying a common-noun meaning (↓ `PRED`) to a single argument provided by a subject.

The thematic and nonthematic grammatical-function designators will typically be function-application expressions such as (↑ `SUBJ`), (↑ `OBJ`), etc. (which, for convenience, may be typed simply as `SUBJ` and `OBJ`). The attribute names in these expression denote governable grammatical functions, and thus they should match one of the `GOVERNABLERELATIONS` patterns specified in the active configuration (see Section 6). The special symbol `NULL` may also appear and can be used to encode the fact that a particular semantic argument has no surface manifestation. The symbol `NULL` thus serves as the null function Ø discussed by Bresnan (1982b).

GWB's implementation of LFG's completeness and coherence conditions depends on the thematic and nonthematic grammatical-functional designators. An f-structure is incomplete if it has a semantic-form `PRED` value with a a_i or n_i designating a grammatical function not locally present in the f-structure. An f-structure is incoherent if it contains a local semantic-form `PRED` value and a governable grammatical function (as specified by the `GOVERNABLERELATIONS` patterns of the current configuration) and that function is not sanctioned by one of the a_i or n_i in the `PRED` value (as noted in Section 3.2 of Chapter II, this is slightly different than the condition originally proposed by Kaplan and Bresnan, 1982). The distinction between the thematic and nonthematic designators (and the bracket-external notation for nonthematics) follows Bresnan (1982a): the thematic functions are mapped to arguments of the sematic relation and thus must satisfy its selectional restrictions. The nonthematic functions are purely syntactic and may be filled, for example, by semantically empty expletives. The equi predicate *persuade* thus differs from the raising predicate *expect* only in how grammatical functions are allocated to the thematic and nonthematic sets:

'PERSUADE<(↑ SUBJ) (↑ OBJ) (↑ COMP)>'
'EXPECT<(↑ SUBJ) (↑ COMP)> (↑ OBJ)'

GWB enforces this formal distinction by also marking f-structures as incomplete if a thematic function is present but does not have a semantic-form PRED value. No additional requirements are imposed on the values of nonthematic functions.

As described below, the individual components of a semantic form may be referenced in functional descriptions by use of the special attributes FN, ARG1, ARG2..., NOTARG1, NOTARG1, ...

↑
↓ [f-structure metavariables]

These designators denote the f-structures corresponding to nodes of the c-structure via the c-structure to f-structure correspondence ϕ. The ϕ correspondence, or projection, was introduced as a fundamental organizing mechanism by Kaplan and Bresnan (1982), but its role in interpreting the f-structure metavariables was not formally elaborated. ↓ was instantiated to the f-structure corresponding to the current node (the node matching the category-symbol containing the ↓ annotation) and ↑ was instantiated as the f-structure corresponding to the mother of the current node. These notions were formalized more carefully in Kaplan (1987, 1989, 1995): If * denotes the node matching a given category symbol annotated with schemata containing ↓, then that ↓ is taken as a convenient abbreviation for ϕ*, the result of applying the projection ϕ to the matching node. If M denotes the function that takes a c-structure node into its mother, then ↑ is interpreted as a convenient abbreviation for ϕM*. Defining the f-structure metavariables explicitly in terms of ϕ makes it easier to formalize relations such as functional precedence, which are based on the inverse of the ϕ correspondence. As discussed at the end of this section, GWB also supports * and M* as designators of the matching node and its mother, and this allows for correspondences that map from c-structure nodes to information structures other than the f-structure.

(*d r*) [function application and uncertainty]

A parenthetic expression stands for the result of applying the monadic function denoted by the designator *d* to an argument specified by *r*, a regular predicate over designators. In the common case, *d* is an f-structure metavariable and *r* is merely a symbol designator such as SUBJ, OBJ, CASE, or PERSON that specifes the name of a grammatical function or feature. The designator (↑ PERSON) thus denotes the value of the attribute PERSON in the f-structure corresponding to the current node's mother. In the somewhat more general case, *r* is a sequence of symbol designators, as in (↑ SUBJ PERSON). As defined by Kaplan and Bresnan (1982), this denotes the result of applying the function denoted by (↑ SUBJ) to the argument PERSON. In the most general case, *r* is an arbitrary regular predicate describing an arbitrary regular language. Kaplan and Zaenen (1989b) assign an existential meaning in this situation:

(d r)=v iff either

d=v and the empty string belongs to the language denoted by r, or

there is a symbol s such that ((d s) Suffix(s, r))=v

where Suffix(s, r) is the regular language $\{ y \mid sy \in r \}$

The indeterminacy about exactly which symbol is chosen at each point in the unfolding of this recursive definition is what gives rise to the intuition of "functional uncertainty". GWB solves propositions involving functional uncertainty according to the algorithm described by Kaplan and Maxwell (1988a).

The regular predicates that can be combined in r are drawn from the set of regular operators that make up the c-structure description language, described in Section 1 above. For example, the designator

```
(↑ COMP* GF – VCOMP)
```

denotes the infinite regular set whose strings begin with zero or more `COMP`'s and end with any grammatical function other than `VCOMP`. Note, however, that in uncertainty predicates the relative complementation hyphen must be surrounded with white space; this permits the hyphen to be included as a character in the middle of names such as `OBL-TO`. The basic terms of these functional regular predicates are designators denoting grammatical function and feature names. Typically, these will be primitive symbol designators, as in the example above, but expressions that denote such symbols are also allowed:

```
(↑ (↓ PCASE))
(↑ COMP* {SUBJ|OBJ|(↓ PCASE)})
```

Of course, the set of constraints will be unsatisfiable if other propositions require the designator in an uncertainty language to denote something other than a symbol (e.g. a semantic form or f-structure).

Functional predicates share the same epsilon symbol (e by default) as c-structure predicates. They may also make use of regular predicate abbreviations defined along with c-structure rules and abbreviations in c-structure Rule windows. Thus, you can include definitions such as

```
COMPFNS = {COMP|XCOMP}.
TERMFNS = {SUBJ|OBJ|OBJ2}.
```

and then make use of an uncertainty such as `(↑ COMPFNS* TERMFNS)`. There is no difference between metacategories and macros in functional regular predicates, since there are no schemata to be distributed differentially and there is no special treatment of epsilons to worry about.

For convenience, three special symbols come as predefined abbreviations:

`GGF` stands for the set of governable grammatical functions, those functions in the current grammar and lexicon that match the patterns specified in the `GOVERNABLERELATIONS` component of the active configuration.

`SGF` stands for the set of semantic functions, those functions in the current grammar and lexicon that match the patterns specified in the `SEMANTICFUNCTIONS` component of the active configuration. Typically these will include `ADJUNCT` and `MODIFIER` functions. They are not governed, but the f-structure is incomplete if their values do not have semantic-form `PRED`s.

`GF` stands for the universe of grammatical functions, the union of `GGF` and `SGF`. This set is also what ? denotes in functional regular predicates.

As originally presented by Kaplan and Bresnan (1982), it would be inconsistent for the designator *d* to denote anything other than an f-structure. GWB implements a modified form of the extension described by Kaplan and Maxwell (1989a) that gives meaning to the case when *d* designates a set. They proposed that if *d* designates a set, then (*d* *r*) denotes the entity (*g* *r*), where *g* is the generalization of all the elements in the set *d*. This permits a natural account of how features and functions distribute across the conjuncts in a constituent coordination construction. GWB provides for this behavior only for attributes that are not specified in the configuration as "nondistributive" (see Section 6 of this chapter). If *r* is a nondistributive attribute, then *d* is interpreted as if it is an ordinary f-structure containing a value for attribute *r*. In effect, *d* will have a mix of set and f-structure properties. This modification allows certain features, such as the identity of the conjunction, to be expressed directly in the f-structure, whereas the original Kaplan and Maxwell formulation forced the conjunction out of the f-structure and into another projection.

GWB implements another extension that permits reference to the components of a semantic form. If *d* denotes a semantic form, then the designator (*d* `FN`) denotes its *function* component, (*d* `ARG1`), (*d* `ARG2`), ... denote its a_1, a_2, ... argument components, and (*d* `NOTARG1`), (*d* `NOTARG2`), ... denote its n_1, n_2 nonthematic function components. Thus you can insure that some noun phrase is realized as a pronoun by asserting `(↑ PRED FN)=PRO`. Asserting `(↑ PRED)='PRO'` would not have this effect, because of the instantiation of semantic forms (`'PRO'='PRO'` is always false).

%symbol [local names]

A local name can be used as a variable whose scope is limited to the schemata associated with a particular category or lexical item. This makes it convenient to make repeated references to a single entity. For example, suppose that the `PERSON`, `NUMBER`, and `GENDER` features of a noun are grouped together in the value of an `AGR` attribute. A given lexical entry might assert that

```
(↑ AGR PERSON)=3
(↑ AGR NUMBER)=PL
(↑ AGR GENDER)=FEM
```

but these feature equations can be shortened by means of a local name `%A` that captures the value of the `AGR` feature:

```
(↑ AGR)=%A
(%A NUMBER)=PL
(%A PERSON)=3
(%A GENDER)=FEM
```

In this example the local name helps to abbreviate constraints whose effect could also be achieved in the usual more cumbersome way. When a functional uncertainty is involved, however, a local name can make it possible to express conditions that would otherwise be unstatable. Suppose that the three agreement requirements are to be asserted on the value at the end of a functional uncertainty, not on the value of ↑. The schemata

```
(↑ COMPFNS* TERMFNS AGR PERSON)=3
(↑ COMPFNS* TERMFNS AGR NUMBER)=PL
(↑ COMPFNS* TERMFNS AGR GENDER)=FEM
```

would not have the desired effect. Each of the uncertainties is treated separately and can be instantiated by *different* strings drawn from the regular sets. Thus, the first schema might assert the person of a complement's object while the second asserts the number of a different structure, the second complement's subject. A local name can be used not only to shorten the specification but also to insure that all three requirements are imposed on the *same* structure:

```
(↑ COMPFNS* TERMFNS AGR)=%A
(%A NUMBER)=PL
(%A PERSON)=3
(%A GENDER)=FEM
```

The uncertainty may still be resolved by the choice of different strings, and this may result in a disjunction of different assignments for `%A`. However, as desired, all three requirements will be imposed simultaneously on each such assignment.

`%stem` [reference to stem]

`%stem` is a local name with a special, built-in meaning. When it appears in a lexical entry or in a template invoked by a lexical entry, it is taken to denote a symbol whose name is the headword of the entry. Thus, `%stem` denotes the symbol `walk` when it appears in the definition of `walk` or in a template that `walk` invokes. If the template `INTRANS` is defined as

```
INTRANS(P) = (↑ PRED)='P<(↑ SUBJ)>'
```

then the particular predicate must be specified in each invokation, as in `@(INTRANS WALK)`. An alternative is to define `INTRANS` with no arguments, as in

```
INTRANS = (↑ PRED)='%stem<(↑ SUBJ)>'
```

With this definition, the simpler invokation `@INTRANS` in the entry for `walk` will produce `(↑ PRED)='walk<(↑ SUBJ)>'` but `(↑ PRED)='fall<(↑ SUBJ)>'` in the entry for `fall`. A `%stem` reference can thus be used to eliminate redundant specifications in many lexical entries. However, as this example illustrates, the spelling of the `%stem` reference will be exactly identical to the spelling of the headword, including the same pattern of upper and lower casing.

p d [projection-value designators]

The correspondence ϕ between c-structure and f-structure was the only projection discussed by Kaplan and Bresnan (1982). Subsequent work, especially Kaplan (1987, 1989), Halvorsen and Kaplan (1988), and Kaplan et al. (1989), presents a more general view of abstract structures representing different aspects of lingusitic organization and of piecewise correspondences relating the elements of those structures. These piecewise correspondences permit abstract structures to be described in terms of the elements and relations of more concrete ones. GWB provides a simple notation for specifying the projection functions that map between different kinds of structures. The notation allows projection to be composed with each other, thus enabling the statement of *codescriptive* constraints.

In GWB a projection designator *p* is simply a character in the Greek alphabet portion of the Xerox Character Encoding Standard. Thus σ, δ, τ , etc. might all be used to denote different projection correspondences. Section 7 of Chapter I discusses the character encoding used internally by GWB and describes various methods for typing in projection designators. These include using Medley's Virtual Keyboards facility, invoking TEdit's abbreviation-expansion mechanism, or entering the ASCII name of the projection followed by a double colon.

If *d* is a designator of an element in one structure, then *p d* designates the element corresponding to *d* through the *p* projection. For example, if σ denotes the semantic projection, then σ↑ is the semantic structure corresponding to the ↑ f-structure, and σ(↑ `SUBJ`) is the semantic structure corresponding to ↑'s subject. If ρ denotes the correspondence between, say, units of semantic structure and a structure that represents their referents, then ρσ(↑ `SUBJ`) stands for the referent of the semantic structure corresponding to the `SUBJ` of ↑. Composition of projections is thus indicated by simple concatenation of projection designators, with a leftward projection applying to the element denoted by the designator to its right.

It is worth noting the difference between the conceptual principles embodied in LFG's projection architecture and the use of the distinctive attributes in other grammatical formalisms to represent different kinds of linguistic information in a single structure. Projections are aimed at separating and modularizing different aspects of linguistic organization, so that systems with relatively little interaction can be treated as different kinds of entities. Thus, apart from the explicit difference in notation, GWB displays different projections as separate structures in different windows. Also, for certain operations projections have behavior that is formally different from the values of ordinary attributes. Projection (but not attribute) inverses are used to define functional precedence and other predicates (see below and Zaenen & Kaplan, 1995), and generalization takes place over set-valued attributes but not projections, as described by Kaplan and Maxwell (1988b) and implemented in GWB.

* [c-structure metavariables]
M*

These designators can appear in schemata to denote, respectively, the node matching the associated category and the mother of that node. The f-structure metavariables ↓ and ↑ are thus exactly equivalent to the specifications ϕ* and ϕM*, where ϕ is the f-structure-to-c-structure correspondence. GWB will accept either notation but it will normalize to the abbreviatory arrows when any schemata are printed out. These node designators make it possible to put information structures other than f-structures into correspondence with c-structure nodes. These alternative projections may be kept separate when corresponding f-structures are identified, and they may be equated when the nodes' f-structures are distinct. For example, defining a μ projection to represent the morphosyntactic features of particular nodes can permit various morphological agreement requirements to be enforced without constraining the functional dependencies encoded in the f-structure. If English past-participles carry the schema (μM* `PARTICIPLE`)=`PAST`, the perfect auxiliary `have` can insure that its morphological complement is of the right type by the schema (μM* `MCOMP PARTICIPLE`)=c `PAST`. In μ-structure the features of the embedded verb would be grouped hierarchically under the `MCOMP` attribute even though the f-structure is arguably flat (by means of the schemata (µM* MCOMP)=µ* and ↑=↓ associated with the lower VP category).

4.2. Propositions

Propositions are elementary formulas in the functional description language that assert properties of f-structures and other projections. These propositions are true or false of a model consisting of a structure and an assignment of the variables in the proposition to the elements of that structure. The formulas involve a fixed set of predicates and relations as detailed below. These predicates and relations are asserted of arbitrary designators d_1, d_2, and d.

TRUE [truth values]
FALSE

These are the truth-value constants that provide a grounding for the logical system. They do not commonly appear in grammatical specifications, but they may arise in the course of grammatical analysis and show up in some of GWB's output displays. They may also be useful in the invocation of functional templates as described in Section 5.

$d_1 = d_2$ [equality]
d_1 =c d_2

The equality predicates are true if d_1 and d_2 denote exactly the same entity. The difference between the defining equality and the constraining equality =c is that the latter is true only if the equality is implied by all the defining propositions in the functional description. Technically, this is a slightly stronger condition than the one given in Kaplan and Bresnan (1982) because it requires that the constraining equality be true of all extensions of the minimal model, whereas Kaplan and Bresnan only imposed the constraint on the minimal model alone. The letter c (which may be in upper or lower case) must appear immediately next to the =, with no intervening white-space. If d_2 is a symbol designator, there must be white-space between it and the preceding c. Thus, (↑ SPECIES)=COW is interpreted as a defining equation involving the symbol COW. The proposition must be written as (↑ SPECIES)=C OW to obtain a constraining equality on the symbol OW.

$d_1 \in d_2$ [set membership]
$d_1 \in$c d_2

The membership predicate is true if d_2 denotes a set and d_1 denotes one of its elements. The suffix c again distinguishes between defining and constraining assertions. Techniques for typing the set-membership symbol are discussed in Chapter I, Section 7.

d [existential constraint]

A designator standing by itself is true if the designator has a denotation in the minimal structure satisfying the defining propositions in the f-description. It makes no sense to assert an existential constraint of a symbol or semantic-form designator, since these always have a denotation. GWB therefore issues an error message when it encounters such a constraint. This is usually symptomatic either of a conceptual confusion or of a notational mistake elsewhere in the f-description.

$d_1 < d_2$ [functional and abstract precedence constraints]
$d_1 > d_2$

These propositions assert abstract ordering relations between the entities denoted by d_1 and d_2. Based on the inverse of structural correspondences, they generalize to other projections the f-precedence relation $<_f$ discussed by Kaplan and Zaenen (1989a) and Zaenen and Kaplan (1995). If d_1 and d_2 denote f-structure elements, then $d_1 < d_2$ is true if and only if all the nodes that ϕ maps to d_1 precede in the c-structure all the nodes that ϕ maps to d_2. Kaplan and Zaenen express the f-precdence condition more formally in terms of the inverse of the structural correspondence and $<_c$, the usual precedence relation among c-structure nodes:

$$d_1 <_f d_2 \text{ iff for all } n_1 \in \phi^{-1}(d_1) \text{ and } n_2 \in \phi^{-1}(d_2),\ n_1 <_c n_2$$

If d_1 or d_2 involves a composition of other correspondences and thus denotes elements in other abstract structures, that combination of correspondences is similarly inverted to produce a set of nodes whose c-precedence can be tested. Thus GWB determines whether $\tau\sigma(\uparrow \texttt{SUBJ}) < \tau\sigma(\uparrow \texttt{OBJ})$ is true in the following way. It first finds the two elements of τ-structure that the two designators denote and identifies all the elements of σ-structure that map to those τ elements. It then finds all the f-structure units that map to those σ elements and all the nodes that map to those f-structure units. Finally, it compares the c-structure orderings of nodes in those two sets. Note that the same operators < and > are used with quite different effects both for the linear precedence predicate in c-structure rules and for abstract precedence in functional descriptions.

The second abstract precedence relation > is provided as a convenient alternative to <: the proposition $d_1 > d_2$ is equivalent to $d_2 < d_1$. Note that in GWB both of these are taken as constraining relations, not defining ones; in effect, they carry an implicit "c" constraint marker. They are applied only to structures and correspondences that already satisfy all the defining propositions of the f-description; GWB does not attempt to modify or extend structures or correspondences in order to make these assertions hold.

`#CAT[`*d categories*`]` [functional and abstract category constraints]

The `CAT` predicate is a formal extension to the published LFG notation that is also defined in terms of the inverses of structural correspondences. A `CAT` proposition is satisfied if the entity denoted by the designator *d* is a projection of at least one node whose category is in *categories*. If *d* denotes an f-structure, then

`#CAT[`*d categories*`]` iff there is some $n \in \phi^{-1}(d)$ such that category(n) $\in$ *categories*.

As with abstract precedence, if *d* involves a composition of other correspondences and thus denotes an element in some other abstract structure, that combination of correspondences is similarly inverted to produce a set of nodes whose categories can be tested.

The `CAT` predicate permits constraints to be imposed on the categories of the nodes that map to an f-structure without actually copying a category feature into the f-structure (or doing it surreptitiously as Kaplan and Bresnan (1982) did when they used grammatical-function names like `VCOMP`, `ACOMP`, and `PCOMP`). Asserting contraints by means of this correspondence-based predicate maintains the essential modularity of c-structure and f-structure properties while still allowing for limited and controlled interactions to be expressed.

The *d* argument is an ordinary designator (e.g. (↑ XCOMP) or σ↑). The *categories* argument is a collection of atomic category labels enclosed in set-brackets. For example, the proposition

```
#CAT[(↑ XCOMP) {NP' AP}]
```

might be included in the English lexical entry for `become` to indicate that the XCOMP of `become` must correspond to some node labeled either NP' or AP. This would disallow strings like "The girl became to go" and "The girl became in the park", even though the VP rule might allow VP', AP, NP', and PP all to be associated with the category-independent XCOMP function. If the XCOMP is assigned to the top of an ↑=↓ head chain, then the constraint can also be stated in terms of just the lexical category labels:

```
#CAT[(↑ XCOMP) {N A}]
```

If only a single category is specified as the *categories* argument, the set-brackets may be omitted. Thus the proposition

```
#CAT[(↑ XCOMP) V]
```

can appear in the lexical entry of verbs that take only verbal complements.

One advantage of a CAT specification over a VCOMP, ... NCOMP arrangement is that a predicate such as `consider` that allows any kind of complement requires no marking at all instead of being four ways ambiguous.

d_1 << d_2 [subsumption]
d_1 >> d_2

These propositions provide explicit grammatical access to the subsumption ordering of f-structures and other abstract structures. The subsumption ordering corresponds to a notion of information inclusion, in the sense that one structure (the subsum*er*) subsumes another if the second (the subsum*ee*) contains at least the information found in the first. Thus, d_1 << d_2 is true if and only if the structure denoted by the subsumee d_2 contains all the attributes of the subsumer d_1 structure, and on each of the common attributes, its value in d_1 is either equal to or subsumes its value in d_2. Subsumption provides for asymmetric propagation of information. If you assert that (↑ OBJ NUM)=1 and (↑ SUBJ)<<(↑ OBJ), this does not establish a value for (↑ SUBJ NUM): an existential constraint on the subject's number would not be satisfied. On the other hand, if you instead assert that (↑ SUBJ NUM)=1 along with the subsumption, the number value will be propagated "upwards" to the OBJ and would be inconsistent with any incompatible value independently asserted of the object's number. For convenience, d_1 >> d_2 is equivalent to d_2 << d_1.

Subsumption should be used with a degree of caution. When other propositions assert certain internal equalities in the subsumer, the f-description as a whole may be undecidable and GWB's sentence analysis procedure may not terminate. There have been proposals for restricted versions of subsumption that avoid the mathematical pitfalls, but the linguistic significance of these restrictions is not yet understood. Experience with the current relation in actual grammars may help to sharpen the issues and lead to theoretical refinements as well as improvements in GWB's implementation.

" *any string of characters* " [comment]

Just as in the regular predicate language, character strings enclosed in double-quotes are treated as invisible comments and assigned no other interpretation in the functional description language. They permit you to add explanatory notes at any position in a formula where a proposition is allowed.

4.3. Boolean combinations

The elementary propositions just described can be put together in Boolean combinations to make up the schemata that are attached to c-structure categories and appear in lexical entries. In some cases there are special rules of interpretation for the Boolean operators having to do with the defining/constraining distinction in LFG theory. The possible schemata are specified recursively as follows:

p [elementary propositions]

All of the elementary propositions described above can serve as schemata.

$s_1\ s_2 \ldots s_n$ [conjunction]

A sequence of schemata combined with no other operator (but separated by white-space as needed) is interpreted conjunctively. The sequence is true if and only if all of the individual schemata are true.

[*s*] [grouping]

Square brackets are used merely to explicitly mark that the components of *s* are to be treated as a unit with respect to other enclosing operators. This is especially useful in defining the scope of negation and in demarcating the values supplied for template substitution.

$\{ s_1 \mid s_2 \mid \ldots \mid s_n \}$ [disjunction]

A disjunction is satisfied if and only if at least one of the s_i is satisfied. If the s_i are defining as opposed to constraining schemata, GWB will attempt to construct separate minimal models for each of them, perhaps giving rise to ambiguous grammatical interpretations. If the s_i are constraining, then the disjunction will be true if at least one of the disjuncts is true of an independently defined structure; alternative interpretations will not be created.

By the usual rules of logic a disjunction would reduce to `TRUE` if one of its disjuncts is `TRUE`. This reduction does not go through in LFG, however, because of the defining/constraining distinction. Consider the formula `{(↑ TENSE)=PAST | TRUE}` which might be used to indicate that a sentence is or is not marked as past tense. If this is reduced to `TRUE` and hence ignored, the minimal f-structure solution will have no `TENSE` feature at all (assuming that `TENSE` is not elsewhere defined). But then that structure will fail to satisfy an existential constraint `(↑ TENSE)` occurring elsewhere in the f-description, which is not the intended result. Thus, GWB does not perform this reduction. Instead it operates on even `TRUE`-containing disjunctions, always attempting to find minimal solutions for each of the disjuncts taken separately.

{ s } [optionality]

This is true whether or not the s is satisfied. It is a short-hand equivalent for the disjunction { s | `TRUE` }, and as just explained, GWB attempts to find one minimal solution that satisfies an f-description containing s instead of the optionality and also one not containing s.

~s [negation]

A formula ~s is true of a model if s itself is false. Negation in LFG serves as a filter on the minimal structures satisfying the positive defining propositions in the f-description. GWB does not try to extend structures (for example, by adding conflicting features) so that a negative condition is satisfied. This means that certain conventional logical equivalences do not hold in LFG. It is thus not the case that ~[~s_1 ~s_2] characterizes the same structures as {s_1 | s_2}, as De Morgan's laws would suggest. With the first specification, the minimal model for all other propositions must satisfy one of s_1 or s_2. Solutions for the second specification may include two different extensions of that minimal model, one satisfying s_1 and one satisfying s_2. As another consequence, functional uncertainties in the scope of LFG's nonconstructive negation are decidable, whereas it is known that uncertainties in the scope of classical negation are not (Baader et al., 1991; Keller, 1991).

If s is an elementary proposition, the negation operator ~ may be placed as a prefix on the relation symbol: for example, `~(↑ SUBJ)=↓` may be written instead as `(↑ SUBJ)~=↓`. Also, note that the negation operator binds more tightly than conjunction: `~(↑ TENSE) (↑ CASE)` is interpreted as `[~(↑ TENSE)] (↑ CASE)` instead of `~[(↑ TENSE) (↑ CASE)]`.

5. Functional templates and lexical rules

One of the hallmarks of Lexical Functional Grammar is its use of lexical redundancy rules to express generalizations that hold across large families of lexical entries. Bresnan (1982c) discusses the motivation for a passive redundancy rule, and Kaplan and Bresnan (1982) present a simple formalism for encoding such relation-changing lexical rules. Lexical rules map lexical entries to lexical entries by either adding new schemata or by systematically modifying the designators appearing in existing schemata. The result is an entry that stands as a disjunctive alternative to the original. The passive rule, for example, applies to transitive verb entries. It introduces a schema insuring that the verb is in its past-participle form, it modifies existing schemata by converting object designators to subject designators, and it also changes existing subject designators, either converting them to oblique agent designators or else deleting them. Using GWB's notation for disjunction and the `NULL` symbol to indicate the absence of a grammatical function, the Kaplan/Bresnan passive rule would be written as

```
(↑ PARTICIPLE)=c PAST
(↑ OBJ)→(↑ SUBJ)
{ (↑ SUBJ)→(↑ OBL-AG) | (↑ SUBJ)→NULL }
```

The rewriting arrow → is the special operator that indicates how designators are to be systematically modified. Suppose this rule is applied to the functional annotation `(↑ PRED)='kick<(↑ SUBJ)(↑ OBJ>)'` for the simple transitive verb `kick`. The result would be the alternative passive entries

```
(↑ PARTICIPLE)=c PAST                     (↑ PARTICIPLE)=c PAST
(↑ PRED)='kick<(↑ OBL-AG)(↑ SUBJ)>'       (↑ PRED)='kick<NULL (↑ SUBJ)>'
```

Note the importance of rewriting the complex designator `(↑ SUBJ)` and not just the grammatical-function name `SUBJ` as the simpler `SUBJ→OBL-AG` would do. This would have the unintended effect of also changing the embedded `SUBJ` in a functional-control equation, so that the schema `(↑ XCOMP SUBJ)=(↑ OBJ)` for a raising verb would be transformed to `(↑ XCOMP OBL-AG)=(↑ SUBJ)`.

The early LFG literature was quite precise in its discussion of the effects of lexical redundancy rules, but it was much less clear about the circumstances that would trigger the application of those rules. The presence or absence of particular grammatical functions is one of the conditioning factors, but it is not sufficient. The verbs `cost` and `weigh` are well-known examples in English of words that take subjects and objects but do not passivize, and `donate` is a commonly used example of a ditransitive verb that does not undergo dative-shift. The existence of lexical exceptions to seemingly general syntactic variations was taken to support the claim that these phenomena were lexical in nature. However, it also called for some way of either explicitly marking exactly when particular rules can apply or predicting their application from other lexical properties.

The early literature assumed, mostly tacitly, that lexical items would be annotated with some set of morphosyntactic features, and that these, together with some default marking conventions, would govern the application of rules. A complete and formally coherent notation for this purpose was never proposed, and theoretical work in this area has shifted away from the marking approach. More recent work in Lexical Mapping Theory (Bresnan & Kanerva, 1989) has attempted to develop general principles for predicting lexico-syntactic variations from the semantic properties of the lexical predicates and a skeletal assignment of grammatical functions to particular semantic arguments. This is a much more interesting and promising approach, but it has not yet reached the stage where precise formal mechanisms have been defined and justified. Thus, in the absence of clear theoretical guidance, GWB provides its own abbreviatory conventions for expressing common f-description patterns that are shared by many lexical items and for marking lexical entries to indicate how they participate in those generalizations.

Redundancies in the f-description language are encoded by means of *functional templates*. These are like regular-predicate macros in that they permit the name assigned to a complex formula to be used in place of that formula in larger expressions. Repeating the example used in Section I.5, we know that all English common nouns have the general pattern of syntactic specifications illustrated by the following entries for `airport` and `girl`:

```
airport N S      (↑ PRED)='airport'
                 { (↑ NUM)=SG (↑ SPEC)
                  |(↑ NUM)=PL}.

girl N S         (↑ PRED)='girl'
                 { (↑ NUM)=SG (↑ SPEC)
                  |(↑ NUM)=PL}.
```

Each of these provides the appropriate 0-place semantic relation and also asserts that a singular noun must appear with a specifier elsewhere in the sentence. A parameterized functional template (defined in a Template Window) can express the common specification:

```
CN(P) = (↑ PRED)='P'
        { (↑ NUM)=SG (↑ SPEC)
         |(↑ NUM)=PL}.
```

The individual lexical entries for `airport`, `girl`, and all other common nouns can then be abbreviated to a simple invocation of the template:

```
airport N S     @(CN airport).
girl N S        @(CN girl).
```

Just like a regular-predicate macro, a template invocation is marked by an @, followed by a parenthetic expression that gives the name of the template and a sequence of values to be substituted for the formal parameters in the template definition. In this case the single parameter is `P`, and it is replaced by `airport` and `girl`, respectively, to produce the formulas in the original lexical entries. If all common nouns are marked in this way, they would all be affected by any change to the template definition. For example, the extended definition

```
CN(P) = (↑ PRED)='P'
        (σ↑ REL)=P
        { (↑ NUM)=SG (↑ SPEC)
         |(↑ NUM)=PL}.
```

with the added schema `(σ↑ REL)=P` would make all common nouns also assert `P` as the value of a relation attribute in the σ structure.

Template invocations thus serve as the notation in GWB by which individual lexical items are marked to indicate the lexical classes they belong to, and template definitions characterize the behavior of items in those classes. Transitive verbs might all be marked by the template invocation `@(TRANS v)`, where `v` is the name of the particular predicate, and intransitive verbs might be marked `@(INTRANS v)`, where these templates are defined as:

```
TRANS(P) =    (↑ PRED)='P<(↑ SUBJ) (↑ OBJ)>'.
INTRANS(P) =  (↑ PRED)='P<(↑ SUBJ)>'.
```

Many verbs would invoke just one of these macros, but verbs that can be either transitive or intransitive would invoke both of them disjunctively:

```
stop V S-ED     { @(TRANS stop) | @(INTRANS stop) }.
```

This illustrates the fact that template invocations may appear in the position of any proposition in the f-description language, so that the formulas they stand for can be combined by the usual Boolean operators.

The definition of a template can be any Boolean combination of schemata containing formal-parameter symbols in positions where schemata or symbols might otherwise appear. The definition can also include other template invocations. Thus, the fact that most transitive and ditransitive verbs can also be passivized can be encoded in the definitions

```
TRANS(P) =    @(PASS (↑ PRED)='P<(↑ SUBJ) (↑ OBJ)>').
DITRANS(P) =  @(PASS (↑ PRED)='P<(↑ SUBJ) (↑ OBJ) (↑ OBJ2)>').
```

The passive template can then be defined to specify the systematic rewriting of designators, in accordance with the original lexical redundancy rule described above. This might have the following form:

```
PASS(SCHEMATA) = {SCHEMATA | SCHEMATA (↑ PARTICIPLE)=c PAST
                                      (↑ OBJ)→(↑ SUBJ)
                                      { (↑ SUBJ)→(↑ OBL-AG)
                                       |(↑ SUBJ)→NULL}}.
```

The single parameter for this template can be a Boolean combination of schemata; in the case of `TRANS` and `DITRANS`, it is instantiated as a single `PRED` schema. The result after substituting this schema and expanding the template will be a disjunction with three branches. One disjunct will contain an isolated copy of the original schema, one will contain the agentive passive alternative, and the third will contain the agent-deletion passive. The passive alternations are created by rewriting designators according to the → specifications.

A *designator rewrite* is an expression of the form

match → *result*

where *match* and *result* are arbitrary designators. A designator rewrite can only appear in a position appropriate for a proposition and only in a template definition. A template definition containing such a specification is interpreted in the following way. First, the parameters of the template invocation are substituted for the formal parameters to give a parameter-free Boolean formula. The resulting formula is then converted to disjunctive normal form, a formula consisting of a top-level disjunction each of whose disjuncts is a conjunction of propositions and designator rewrites. Finally, the designator rewrites are removed from each disjunct and applied to the schemata remaining in the disjunct to systematically change any designators matching the *match* of a rewrite into the corresponding *result*. If the *match* for one rewrite is a subexpression of the *match* for another, the *result* corresponding to the larger *match* will be installed.

These three steps can be illustrated by the expansion of the template invocation `@(TRANS kick)`. The symbol `kick` is first substituted for `P` in the `TRANS` definition to give the formula

```
@(PASS (↑ PRED)='kick<(↑ SUBJ) (↑ OBJ)>')
```

Next, the `PRED` schema is substituted for `SCHEMATA` in the definition of `PASS`, producing

```
{ (↑ PRED)='kick<(↑ SUBJ) (↑ OBJ)>'
 |(↑ PRED)='kick<(↑ SUBJ) (↑ OBJ)>'
  (↑ PARTICIPLE)=c PAST
  (↑ OBJ)→(↑ SUBJ)
  { (↑ SUBJ)→(↑ OBL-AG)
   |(↑ SUBJ)→NULL}}
```

The inner disjunction is promoted to the top level to give the disjunctive normal form

```
{ (↑ PRED)='kick<(↑ SUBJ) (↑ OBJ)>'

 |(↑ PRED)='kick<(↑ SUBJ) (↑ OBJ)>'
  (↑ PARTICIPLE)=c PAST
  (↑ OBJ)→(↑ SUBJ)
  (↑ SUBJ)→(↑ OBL-AG)

 |(↑ PRED)='kick<(↑ SUBJ) (↑ OBJ)>'
  (↑ PARTICIPLE)=c PAST
  (↑ OBJ)→(↑ SUBJ)
  (↑ SUBJ)→NULL }
```

The designator rewrites are then executed within their disjunct scopes to produce the final result:

```
{ (↑ PRED)='kick<(↑ SUBJ) (↑ OBJ)>'

 |(↑ PRED)='kick<(↑ OBL-AG) (↑ SUBJ)>'
  (↑ PARTICIPLE)=c PAST

 |(↑ PRED)='kick<NULL (↑ SUBJ)>'
  (↑ PARTICIPLE)=c PAST }
```

The same process would be carried out in the expansion of the `DITRANS` template, with the result containing the appropriate three-place `PRED` schemata. A variant of the `DITRANS` template could be defined that would first invoke a `DATIVE` template to disjunctively rewrite the OBJ and OBJ2 designators before offering them to the PASS. The result would allow for all appropriate dative/passive combinations. Presumably verbs like give but not donate would invoke this dative variant.

Another aspect of template interpretation is illustrated by the specification of functional control verbs. The verb expect can take a simple object alone (*I expected Mary*), a that-complement alone (*I expected that Mary would go*), or an infinitival complement with or without a that-complement (*I expected to go*; *I expected Mary to go*). The complement complexities for verbs in this class can be isolated into a template RAISEOBJ, so that expect can be defined as

```
expect V S-ED   { @(TRANS expect) | @(RAISEOBJ expect) }.
```

The RAISEOBJ provides a basic PRED schema for the that-complement configuration, substituting this for the formal parameter of an FCONTROL template:

```
RAISEOBJ(P) =  @(FCONTROL (↑ PRED)='P<(↑ SUBJ) (↑ COMP)>').
```

The interesting specifications appear in the definition of the FCONTROL template. This template will result in a set of three alternatives, one with just the original schema COMP and the others specify either OBJ or SUBJ control of a XCOMP:

```
FCONTROL(SCHEMATA) = SCHEMATA {(↑ COMP)→(↑ XCOMP)
                               {  (↑ XCOMP SUBJ)=(↑ OBJ)
                                 |(↑ XCOMP SUBJ)=(↑ SUBJ)}}.
```

In this definition, instead of putting SCHEMATA in two branches of a disjunction as we did in the PASS template above, we use the equivalent but more succinct form of conjoining the original SCHEMATA to an optional specificaton of the infinitival variations. The result of @(RAISEOBJ expect) after substitution, DNF expansion, and designator rewriting is the three-way disjunction

```
{  (↑ PRED)='expect<(↑ SUBJ) (↑ COMP)>'
 |(↑ PRED)='expect<(↑ SUBJ) (↑ XCOMP)>'
  (↑ XCOMP SUBJ)=(↑ OBJ)
 |(↑ PRED)='expect<(↑ SUBJ) (↑ XCOMP)>'
  (↑ XCOMP SUBJ)=(↑ SUBJ) }
```

The middle object-control disjunct is not acceptable in its present form, since it calls for the governable OBJ function while at the same time asserting that the local PRED semantic form does not permit an object. To avoid such unintended incoherences, GWB carries out one further step in the template expansion process: If a disjunct contains a PRED-defining schema and a schema introducing a governable function not allowed by that semantic form, the semantic form is modified to permit the new function as a nonthematic grammatical relation. Thus, the PRED schema in the middle disjunction is changed to the more appropriate

```
(↑ PRED)='expect<(↑ SUBJ) (↑ XCOMP)>(↑ OBJ)'
```

Of course, RAISEOBJ can be composed with PASS to provide for the passive/raising occurrences of verbs like expect.

These examples have focussed on the use of templates to eliminate redundancies across lexical entries. But templates can also be used in the annotations on syntactic rules, as one way of factoring out collections of schemata that are common to several different constructions. In some cases it may even be advantageous for a template to be used in both a lexical entry and a grammatical rule. For example, suppose that a template BE has been defined to include the complex pattern of constraints that make up the definition of the verb `be`. These constraints express, among other things, how the various forms of `be` interact with participles to form passives and progressives. It has often been noted that reduced relative clauses and other constructions seem to honor these same constraints, and this is what motivated early proposals for transformations such as the "WH-IS" deletion rule that removes a deep-structure `be`. The shared constraints that such a transformation provided can be obtained in LFG simply by placing the invocation `@BE` in the c-structure rule that describes the surface pattern of reduced relative clauses.

To summarize, a functional template *T* is defined in a Template window by statements of either of the forms

$$T = s.$$
$$T\ (param_1\ param_2\ ...\ param_n) = s.$$

Where *s* is a Boolean combination of schemata and designator rewrites. The second form provides for some number *n* of formal parameters, where each $param_i$ is a symbol that presumably appears in *s* in the position of either a proposition or a designator. The ith realization at a particular invocation is systematically substituted for the symbol $param_i$ everywhere in *s*.

An invocation of a template *T* appears in the position of a proposition in some other lexical or in the annotation of a category in a grammatical rule. It is a term of either of the forms

$$@T$$
$$@(T\ real_1\ real_2\ ...\ real_n)$$

The first form is appropriate for a template with no realizations. The realization expressions $real_i$ in the second form are themselves arbitrary schemata and designators. When an invocation is expanded, those expressions are first substituted in place of the corresponding $param_i$ symbols in the definition of *T*. Second, if there are designator rewrites in the result of performing any necessary substitutions, it is converted to disjunctive normal form and the rewrites are executed. Third, any nonthematic function adjustments are carried out. The formula that emerges from this expansion process takes the place of the template invocation. As templates mutually invoke each other in complex patterns of composition, the possibility arises, just as for metacategories and macros, that a template will lead back to itself in a self-recursive cycle. A warning message will be printed in the Prompt Window if this situation is detected, and the expansion will be terminated by substituting the constant `FALSE` for the self-recursive invocation.

Usually a warning message will also be printed in the Prompt Window when a template invocation is expanded and the number of realization specifications is either greater than or less than the number of parameters specified in the template definition. This often indicates that a mistake has been made, but sometimes it is convenient to omit a realization on purpose. If realizations corresponding to some number of right-most parameters are not provided in an invocation, the template expansion is constructed as if all instances of those parameter names in the template body were simply not present. Just as with regular macro definitions (Section 2.1), you can attach a question-mark to a parameter name to indicate that you intend to omit the corresponding realization in some invocation, and this will cause the warning message to be suppressed. An illustration of the value of this feature is provided by the `RHO ENGLISH` configuration in the `DEMOENGLISH` file we discussed in Chapter II. The ρ projection in this configuration contains selectional features that predicates can use to impose restrictions on the types of their arguments. The predicate `devour`, for instance, might require that its first argument be animate, and certain nouns such as `Susan`, `dog`, and `girl` might be marked as such. The `ANIM` template is defined in `SELECTION ENGLISH` as

```
ANIM(ARG?) = (ρ↑ ARG ANIM)=+  @(PHYSOBJ ARG).
```

This marks the restriction structure with the positive `ANIM` feature and also expresses the fact that only physical objects can be animate. The lexical entry for `devour` contains the invocation `@(ANIM ARG1)` to indicate that its first argument must be animate, while the lexical entry for `kill` would specify that `@(ANIM ARG2)`. The nouns like `girl` that have animacy as an inherent feature can share the constraints specified in this definition simply by omitting the `ARG` specification in the template invocation. The parameter name `ARG` would be treated as a comment, so that `@ANIM` would in effect expand to

```
(ρ↑ ANIM)=+  @(PHYSOBJ)
```

This asserts that the selection structure corresponding to the f-structure containing `girl` must itself have the `ANIM` feature with value `+`. The question-mark indicates to GWB that this is an intended use of this template and no warning message is produced. Of course, the same effect can be achieved by means of two versions of the animacy template, one taking argument specification and one not; the question-mark convention allows statements about animacy to be localized in a single definition.

GWB's functional templates provide a well-defined and carefully implemented mechanism for capturing generalizations that would otherwise be distributed throughout the lexicon and grammar. Mutual invocation, parameter substitution, and designator rewriting can be combined in rich and powerful ways to represent a broad range of common patterns. Direct marking and Boolean specifications are perhaps not as elegant as the principles that might emerge from further theoretical efforts in LFG. But they do allow, given the current state of the art, for the precise control of lexical and syntactic behavior that explicit computational tests of large-scale grammars require.

6. Configuration components

A configuration identifies all the linguistic specifications that make up an active analysis environment and specifies other parameters that affect the behavior of the system when that environment is installed. The Sentence Input Windows and the LFG Window menus both have `Edit Config` items for creating a window for display and editing of configurations. They also have `Change Config` items for installing one of the configurations defined in GWB's internal database. You select the desired configuration from the menu of the available ones that pops up when you select `Change Config`.

A configuration has a version/language name just like any other linguistic specification. This name is used to identify the configuration for various system operations (installing it, saving it on a permanent file, etc.). The name appears at the top of the configuration editing window and is then followed by a sequence of component specifications each of which is a pair consisting of a component-name and a value terminated with a period. The component names are simple strings without punctuation or white space. Values can either be singletons or sequences, depending on the component. The general format is illustrated by the following example:

```
Configuration Window
SIGDEMO ENGLISH
 RULES (DEMO ENGLISH)
         (SIGDEMO ENGLISH)
         (XSIGDEMO ENGLISH).
 ROOTCAT ROOT.
 LEXENTRIES (DEMO ENGLISH)
              (DMORPH ENGLISH)
              (XSIGDEMO ENGLISH).
 TEMPLATES (DEMO ENGLISH)
             (SIGDEMO ENGLISH)
             (SELECTION ENGLISH)
             (XSIGDEMO ENGLISH).
 MORPHTABLES (DEMO ENGLISH).
 GOVERNABLERELATIONS  SUBJ OBJ OBJ2 COMP ?COMP OBL-?* POSS.
 SEMANTICFUNCTIONS  ADJ RELMOD TOPIC FOCUS.
 NONDISTRIBUTIVE  ADJ XADJ CONJ.
 PROJECTIONS (φ F- (HEAD TOPIC))
               (σ Semantic (REL TIME ARG1 ARG2 ARG3)).
 EPSILON e.
 PARAMETERS .
 INPUTKEYBOARD DEFAULT.
 LEXICALFONT DEFAULTFONT.
 TESTFILE .
----
```

The following list describes the components that make up a configuration and the way they affect system behavior when the configuration is installed. Some of the components have default values which GWB fills in as starting points when a new configuration is created.

RULES: A list of names that specifies the grammatical rules that are active in this configuration. The rules are installed in the order in which they are specified, so that the last name has priority over the names that precede it.

ROOTCAT: The default root category for the grammar. Defaults to S (for Sentence). Unless otherwise indicated, when a string is typed in, GWB will attempt to find a tree rooted in this category.

LEXENTRIES: A list of names that specifies the lexical entries that are active in this configuration. Like rules, lexical entries are installed in the order that they are specified, so that the last name has priority over all the names before it.

TEMPLATES: An ordered list naming the templates that will be active in this configuration.

MORPHTABLES: A list of names that specifies the morphological tables that are active in this configuration. A morphtable name can only have two parts: the version and the language (such as `(TEST ENGLISH)`). This means that there can only be one morphological table per version-language pair (although there can be more than one name specified in a configuration).

GOVERNABLERELATIONS: The attributes that are defined to be “governable” in the LFG sense: an f-structure will be marked as incoherent if it contains a value for a governable relation that is not subcategorized by the lcoal `PRED`. These attributes are specified as a list of elements, such as `SUBJ OBJ OBJ2 COMP`. The list can also contain patterns in the regular predicate notation, with the interpretation that all attributes that match a pattern are also governable. Unlike c-structure rules and expressions of functional uncertainty, the terms of these regular predicates are individual characters, and adjacent characters in the pattern are interpreted as matching a corresponding sequence in an attribute name. Also, the question mark sign (?) stands for any single letter, and the hyphen (-) is interpreted literally, not as an indicator of relative complementation. Thus, `?COMP` matches any attribute that has 5 letters ending in `COMP`, and `OBL-?*` matches any attribute that begins with `OBL-` and has zero or more characters following. If no value is specified for this component, then GWB will use the default value `SUBJ OBJ OBJ2 OBL-?* POSS COMP ?COMP`.

SEMANTICFUNCTIONS: A list of attributes that are not "governable" in the LFG sense yet are always associated with predicates. For instance, `ADJ`, `RELMOD`, `TOPIC`, and `FOCUS` might all be labeled semantic functions. If you declare an attribute to be "semantic", then GWB will mark an f-structure as incoherent if that attribute appears in an f-structure without a local `PRED`, and the f-structure value of such an attribute will be marked incomplete if it does not contain its own `PRED`. The default value for this component is `ADJ XADJ`. The semantic functions may also be specified in terms of regular-predicate patterns.

NONDISTRIBUTIVE: A list of attributes which, when asserted of a set, are not distributed across the elements of a set but rather are taken as properties of the set itself (see Section 4 of this chapter). Thus by registering `ADJ`, `XADJ`, and `CONJ` as nondistributive, their values will not be distributed across the individual conjuncts of a coordination construction. Instead, they will show up as attribute values of the set itself, which will thus have a mix of f-structure and set properties. Regular-predicate patterns can also be used to specify the nondistributive function.

PROJECTIONS: Projections are denoted in functional schemata by characters in the Greek alphabet (perhaps entered using the Greek virtual keyboard). The `PROJECTIONS` component of a configuration enables you to declare the projections you expect to use and to associate descriptive labels with the projection symbols that will appear in window titles and other system messages. The value of this component is a set of projection-descriptors, one for each projection. A projection-descriptor is a parethesized list of the form (projection-symbol label attribute-ordering), such as

```
(σ Semantic (REL ARG1 ARG2 ARG3))
```

The attribute-ordering is an optional parenthesized list of attribute names that determines the order in which attributes will be presented when a structure in that projection is displayed. Thus, this declaration indicates that the σ projection is labeled as the `Semantic` projection and that the attributes `REL`, `ARG1`, `ARG2`, and `ARG3` will be presented in that order. Other attributes not specified in the attribute-ordering will be ordered according to the abstract precedence relation implicitly defined for values in that projection. As discussed for example by Kaplan and Zaenen (1989a) and Zaenen and Kaplan (1995), this is determined by inverting all the correspondences that lead to that projection from the concretely ordered c-structure.

Your grammar, lexical entries, and templates may contain schemata that refer to projections not declared in the configuration. Those schemata will simply be ignored in any grammatical processing. This makes it convenient to share entries and templates from a grammar with a more elaborate projection configuration than the one you are working on, or to exclude consideration of some projections while you focus on others.

EPSILON: The symbol in c-structure rules and expressions of functional uncertainty that GWB interprets as the empty string (epsilon). The default epsilon symbol is e.

PARAMETERS: This optional component provides a general escape-hatch for setting arbitrary Lisp variables that you might want to take on specific values only when a particular configuration is active. The value of this component is a list of variable-value pairs, each of which is enclosed in parentheses. The variable in each pair will be set to the specified value when the configuration is installed, and it will be restored to its pre-existing default value when the configuration is deactivated. You can modify the default value of a variable by means of the usual Lisp mechanisms, for example, by setting its value with a `SETQ` in your Medley initialization file or by entering a `SETQ` expression in an Interlisp Executive Window. The default value will be in force for all configurations that do not override it through an explicit parameter specification.

To illustrate, the Lisp variables `CSTRUCTUREMOTHERD` and `CSTRUCTUREPERSONALD` (in the Interlisp package) control the way the c-structure trees are layed out. The mother variable determines the vertical distance between nodes in the tree and the personal variable controls the minimal horizontal space surrounding each node. The distance in each case is determined by multiplying these variables by the height of the font used for c-structure nonterminal node labels. Initially, these variables are set to 1.5 and .4, respectively. If the rules selected by a particular configuration tend to produce long non-branching vertical chains, the trees will be more compact if these values are changed. This can be accomplished by including the line

```
PARAMETERS  (CSTRUCTUREMOTHERD 1.0) (CSTRUCTUREPERSONALD .3).
```

in the configuration. If you prefer more compact trees in every configuration, you can instead change the default values by entering the expressions

```
(SETQ CSTRUCTUREMOTHERD 1.0)
(SETQ CSTRUCTUREPERSONALD .3)
```

in either your personal initialization file or in an Executive Window.

Here is a list of other parameters that affect the system's behavior:

`CSTRUCTUREFONT`: This defines the font or fontclass that is used for the labels of nonterminal c-structure nodes. It is initalized to the fixed-pitched `TERMINAL` font. The `LEXICALFONT` is always used for terminal node labels.

`ALLOWEMPTYNODES`: As discussed in Section 2 of this Chapter, the empty string will not be subtracted from c-structure regular predicates if this parameter is explicitly set to `TRUE`.

`PRESERVEEPSILONS`: Also as discussed in Section 2, explicit epsilon symbols in c-structure rules will be preserved and appear as distinct tree nodes if this parameter is set to `TRUE`.

`ASCIIONLY`: Normally, special operators and projection symbols in the rule and f-description languages will be printed in edit windows and on permanent files using their (possibly 16-bit) encodings in the Xerox Character Encoding standard. This may make it difficult to apply other text editors to grammar and lexicon files, to send them in mail messages, or to load them into other processing systems. If the parameter `ASCIIONLY` is set to `TRUE`, these symbols will be printed using characters drawn only from the standard 7-bit ASCII character set. In particular, they will be rendered as the ASCII characters, described in Section 7 of Chapter I, by which they can be entered into the GWB editing windows. Files created in this mode can still be loaded back into GWB.

SYMBOLSINLINE: This parameter affects the display of f-structures. Symbol-valued attributes usually appear on separate lines in an f-structure display, just like features with more complex values. F-structures with many simple grammatical features (case, number, person) can thus become quite large vertically. The integer-valued SYMBOLSINLINE parameter can be used to reduce the vertical size of the display. If the number of symbol-valued features in an f-structure is greater than SYMBOLSINLINE, those features will be displayed on a single line separated by commas. The default value of SYMBOLSINLINE is 4.

INPUTKEYBOARD: The keyboard used in the input window for typing in new sentences (see Chapter I, Section 7). This must be the name of a Virtual Keyboard listed when the Keyboards menu item is selected. If you specify a particular keyboard as the input keyboard, then you can toggle back and forth between it and the usual LFG keyboard with CTRL–K.

LEXICALFONT: The font used to print lexical items wherever they appear. The lexical font is installed in the Sentence Input window when the configuration is activated, it is use to print lexical items in Lexicon editing windows, and it is used to display the terminal nodes in c-structure trees. If a lexical font is not specified, Medley's current DEFAULTFONT will be used. The Classic font family is the one with the most complete collection of Xerox Character Encoding characters (see Chapter I, Section 7) and will probably give the best results for Western European, Greek, and Cyrillic alphabets.

TESTFILE: During the course of a session, the Sentence Input window fills up with strings that have been entered and the summary statistics of their analysis. Whenever GWB recognizes that you are leaving a configuration, you will be asked whether you want to save the current contents of the Sentence Input window on a file whose name is provided as the value of the TESTFILE component. Similarly, whenever you install a configuration, the Sentence Input window will be initialized to the previously saved contents of this file. In this way, the GWB helps you maintain a permanent record of the strings you have analyzed, allowing you easily to reanalyze those sentence as your grammar and lexical entries evolve.

Chapter IV

Windows and Menus

This chapter takes you on a tour of the Grammar Writer's Workbench window by window and menu by menu. It is intended to serve as a reference for the different windows used by the GWB. As such, it may repeat some of the material encountered in previous chapters, but that information may be easier to find here and is presented with a little more detail.

Most of the windows in the system fall into one of two groups: edit windows and display windows. There is a different type of edit window for each of the following: rules, lexical entries, configurations, templates, and morphology tables. All of the edit windows are TEdit windows with additional menus and commands. If you click in the title bar of an edit window with the middle button you will get the standard TEdit menu. If you click with the left button you will get a special LFG menu specific to that type of edit window. Many of the menu items appear in more than one menu, so that there are only a dozen or so distinct menu items. Online documentation for each menu item can be obtained by sliding the mouse over the menu item (so that it inverts) and then waiting for a few seconds. The documentation will be printed in the prompt window at the upper left of your screen.

Many of the menu items do nothing more than create other menus of things to edit or invoke. When this is the case, there are several options for determining the system's behavior. If the `SHIFT` key is held down when the menu is created, then the menu will be attached to a window of the appropriate type, most likely the window that the menu came from. If the `CTRL` key is held down when the menu is created, then a fixed but unattached menu will be created, and you will be prompted to position the menu where you want it on the screen. If no key is held down, then a pop-up menu is created that will disappear once something is selected or the selection is aborted by clicking outside of the menu.

Sometimes a menu item that only creates menus will have a sub-menu of version/language names to its right. This allows you to choose the version/language that you want to draw from for the items in the menu to be created. If no language version is chosen, then a menu will be created of the items that are currently active.

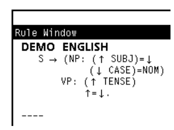
Whenever you select an item to edit, you also have the option of reusing an existing window or opening a new one. If you select the item with the middle button held down, then a new edit window will be created. If you select the item with the left mouse button held down, then an existing window will be reused, if possible. First priority is given to the window that the menu is attached to, if it is available. If there are no edit windows of this type, or if all of the windows have modified but uninstalled items, then you will be prompted to indicate the location of a new edit window.

Another way to obtain an item to edit is by "visiting" it. To visit an item, type `CTRL-V` in an edit window, and then type the name of the item. The name can either be a short name (e.g. `NP`) or a fully qualified name (e.g. `DEMO ENGLISH NP VP`). If you give just the prefix of a fully qualified name (e.g. `(DEMO ENGLISH)`), then the system will fetch all of the items with that prefix.

You can also obtain an item by "showing" it. If you type `CTRL-S` when the cursor is blinking on a particular designator in an editing window or in the Sentence Input Window, edit windows will be opened for all currently active items with the indicated name. Thus, if you are pointing at something that is both a c-structure category and also the name of an f-structure template, both a rule-editing window and template-editing window will be opened. If you point at a word in the Sentence Input Window and type `CTRL-S`, the current lexical entry for that word will be shown in a lexicon window.

No matter how you obtained an item, when you are done editing it, you can install your changes by typing `CTRL-X.` You can cancel your changes by typing `CTRL-C`. This will not change the content of the edit window, but will clear the "dirty" bit so that the window can be reused to edit other items.

The display windows for the c-structure, chart, and various projections use fixed menus attached to the top of the window. These windows also let you obtain more information about the items that are displayed in the window by clicking on the displayed objects with various combinations of `SHIFT` and `CTRL`.



1. Rule windows

A rule window lets you edit grammatical rules. Click in the title bar of a rule window with the left or middle mouse button to obtain its menu. The menu has the following items:

Edit Rule — Creates a menu of rules to edit. A sub-menu to the right allows you to choose which language the rules are drawn from. If no language is specified, then a menu of the active rules is created.

Find Category — Creates a menu of terminal and non-terminal categories to search for. The categories are determined by scanning the right-hand sides of all relevant rules, the ones that are currently active or the rules in a particular version/language if you slide off to the right. When a category in the menu is selected, then all of the rules that include that category on their right hand side are displayed.

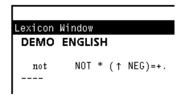
Find Attribute — Creates a menu of attributes to search for (e.g. `SUBJ`, `NUM`, etc.). The attributes are determined by scanning the right-hand sides of all relevant rules. When an attribute in the menu is selected, then all of the rules that use that attribute in their right hand side schemata are displayed.

Find Symbol — Creates a menu of symbols that are used as feature values (e.g. `SG`, `NOM`, etc.). The symbols are determined by scanning the right-hand sides of all relevant rules. When a symbol in the menu is selected, then all of the rules that use that symbol are displayed.

Find Template — Creates a menu of templates to search for. The templates are determined by scanning the right-hand sides of all relevant rules. When a template in the menu is selected, then all of the rules that invoke that template are displayed.

Edit Template — Creates a menu of templates to edit, as described in Section 3.

Tedit Menu — Brings up the normal TEdit text-editing menu containing items for putting and getting files and for formatting text.



2. Lexicon windows

A lexicon window lets you edit lexical entries. Typing CTRL-K in this window toggles the keyboard between the lexical keyboard of your active configuration (good for typing in the head words of the entries) and the standard LFG keyboard (good for typing in the categories and schemata). Click in the title bar of a lexicon window with the left or middle mouse button to obtain its menu. As described in Section I.3, you can define a lexical entry by typing in the full text of its definition, by retrieving another entry as the start of a sequence of edits, or by specifying that the new word is "like" an old one to get automatic assistance in constructing the new definition. A lexicon window menu has the following items:

Edit Lex — Creates a menu of lexical entries to edit. The menu is hierarchical, with the top level having one entry for each letter that a word in the lexicon begins with, and the sub-menus either having one entry for each letter at that level or all of the remaining words, if the number is small. If the SHIFT key is held down, then the menu is attached on the right side of a lexicon window.

Find Category — Creates a menu of terminal categories to search for. The categories are determined by scanning the lexical entries that are currently active. When a category in the menu is selected, then all of the lexical entries that have that category are displayed.

Find Attribute — Creates a menu of attributes to search for (e.g. SUBJ, NUM, etc.). The attributes are determined by scanning the definitions of the relevant lexical entries (currently active or in a version selected by sliding to the right). When an attribute in the menu is selected, then all of the lexical entries that use that attribute in their definition are displayed.

Find Symbol — Creates a menu of symbols that are used as feature values (e.g. SG, NOM, etc.). The symbols are determined by scanning the relevant lexical entries. When a symbol in the menu is selected, then all of the lexical entries that use that symbol are displayed.

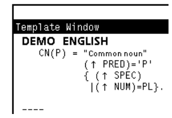
Find Template — Creates a menu of templates to search for. The templates are determined by scanning the definitions of the relevant lexical entries. When a template in the menu is selected, then all of the lexical entries that use that template in their definition are displayed.

Find Morphcodes — Creates a menu of words that serve as a root or affix in the morphological analysis of other irregular forms. When a word in the menu is selected, then all of the lexical entries are displayed whose definitions somehow depend on the selected root or affix.

Find Root — Creates a menu of morphology codes to search for. The morphology codes are determined by scanning the definitions of the relevant lexical entries. When a code in the menu is selected, then all of the lexical entries that use that code in their definition are displayed.

Edit Template — Creates a menu of templates to edit. See section 3 for more detail.

Tedit Menu — Brings up the normal TEdit text-editing menu.



3. Template windows

A template window lets you edit templates. Click in the title bar of a template window with the left or middle mouse button to obtain its menu. The menu has the following items:

Edit Template — Creates a menu of templates to edit. A sub-menu to the right allows you to choose which language the templates are drawn from. If no language is specified, then a menu of the active templates is created.

Template Lattice — Creates a menu of the currently active templates so that you can select the focus for a graphical display of the hierarchy of active template references. If you select `All`, you will see a lattice showing references among all the currently active templates. As illustrated in Section 1.5, templates to the right invoke (and thus inherit information) from templates they are linked to that appear further to the left. There is a separate node in this graph for each reference to a given template; this avoids the complexity and clutter of lines that would otherwise connect to a single occurrence representing that template. The separate nodes for a multiply-referenced template have a dotted border to indicate that other nodes representing that template appear elsewhere in the graph.

If you select a particular template from the menu, the display will show the reference hierarchy from the selected template's point of view. Again, templates shown on the left are referenced by templates shown further to the right. The selected template has a solid border to indicate that it is the focus. A graph focussing on one template is simpler than the `All` lattice so that it is useful to show linking lines to a single node representing a multiply-referenced item. Thus, by focussing on a template that the `All` lattice shows with a dotted border, you will see a smaller graph with all the details of that template's relationships made explicit.

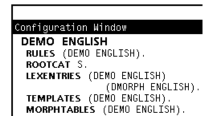
The nodes in a template lattice are mouse-sensitive: clicking on a node will display a new lattice with the new selected node taken as the focus. The new graph will appear in a new window if you are selecting from the `All` graph or if you click with the middle button. Otherwise, the new graph will replace the old one. Following the conventions of the c-structure display, if you click while holding down the `CTRL` key, a window for editing the selected template will appear.

Find Attribute — Creates a menu of attributes to search for (e.g. `SUBJ`, `NUM`, etc.). The attributes are determined by scanning the definitions of the relevant templates (currently active or in a version selected by sliding to the right). When an attribute in the menu is selected, then all of the templates that use that attribute are displayed.

Find Symbol — Creates a menu of symbols that are used as feature values (e.g. `SG`, `NOM`, etc.). The symbols are determined by scanning the relevant templates. When a symbol in the menu is selected, then all of the templates that use that symbol are displayed.

Find Template — Creates a menu of templates that are referenced by other templates. The templates are determined by scanning the definitions of all relevant templates. When a template in the menu is selected, then all of the templates that use that template are displayed.

Tedit Menu — Brings up the normal TEdit text-editing menu.



4. Configuration windows

A configuration window lets you edit a configuration. Click in the title bar of a configuration window with the left or middle mouse button to obtain its menu. The menu has the following items:

Edit Config — Opens an edit window on a configuration. A sub-menu to the right allows you to choose which configuration. If none is chosen, it defaults to the active configuration.

Change Config — Creates a menu of configurations. If one of the configurations in the menu is selected, then that configuration is installed as the active one.

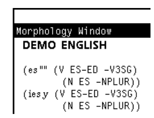
Parse Cat — Creates a menu of possible parse categories from the set of active rules. Selecting a category makes it become the default parse category.

Edit Rule — Creates a menu of grammatical rules to edit. See section 1 for more detail.

Edit Lex — Creates a menu of lexical entries to edit. See section 2 for more detail.

Edit Template — Creates a menu of templates to edit. See section 3 for more detail.

Tedit Menu — Brings up the normal TEdit text-editing menu.

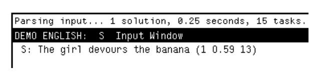


5. Morphology windows

A morphology window lets you edit a morphology table. Click in the title bar of a morphology window with the left mouse button to obtain its menu. The menu has only one item:

Edit Morph — Opens an edit window on a morphology table. A sub-menu to the right allows you to choose which morphology table. If none is chosen, it defaults to the active morphology table.

Tedit Menu — Brings up the normal TEdit text-editing menu.

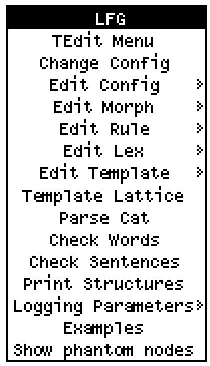


6. The Sentence Input window

The Sentence Input window lets you enter sentences to be parsed. The title bar displays the name of the current configuration followed by the default parse category. If a string is typed in followed by a `CTRL-X`, the string will be parsed and the results displayed in the display windows. In addition, the system will display the number of solutions, the parse time, and the number of tasks in the message area above the title bar and will optionally append this information after the sentence, depending on the setting of the logging parameter (see *Logging Parameters* below). If you prepend the input with a category followed by a colon (e.g. `NP:`), then that becomes the parse category for just this input. To remind yourself of your own grammatical intuitions, you may also prefix an input string with * or ?; those marks will be stripped off before the string is analyzed.

In some situations the analysis of the input sentence cannot be carried out. This is the case when the input contains words that are undefined and cannot be morphologically analyzed or when the grammar or lexicon refers to undefined templates, macros, or meta-categories. In any of these cases, an appropriate message will be displayed in the prompt window attached to the Sentence Input Window, followed by the message `[aborted]`. If you wish to pursue the analysis, you must define the missing items in an appropriate editing window. GWB makes this process a little bit easier when the missing items are unknown lexical entries: If you type `CTRL-L` when the caret is in the input window, a lexicon-editing window will be activated and initialized to contain the set of missing words from the last attempted analysis. It is possible but, we hope, unlikely that you will see an `[aborted]` message without an explanatory comment. This is a sign of an error in GWB's internal algorithms; please inform us of the situation in which this arises.

This is the input window's title-bar menu:



The items have the following interpretations:

Tedit Menu — Brings up the normal TEdit text-editing menu.

Change Config — Displays a menu of configurations. If one of the configurations in the menu is selected, then that configuration is installed as the active one.

Edit Config — Opens an edit window on a configuration. See Section 4 for more detail.

Edit Morph — Opens an edit window on a morphology table.

Edit Rule — Creates a menu of rules to edit. See Section 1 for more detail.

Edit Lex — Creates a menu of words to edit. See Section 2 for more detail.

Edit Template — Creates a menu of templates to edit. See Section 3 for more detail.

Template Lattice — Creates a menu of currently active templates so that you can select the focus for a graphical display of the hierarchy of template references. See Section 3 for more detail.

Parse Cat — Displays a menu of possible parse categories. Selecting a category makes it become the default parse category.

Check Words — Applies the tokenizing, morphological analysis, and lexical look up procedures to the words in the strings selected in the Sentence Input Window and determines which words are not recognized by the current morphology and lexicon. When all of the strings have been checked, a summary of the results is printed in the prompt window attached to the Sentence Input Window. This summary indicates how many strings were checked, how many had at least one unknown word, and how many different unknown words were encountered in all the strings. Typing `CTRL-L` while the cursor is in the Sentence Input Window will cause those unknown words to appear in a Lexicon Editing Window. *Check Words* is useful in prescanning a list of sentences in order to build up a complete set of lexical entries prior to syntactic analysis.

Check Sentences — Parses all of the strings that are selected, and checks that the number of solutions matches the number already present, if any, after each string. Strings can be selected by clicking the left button of the mouse anywhere in the first sentence and then clicking the right button anywhere in the last sentence (you can scroll in between, as long as you don't click the mouse anywhere else). When all of the sentences have been checked, a TEdit window opens with the results in it. Each sentence will have its new parse result appended at the end. A parse result is in the form of a triple:

(number-of-solutions number-of-seconds number-of-tasks)

If there is a mismatch between the old number of solutions and the new, then `MISMATCH` is printed after the sentence. At the end is a summary of the total number of mismatches plus some performance information. The sentences with their parse results can be copied into the input window and tested again, if so desired. Lines in the selection that do not begin with a parse category (a category followed by a colon) are skipped over. This permits you to freely intersperse comments among your test sentences. If a particular string contains unknown words, that fact is also indicated in the results window and no further processing is carried out on that string. A list of all the unknown words encountered across all sentences is appended at the end of the window.

Print Structures — This causes strings in the current selection that are prefixed by a parse category to be analyzed, like *Check Sentences*. In this case, however, the grammatical structures are printed to a file and in a format that can serve as input for other programs that you might write to carry out addditional phases of a language-processing application. When you select the *Print Structures* items, you will be asked to supply the name of the output file. As the analysis moves from sentence to sentence, progress is reported in the prompt region attached to the Sentence Input Window.

The first line of the output file records the current time and date. The rest of the file consists of one record, enclosed in square brackets, for each analyzed sentence. A sentence record has the following format:

["*sentence*" *result* (*analysis* $_1$ *analysis* $_2$ ...)]

The words of the sentence are enclosed in double-quotes, and they are followed by a parenthesized result-triple. This is followed by a sequence of subrecords, one for each analysis of the sentence. Each analysis is a parenthesized list of the form

(*c-structure* *proj* $_1$ *proj* $_2$ ...)

The c-structure is a labeled bracketing where each constituent is represented by a parenthesized list beginning with its category label and followed by its daughter constituents. A colon is used to attach a node label and a node number, and parentheses at the leaves of the tree are omitted. Thus, the tree

```
(NP:20 (DET:2 the) (N:3 girl))
```

would appear as `(NP:20 (DET:2 the)(N:3 girl))`. Each of the *proj* $_i$ represents a different projection from the c-structure; typically the f-structure will be the only one, but some grammars may use the * and M* notations to define other c-structure projections. A projection record is a parenthesized list of the form

(*projname* (*ptop* *p* $_1$ *p* $_2$...) *proj* $_1$ *proj* $_2$ ...)

The projection name is the ASCII representation of the projection, for example, `f` for the f-structure, as described in Section I.7.3. The *ptop* is the unit of the projection that corresponds to the top-most unit of the structure (node or prior projection) that it corresponds to. The *p* $_1$ *p* $_2$ are additional units of this projection, if any, that are not accessible from the top-most one.

Finally, the list of projection units is followed by a sequence of other records for further projections (for example, a semantic projection from the f-structure).

A particular attribute-value structure is printed as a list of ordered pairs, with each pair enclosed in parentheses. A special pseudo-attribute `VARS` can appear at the beginning of an attribute-value structure. Its value is a list of integers: positive integers indicate which node or less abstract entity this structure corresponds to, while negative integers indicate coreference or token-identities within this particular projection.

Logging Parameters — The sub-menu at the right of this menu item lets you choose how you want parse results to be logged in the input window. A parse result is a triple of the form

(number-of-solutions number-of-seconds number-of-tasks)

The sub-menu has the following items:

> *Log All Results* — Every time you parse a sentence, append a new parse result after all of the preexisting parse results.
>
> *Log First Result* — Only log the parse result if one is not already there. (This is the default behavior of the system.)
>
> *Log Last Result* — Replace all of the existing parse results with the most recent one.
>
> *Don't Log Anything* — Turn off logging.

Examples — Gives access to a small number of rules, lexical entries, or morphology table entries that illustrate some aspects of the notation.

Show phantom nodes — Normally, when an ordinary c-structure rule is referenced as a macro invocation in a referring rule, it is treated as a meta-category so that the c-structure contains no distinct subree corresponding to it. Instead, as noted in Section III.3.3, its right-side expansion is merged into the referring rule's expansion. Although this produces the intended configurations of c-structure nodes, it may increase the difficulty of grammar debugging, since the grammar specification responsible for a given arrangement is not explicitly marked in the tree. Clicking *Show phantom nodes* toggles an internal switch and causes these phantom-node invocations to be treated as if they were ordinary rule references. This means that the corresponding nodes *will* appear explicitly in the tree. At the same time, the menu item changes to *Hide phantom nodes*, indicating that another click will cause the system to revert to its normal behavior. In principle this switch should only affect the display of the tree and otherwise have no impact on the number or kinds of analyses that are produced. This is not always the case, however, because the restriction against nonbranching dominance chains may have different consequences depending on whether or not the node is a phantom or not. Thus, this feature should be used with caution: it is often helpful in debugging but sometimes can lead to confusing results.

7. The LFG logo window

In the upper right corner of your screen there is a small window with LFG printed in large letters. This is the logo for the Grammar Writer's Workbench, and in addition to being decorative, it provides a root menu for the system. The menu appears if you click in the window with either the left or the middle mouse button. If you accidentally close the logo window, a menu item is added to the system background menu that lets you set up the LFG windows again. (The background menu can be obtained by clicking in the background with the right mouse button.) Here is a discription of the items in the root menu:

Set Up LFG Windows — Kills all of the current LFG windows and then recreates them in their canonical positions. This is useful in cleaning up the screen when it has gotten into a funny state (for example, if Unix or DOS operating system messages have overwritten parts of the Medley screen).

Set Up Input Window — Recreates just the input window.

Kill LFG Windows — Clears the screen of LFG windows, adding to the system background menu an item that can be selected at a later time to recreate the LFG windows .

LFG Manual — This menu item appears if the files for this manual exist in your file system on a directory whose name is the value of the Lisp variable `IL:LFGMANUALDIR` . Selecting this item creates a menu of files in the GWB manual for browing with TEdit. The Contents entry gives a mapping between the contents of the manual and the files they are stored on. If you hold down the shift key when selecting this, then the menu is attached to the left of the logo window.

File Manager — Invokes the File Manager (see Chapter I, Section 6). The sub-menu allows you to choose between `CLEANUP`, `FILES?` and `LOAD`. The default is `CLEANUP`.

Change Config — Creates a menu of configurations. If one of the configurations in the menu is selected, then that configuration is installed as the active one.

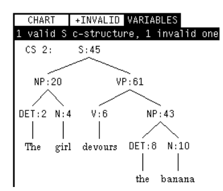
Edit Rule — Creates a menu of rules to edit. See section 1 for more detail.

Edit Lex — Creates a menu of words to edit. See section 2 for more detail.

Edit Template — Creates a menu of templates to edit. See section 3 for more detail.

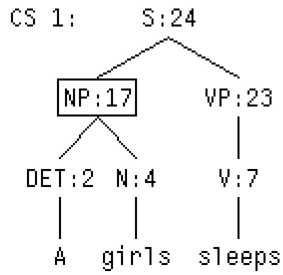
Edit Config — Opens an edit window on a configuration. See section 4 for more detail.

Edit Morph — Opens an edit window on a morphology table.



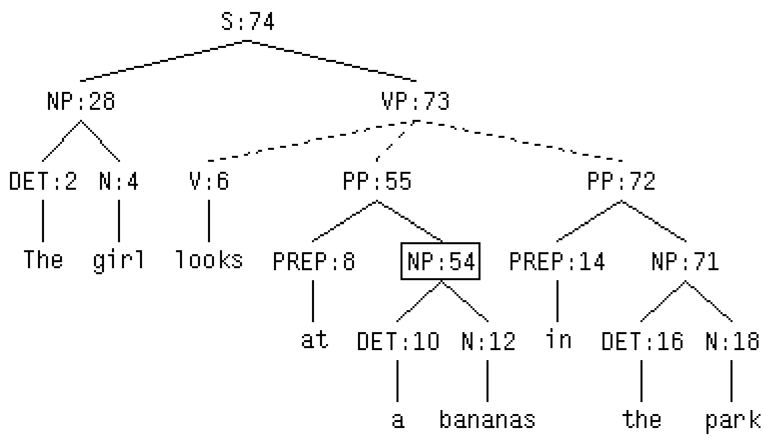
8. The C-structure window

Whenever a sentence is parsed, the c-structures are displayed in the c-structure window. The title bar shows the total number of valid and invalid c-structures, plus the number actually displayed. If the trees do not fit in the window, more of them can be made visible by vertical or horizontal scrolling. A vertical scrollbar appears to the left of the window when you move the mouse slowly past the left edge. A horizontal scrollbar appears at the bottom of the window when you move the mouse slowly past the bottom edge. Invalid c-structures are only displayed if the `+INVALID` button is on; this is turned on automatically if no valid trees can be found. Invalid trees are distinguished from valid trees by the fact that at least one node in each invalid tree is boxed:



The boxing indicates the lowest point in the tree where the functional constraints could no longer be satisfied (e.g., node 17 above).

Sometimes a node in a tree will be linked to its daughters with dashed lines instead of solid lines:



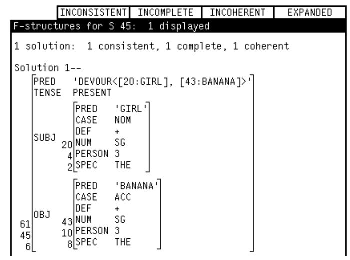
The dashed lines indicate that the VP node can expand to more than one subtree. GWB has chosen one of these subtrees to display, keeping the others in reserve until you specifically request that the alternatives be presented. You can ask for the full set of expansions at that node by clicking with left or middle where the dashed lines are, in the region between the node and its daughters. The window will be redrawn, with the dashed lines for that node replaced by solid lines, and with trees containing the alternative expansions of that node added to the display. A tree may have more than one node with dashed lines, so you may have to make more than one selection in order to see the complete set of trees. This display strategy is intended to avoid situations where the window is cluttered with an enormous number of trees that do not provide much useful information.

Additional properties of the nodes in the c-structure can be obtained by clicking the mouse on the ones that you are interested in. If you click on a node with the left button, then all of the constraints associated with that node are displayed in the description window to the left of the f-structure window. If you click on a node with the middle button, then all of the f-structures associated with that node are displayed in the f-structure window. If you hold down the `CTRL` key while clicking, then the rule associated with that node is displayed in a rule window. If you click on a word at the bottom of the tree, then an edit window is opened on its definition.

The `NODE NUMBERS` button determines whether or not the identifying numbers are displayed with the nodes. These numbers are also used in the f-structure window and other projection windows; they let you correlate c-structure nodes with their corresponding f-structures.

The `CHART` button determines whether or not the chart is displayed. See section 12 for more details on the chart.

The horizontal/vertical layout of the tree displays is controlled by three parameters, CSTRUCTUREMOTHERD, CSTRUCTUREPERSONALD, and CSTRUCTUREFONT. These are described in Chapter III, Section 6. They can be set as parameters in a particular configuration, for example, if a given grammar generates trees of an unusual geometry. As with other configuration parameters, global defaults can also be changed by setting the underlying Lisp variables in the Interlisp package.



9. The F-structure window

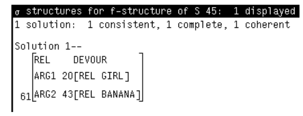
The f-structure window displays the features structures associated with some constituent. The name of the constituent and the number of feature structures displayed is printed in the title bar. Immediately under the title bar is displayed the number of solutions plus how many of them are consistent, complete and coherent. If the feature structures do not fit in the window, then it is possible to see more of them by using the scrollbars that appear to the left and at the bottom of the window when you slowly move the mouse past these edges from inside the window.

The INCONSISTENT, INCOMPLETE, and INCOHERENT buttons at the top of the f-structure window control which feature structures are displayed. If none of the buttons are on, then only good solutions are displayed. Turning buttons on adds to the number of features structures displayed. For instance, if the INCONSISTENT button is on, then any solutions that are inconsistent but not incomplete or incoherent are displayed in addition to the good solutions. If the INCONSISTENT and the INCOMPLETE buttons are on, then any solutions that are inconsistent and/or incomplete but not incoherent are displayed in addition to the good solutions. A solution is considered “incomplete” only if it is not possible for it to be completed. Thus a VP feature structure with a missing subject is not considered incomplete because there is still the possibility that the subject may be filled in at a higher level. Most incomplete solutions show up at the top of the tree; however some solutions are marked incomplete lower down in the tree because the desired feature is known not to appear anywhere in the sentence. Such globally incomplete solutions are discussed in Section 3.3 of Chapter II.

The EXPANDED button displays information about the functional uncertainties. Any attribute that a functional uncertainty passes through will be highlighted by being printed in bold. In addition, any functional uncertainties that are suppressed because they include an empty string and so do not interact with other attributes will be highlighted.

The SYMBOLSINLINE parameter controls whether several symbol-valued attributes are displayed on a single line, usually making the f-structure smaller, or whether they are shown in the more typical vertical arrangement. SYMBOLSINLINE can be specified as a configuration parameter, as discussed in Section III.6.

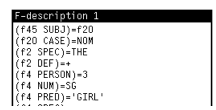
If you click on an f-structure with the left mouse button, then the constraints that describe it (but not any of its projections) will be displayed. If you click on it with the middle button, then a projection of that f-structure will be displayed, if any. If there is more than one projection, then you will be given a menu of projections to choose from.



10. Other projection windows

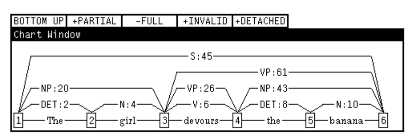
There can be as many projection windows as there are distinct projections. The only way to get a projection window is to click in some other window on a structure that is in the domain of that projection. The projection windows are very similar to the f-structure window, except that there is no local control of which solutions are displayed. If you are looking at a projection and you want to see its inconsistent solutions, then you have to go back to the f-structure window and look for a solution that is marked inconsistent but has no inconsistencies in it. Then you must follow that solution to the projection you are interested in to see whether it is inconsistent there.

If you click on a projection structure with the left button, then the constraints that it satisfies will be displayed in the description window. If you click on a structure with the middle button, then a projection of that structure will be displayed, if any. If there is more than one projection, then you will be given a menu of projections to choose from. Projection windows also have an `EXPANDED` button to control the highlighting of uncertainty paths at that level of representation, and the structure display is also controlled by the `SYMBOLSINLINE` parameter.



11. The description window

The description window displays the constraints associated with some object. The type of the object is indicated in the title bar. If the object is a tree structure node from the c-structure window, then all of the constraints that have been collected from all of the sub-trees will be displayed, including all of the constraints for all of the different projections. Otherwise, only the constraints associated with a particular solution and a particular projection are displayed. If there are any template invocations in the constraints, then those invocations are printed followed by their expansions. The description window can be scrolled using the scroll bars that appear at the left and bottom edges of the window when you slide the mouse slowly past them.

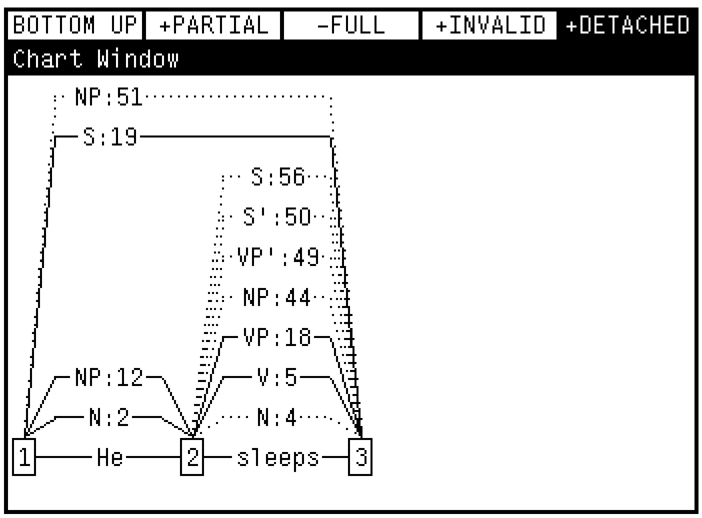


12. The chart window

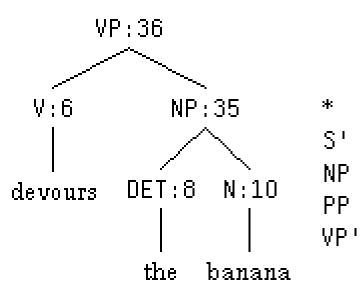
The chart window displays the constituents in the chart. The display of this window is tied to the `CHART` button of the c-structure window menu. The chart window is not visible on the screen unless the `CHART` button of the c-structure window is selected. Deselecting the `CHART` button will close the chart window, and closing the chart window (by using the right-button menu) will cause the `CHART` button to be deselected. If the chart doesn't fit in the window, then it can be scrolled using the scrollbars that appear on the left and bottom edges when you move the mouse slowly past them. Whether or not the constituents have identifiers attached to them is determined by the state of the `NODE NUMBERS` button on the c-structure window.

The buttons on the top determine which constituents are displayed. The BOTTOM UP button adds the constituents that would have been displayed if the chart had been produced bottom up instead of top down. The +PARTIAL button adds the partial constituents. A partial constituent represents an intermediate step in the process of matching a rule. Thus, the rule $\mathtt{S} \rightarrow \mathtt{NP:(}\uparrow \mathtt{SUBJ)=}\downarrow\mathtt{;\ VP}$ allows for a partial constituent that covers an NP but does not incorporate a VP. This partial constituent is labeled with /S and spans the NP. The slash indicates that the constituent by itself is incomplete. The −FULL button suppresses the display of full (non-partial) constituents. The +INVALID button adds the invalid constituents to the display. Constituents can be invalid either because they only have bad solutions (in which case they are boxed) or because one of their daughters has no good solutions (in which case they are not boxed, but some constituent in every one of their alternative subtrees will be boxed).

The +DETACHED button adds the detached constituents. Detached constituents are those that are not part of any rooted tree. They are visibly distinguished from the others by the fact that they are represented by dotted lines instead of solid lines:

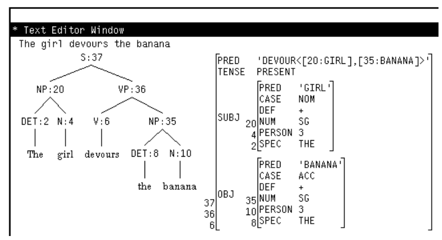


If you click on a word in the chart window, then an edit window is opened on its definition. If you click on a constituent in the chart window, then its trees will be displayed in the c-structure window. If you click on a constituent while the +PARTIAL button is on, then each tree will be followed by a column of category names that represents the possible continuations of that constituent considered as a partial (that is, some constituents that look like they are complete can be continued, and so can be considered partials).



If a * is at the top of the column, then nothing more is needed to complete the constituent. If you click on one of the categories, then the constraints that annotate the rule-element for that category are displayed in the description window. This is useful for understanding whether a continuation is, say, an object NP instead of a second object.

If the chart window is on the screen when you begin the analysis of a new sentence, the window will be cleared in preparation for displaying the chart for the new analysis. However, if the analysis produces a large number of constituents (greater than the value of the Lisp variable `IL:KEEPCHARTWINDOWLIMIT`), then the new chart will not be automatically displayed and the chart window will be closed instead. This is to prevent GWB from spending the time to lay out a large display that may be of little interest to you. If a desired chart display has been suppressed, you can explicitly ask for it to be shown by selecting the c-structure `CHART` button again. The initial value of `IL:KEEPCHARTWINDOWLIMIT` is 150. You can raise or lower its value by executing `(SETQ IL:KEEPCHARTWINDOWLIMIT` *newvalue*`)` in an Executive Window or in your Medley initialization file. If you set the value to `IL:ALWAYS` instead of a number, a visible chart window will always be retained. If you set it to `IL:NEVER`, it will be closed before each new analysis.



13. Printing and including display objects

There are several ways of getting display objects out of windows for printing or including in a document. If, for example, you simply want to take what you see in the f-structure window and send it to a printer, you can select the `Hardcopy` command in the menu that comes up when you click with the right button in the window. This will automatically send the image to your "default printer", which you must set up in your site initialization file. Thus, if `MYPRINTER` is the name of your postscript printer, you would say

```
(SETQ IL:DEFAULTPRINTINGHOST '(MYPRINTER))
```

in your initialization file or in an executive window. This will cause the system to create a postscript file and then execute a platform-specific procedure for sending the file to the printer with that name. On a Unix platform, for example, the shell command `lpr` is invoked to transmit the file. GWB can also create files in the Interpress format and transmit them to Xerox printers.

Alternatively, you can slide to the right off of the `Hardcopy` menu item, and you will be given a choice as to whether you want the image to be printed on some other printer, or whether you want it to be put on a postscript file in your file system, which you can then send to the printer (or ftp to someone else) at a later time. You make up the name of the file, but you probably should give it the extension `.ps` to make sure that it is identified as postscript.

You may also want to take c-structures, f-structures, f-descriptions, or the chart out of their windows and put them into a TEdit file, along with other information, comments, discussion, etc. To do that, you have to open a TEdit window (e.g., use the TEdit item in the right-buttton background menu). Then, with the type-in caret blinking in the TEdit window at the position where you want the insertion to appear, hold down the `SHIFT` key and point at a tree or f-structure, and click the left button. The tree or f-structure should now appear in your TEdit document (you may have to scroll or reshape the window to make it visible). If you hold the `SHIFT` key and click in the f-description window, you can transfer the set of constraints into the TEdit document as editable text. When you release the mouse key, a menu will pop up to ask you whether or not you want an Ascii-only version of the equations. This format is best if the document is to be sent as a plain-text file (for example, as Email) to a non-Medley environment, since it translates all the mathematical symbols and projection designators into charaters and strings in the 127-character U.S. Ascii encoding. When you have finished composing your document, you can print the it by using the hardcopy item in the menu that comes up when you click with the right-button in that window's title bar. You can also save the TEdit file for future use. Plain-text files can be exported to other computing environments.

Finally, you can make snapshots of any portion of the screen by using the `Snap` command on the background menu. This will create a bitmap of whatever portion of the screen that you want. This is useful for temporarily saving a tree or f-structure for comparison with other trees or f-structures. It is possible to include snapshots in a TEdit document by putting the type-in caret in a TEdit window, and then clicking the right mouse button in the background while holding down the `SHIFT` key. This will produce a `Snap` menu item. Select this item and then indicate what part of the screen you want copied. When you are done, the bitmap will appear in the TEdit document where the caret was (again, you may have to reshape or scroll the window to get it to appear). You can edit this bitmap using the menu that appears when you click in it with the left or middle mouse button.

References

Baader, F., Bürckert, H-J., Nebel, B., Nutt, W., and Smolka, G. (1991) On the expressivity of feature logics with negation, functional uncertainty, and sort equations. Research Report RR-91-01, DFKI, Kaiserslautern, Germany.

Bresnan, J. (1982a) Control and complementation. In J. Bresnan (ed.), *The mental representation of grammatical relations.* Cambridge: MIT Press, 1982, 282–390.

Bresnan, J. (1982b) The passive in lexical theory. In J. Bresnan (ed.), *The mental representation of grammatical relations.* Cambridge: MIT Press, 1982, 3–86.

Bresnan, J. (1982c) Polyadicity. In J. Bresnan (ed.), *The mental representation of grammatical relations.* Cambridge: MIT Press, 1982, 149–172.

Bresnan, J. and Kanerva, J. (1989) Locative inversion in Chichewa: a case study of factorization in grammar. *Linguistic Inquiry* 20 (1), 1-50.

Dalrymple, M., Kaplan, R., Maxwell, J., & Zaenen, A. (eds.), *Formal issues in Lexical-Functional Grammar.* Stanford: CSLI Publications, 1995.

Gazdar, G., Klein, E., Pullum, G., and Sag, I. (1985) *Generalized phrase structure grammar*. Cambridge: Harvard University Press.

Halvorsen, P.-K. & R. Kaplan. (1988) Projection and semantic description in Lexical-Functional Grammar. *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, November 1116-1122. Also appears in Dalrymple *et al*., 1995, 279-292.

Kaplan, R. (1987) Three seductions of computational psycholinguistics. In P. Whitelock, H. Somers, P. Bennett, R. Johnson, and M. Wood, *Linguistic Theory and Computer Applications*. London: Academic Press, 149–188. Also appears in Dalrymple *et al*., 1995, 339-367.

Kaplan, R. (1989) The formal architecture of lexical-functional grammar. *Journal of Information Science and Engineering* 5, 305-322, 1989. Also appeared in *Proceedings of ROCLING II*, Taipei, Republic of China, 3–18. Also appears in Dalrymple *et al*., 1995, 7-27.

Kaplan, R. & Bresnan, J. Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan (ed.), *The mental representation of grammatical relations.* Cambridge: MIT Press, 1982, 173–281. Also appears in Dalrymple *et al*., 1995, 29-130.

Kaplan, R. & Maxwell, J. (1988a) An algorithm for functional uncertainty. *Proceedings of COLING 88*, Budapest, 297–302. Also appears in Dalrymple *et al*., 1995, 177-197.

Kaplan, R. & Maxwell, J. (1988b) Constituent coordination in Lexical-Functional Grammar. *Proceedings of COLING 88*, Budapest, 303–305. Also appears in Dalrymple *et al*., 1995, 199-210.

Kaplan, R. & Maxwell, J. (1993) The interface between phrasal and functional constraints. *Computational Linguistics 19,* 571-590*.* Also appears in Dalrymple *et al*., 1995, 403-429.

Kaplan, R. & Wedekind, J. (1993) Restriction and correspondence-based translation. *Proceedings of the Sixth Conference of the Association for Computational Linguistics European Chapter*, Utrecht, 404-411.

Kaplan, R. & Zaenen, A. (1989a) Functional precedence and constituent structure. *Proceedings of ROCLING II*, Taipei, Republic of China, 19–40.

Kaplan, R. & Zaenen, A (1989b). Long-distance dependencies, constituent structure, and functional uncertainty. In M. Baltin and A. Kroch (eds.), *Alternative conceptions of phrase structure*. Chicago: The University of Chicago Press, 17–42. Also appears in Dalrymple *et al*., 1995, 137-165.

Kaplan, R., Netter, K., Wedekind, J., & Zaenen, A. (1989) Translation by structural correspondences. *Proceedings of the 4th Conference of the Association for Computational Linguistics European Chapter*, University of Manchester Institute of Science and Technology, 272–281. Also appears in Dalrymple *et al*., 1995, 311-329.

Keller, B. (1991) Feature logics, infinitary descriptions and the logical treatment of grammar. Cognitive Science Research Report 205, University of Sussex.

Maxwell, J. & Kaplan, R. (1991) A method for disjunctive constraint satisfaction. In M. Tomita (ed.), *Current issues in parsing technology*. Boston: Kluwer Academic Publishers, 173-190. Also appears in Dalrymple *et al*., 1995, 381-401.

Pollard, C. & Sag, I. (1994). *Head-driven phrase structure grammar*. Chicago: The University of Chicago Press.

Xerox Systems Institute (1987) *Character code standard*. Xerox Corporation, document XNSS 058710.

Zaenen, A. and Kaplan, R. (1995) Formal devices for linguistic generalizations: West Germanic word order in LFG. In Dalrymple *et al*., 1995, 215-239.

Appendix

Editing Text with TEdit

[This appendix is excerpted from Chapter 3 of *The TEdit User's Guide*, copyright © 1993 by Venue. It is reprinted by permission. Please see that document for a full discussion of TEdit capabilities.]

This chapter describes basic text editing, such as how to enter, select, delete, move, and copy text. It also tells you how to create and use blanks to be filled in later; how to undo and redo edit operations; and how to type special characters. Finally, this chapter shows you how to work with graphic images.

Entering Text

The blinking caret (‸) in the TEdit window is called the *type-in point.* Medley has only one type-in point active at a time; if the caret is not in a TEdit window, you move it there by moving the cursor into the window and pressing any mouse button. Whatever text you type will appear at the caret.

TEdit automatically breaks text between words and sends the overflow to the next line. Don't hit carriage return at the end of each line; use one only when you want to begin a new paragraph or insert blank space in addition to that provided by line and paragraph *leading*.

To type a carriage return without causing a paragraph break, hold down the meta key while you type the carriage return. This is called a meta-return.

Selecting Text

To change text you have typed in, you first mark where you want the change made by making a *selection;* then use one of TEdit's commands to actually make the change.

You can make only one selection at a time. Most TEdit formatting operations act on the currently-selected text, called the current selection. You can tell when text is selected because it is underlined or *highlighted* (inverse video). The current selection usually has a caret flashing at one end (see figure 6).

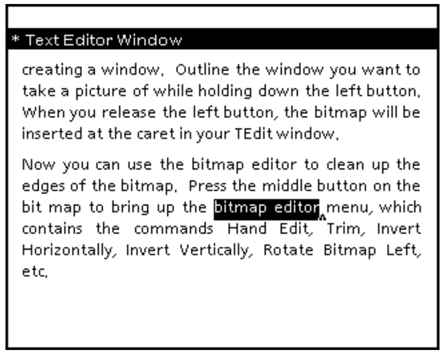


Figure 6. Selected Text. A TEdit window showing some text highlighted as a delete selection.
When you begin typing, "bitmap editor" is deleted and replaced by the text you type

Units of Text Selection

You select text by pointing with the cursor, then pressing one of the mouse's buttons. The left button always selects the smallest units of text. In the text-editing region, it selects the character you're pointing at; in the line bar, it selects the single line you're pointing at.

The middle button selects larger units. In the text-editing region, it selects the word the cursor is over. TEdit sees a word as a contiguous collection of letters and numbers, punctuation marks, special symbols (such as percent sign), or of white space characters (such space or tab). When you select a word with the middle button, TEdit starts with the character the mouse is over and scans left and right until it finds a *word boundary*—a change from one of the four types of "words" to another.

In the line bar, the middle button selects the paragraph the cursor is next to. To TEdit, a paragraph consists of all the text between two carriage returns and includes the second carriage return. (If you have typed a meta-carriage return as described above, it is *part* of the paragraph rather than terminating it.)

The right button always extends a selection. If you select a place in the text with the left or middle button, move the cursor somewhere else, and press the right button, all the text between the two points is selected. If the existing selection is a whole word, line, or paragraph, the extended selection will also consist of whole words, lines, or paragraphs.

There are also special ways of selecting text that carry an implicit command with them. These are described below in the sections on deleting, moving, and copying text.

You can select all the text in a document by choosing the **All** command in the Expanded menu (see Chapter 4, The TEdit Menu). The **All** command is especially useful for general formatting operations, such as changing the typeface for an entire document.

Deleting Text

There are five ways to delete text from a TEdit window:

1. Selecting the text, then pressing the DELETE key.
2. Extending a selection with the right button will cause the selection to be highlighted. When you type any new text the highlighted selection is deleted; i.e., you do not have to use the DELETE key. If you are replacing text, this method is very efficient.
3. If you hold down the CONTROL key while selecting text, the text (which is highlighted) will be deleted when you release the key. You can abort a control-selection by holding down any mouse button, releasing the CONTROL key, then releasing the mouse button.
4. You can delete text one character at a time by pressing the backspace key (or CONTROL-A); the character to the left of the caret will disappear.
5. You can delete text one word at a time by pressing CONTROL-W, which deletes the word just before the caret.

Moving Text

To move text, put the caret where you want the text moved *to*. Then select the text to be moved while holding down the MOVE key (or the CONTROL and SHIFT keys). The text to be moved will be highlighted. When you release the key(s), the text will be moved to the location of the caret and deleted from its original location. You can use this method to move text within a TEdit window or to move it from one window to another. To abort a move, hold down any mouse button, release the MOVE (or CONTROL and SHIFT) key(s), then release the mouse button.

Copying Text

If you want to copy text, first put the caret where you want the text copied *to*. Then hold down the COPY or SHIFT key while selecting the text to be copied. This *copy source* is marked with a dashed underline. When you release the COPY (or SHIFT) key, the text will be copied to the location of the caret. You can copy text within a TEdit window or from one window to another. You can abort a copy by holding down any mouse button, releasing the COPY key, then releasing the mouse button.

Marking Fields to Be Filled In

TEdit's *next* feature enables you to easily create forms with blanks (called fields) to fill in, then move forward from one field to another (see figure 7). You mark an area as a field by enclosing it in double reverse angle brackets, like this:

```
>>text to replace<<.
```

To locate the first field after the caret, press the NEXT key. TEdit will highlight the area as the current selection; whatever you type will replace the angle brackets and any text they contain. You can sequentially locate the other fields by pressing NEXT.

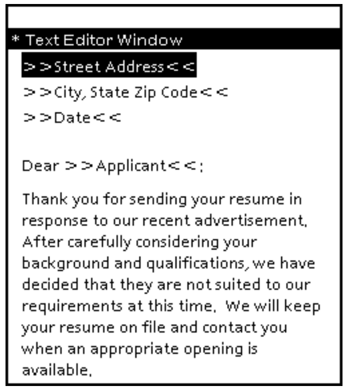


Figure 7. TEdit's Field Filling Feature. A portion of a form letter written using the TEdit next feature

Undoing an Edit Operation

TEdit allows you to undo your most recent edit operation, which is handy when you have done something like select a large area of text to copy and then press the DELETE key instead of the COPY key. You can undo an operation by pressing the UNDO key immediately afterwards. Undo is itself undoable, so you can never back up more than a single operation.

Redoing an Edit Operation

To redo your most recent edit command on the current selection, press the AGAIN key immediately afterwards.c For example, if you insert some text, then place the caret elsewhere, pressing AGAIN will insert a copy of the text in the new place also. If the last command was a deletion, pressing AGAIN will delete the currently selected text; if it was a font change, the same change will be applied to the current selection.

Typing Special Characters

TEdit has 20 *abbreviations* that enable you to type special characters that are not available on your workstation keyboard. (You can also add your own as described in Chapter 8, TEdit's Programmatic Interface).

To produce the special characters you:

1. Type in the abbreviation for the desired character.
2. *Expand* the abbreviation by selecting it and pressing the EXPAND key. A single-character abbreviation is automatically selected right after you type it in; just press the EXPAND key.
3. To expand a multi-character abbreviation, you must explicitly select it before pressing EXPAND.

After you press EXPAND, the abbreviation will be replaced by its expansion. The abbreviations and their expansions are:

Table 1. TEdit's abbreviations and their expanded characters

Abbreviation	Expanded character name	Expansion Character
"	Open double quotation mark	“
~	Close double quotation mark	”
‘	Open single quotation mark	‘
’	Close single quotation mark	’
b	Bullet	•
c	Copyright sign	©
m	Em-dash (used to separate text phrases)	—
n	En-dash (used to indicate inclusive numbers, as in “pages 3–6”)	–
d	Dagger	†
D	Double dagger	‡
s	Section sign	§
1/2	Built-up fraction	½
1/4	Built-up fraction	¼
2/3	Built-up fraction	⅔
3/4	Built-up fraction	¾
DATE	The current date	October 15, 1985
>>DATE<<	The current date	October 15, 1985
p	Pilcrow (proofreader's paragraph mark)	¶
t	Trademark	™
tm	Trademark	™
r	Registered trademark	®
1/3	Built-up fraction	⅓
x	Times sign	×
/	Division sign	÷
o (oh)	Degrees sign	°
L	Pound sterling sign	£
Y	Yen sign	¥
+	Plus-or-minus sign	±
^(shift-6)	Up arrow (NS character)	↑
ua	Up arrow (NS character)	↑
|	Down arrow (NS character)	↓
da	Down arrow (NS character)	↓
<-	Left arrow (NS character)	←
la	Left arrow (NS character)	←
_ (underscore)	Left arrow (NS character)	←
->	Right arrow (NS character)	→
ra	Right arrow (NS character)	→
=	Two-way arrows (NS characters)	⇔

Working With Graphic Images

It's easy to add graphic images like rules, bitmaps, graphs, and drawings to your documents. Rules are created with HRule and graphs with Grapher, both described in the *Medley Library Manual*. Bitmaps of parts of the screen are created with “SHIFT-snap” and edited with EditBitMap (see the *Medley Library Manual*). Drawings are created with Sketch and are fully discussed in the *Sketch Reference Manual.*

Once you have created a graphic image, you can copy or move it into a TEdit document just as you would copy or move text from one TEdit window to another. TEdit sees each graphic image as a single character, which means it can be selected, copied, moved, and deleted like any other character. You can use the Paragraph Looks menu to center a graphic image or line it up; it will qualify as a single paragraph if you insert a carriage return before and after it.

When you scroll a TEdit window, the graphic image will move all at once, sometimes leaving unexpected blanks until you haven't scrolled far enough for the entire image to appear in the window. If the image still does not appear, try enlarging your window. The image will be saved when you save the TEdit file, and will be printed when you hardcopy the document.

Putting Bitmaps Into TEdit Files—SHIFT-snap

An easy way to put bitmaps into your TEdit files is to use the SHIFT-snap facility. To activate the facility, which is loaded when you load TEdit, you:

1. Make sure the caret is in the TEdit window and at the position where you want the bitmap placed.
2. Move the cursor into the screen background.
3. Hold down the SHIFT key.
4. Press the right mouse button. A small box containing the word "snap" will appear at the cursor (see figure 8). Once the box appears you can release the SHIFT key.

Figure 8. The "snap box." On the left as it appears when first called and on the right as it appears after you move the cursor into the box

5. Move the cursor into the box; "SNAP" becomes highlighted (see figure 8).
6. Release the mouse button. The cursor changes shape into the expanding box cursor ( ).
7. Press the left button. The cursor changes to a box that appears reinforced at one corner; the "reinforced" corner is the control point that follows the direction of the mouse's movement.
8. Move the mouse so that the area of the screen that you want to take the SHIFT-snap of is within the box. You can take a SHIFT-snap of anything that's on the screen. If you need to shift the control point to another corner press and hold the right mouse button while continuing to hold down the left button. The cursor changes shape to a forceps ( ). Move the forceps to the new control corner and release the right button. The control point is moved to that corner.
9. Release the left button. A bitmap picture of the area contained in the box is placed in your TEdit window at the point of the caret. If the bitmap doesn't appear, it may be too large to be displayed in the available window space; either scroll the window or make the window larger.
10. Once the bitmap is in your TEdit file, if you move the cursor into it and press the left mouse button, the bitmap editor menu appears. See the EDITBM Library module for details.