KPML Development Environment

**— multilingual
linguistic resource development
and sentence generation**

**Release 1.1
January 1997**

John Bateman
e-mail: `bateman@gmd.de`

Institut für integrierte Publikations- und Informationssysteme (IPSI)
German Centre for Information Technology (GMD)
Dolivostr. 15, Darmstadt
Germany.

URL="`http://www.darmstadt.gmd.de/publish/komet/kpml.html`"

Acknowledgements

Many people have contributed during the beta releases (0.1–0.9) to the current state of KPML, both in terms of actual code additions/corrections and in terms of critical feedback: particular thanks go to Markus Fischer (ITRI, Brighton) for the original port to CLIM-2, to Cécile Paris and Keith Vander Linden (ITRI, Brighton) for various bug fixes, to Richard Whitney (ISI, Los Angeles) for the code for the Loom 2.1 conversion, to John Wilkinson (Univerisity of Waterloo, Canada) for several speed-ups, to Tony Hartley (ITRI, Brighton), Cécile Paris, Brigitte Grote (FAW, Ulm), Elke Teich (IPSI, Darmstadt), Liesbeth Degand (Université Catholique of Louvaine-la-neuve), Bernhard Hauser (Technische Hochschule, Darmstadt) for providing much feedback throughout the early releases, and to Melina Alexa (IPSI, Darmstadt) and Fabio Rinaldi (University of Udine and IRST, Trento) for test driving the current interface and the documentation.

Fabio Rinaldi also receives a special additional vote of thanks for preparing the WWW-versions of this documentation.

Section 2.1 is adapted from Bateman, Matthiessen & Zeng (1996). Appendix B, concerning the interface between inquiry implementations and knowledge base, is taken from Bob Kasper's contributions to the Penman documentation.

The 'old-style' KPML interface (Chapter 8) is an outgrowth of a Penman window interface written by Richard Whitney and Kevin Knight (ISI), which was in turn based on a text planner interface by Vibhu Mittal and Cécile Paris (ISI). The simpler, non-graphical generation debugging tools and grammar tracing facilities build on those of the Penman system—particularly those parts concerned with the grammar. Hence, corresponding parts of this guide are updates of the Nigel Manual (Penman Project, 1989, ISI), originally prepared by Lynn Poulton, Christian Matthiessen and John Bateman.

The original beta versions of the KPML interface and resource development environment (0.1–0.5) and its documentation were prepared in the context of a DAAD/British Council cooperation project (DAAD/ARC-313) between GMD/IPSI and ITRI (University of Brighton). The development of the Penman system on which the KPML system builds was supported by the U.S. National Science Foundation Grant IST-8408726, and U.S. Federal Contract numbers MDA903-81-C-0335, MDA903-87-C-0641, F49620-84-C-0100, and F49620-87-C-0005. The multilingual extensions to the system have been supported in part by the German Ministry for Research and Technology (BMFT: Project 'INTEGRA') and in part by the Australian Research Council. The development of the stratification available in the experimental KPML-E versions of the system for supporting text type (genre) networks and register has been supported by U.S. National Science Foundation Grant IRI-9003087, the European Union Basic Research Action EP-6665 ('Dandelion'), and the German BMFT Project 'INTEGRA'.

Contents

List of Figures

List of Tables

Prerequisites

This user manual assumes a basic knowledge of using Common Lisp and editing and writing files.

Basic knowledge of GNU Emacs is also very helpful, since this is the preferred environment for using KPML.

KPML is *not* intended as a system for use in isolation of the above functionalities, although resource browsing and generation of existing examples can be carried out without requiring knowledge of Lisp, Emacs, or the operating system under which KPML is running.

The preferred software required to run KPML is Allegro Common Lisp version 4.2 or newer with the Common Lisp Interface Manager (CLIM) 2.0 or newer. KPML will run with other Common Lisps and CLIM 1.0 but with reduced functionality. Also required in the default release configuration is the knowledge representation system LOOM, available free of charge from USC/ISI (e-mail: `tar@isi.edu`). It is possible to use KPML without LOOM however.

Basic knowledge of the organization of a systemic-functional account is also assumed, although Chapter 2 provides a brief overview and pointers to further information.

Chapter 1

Introduction

1.1 The purpose of the system

The KPML (Komet-Penman multilingual) development environment is a system for developing and maintaining large-scale sets of multilingual systemic-functional linguistic descriptions (as originally set out in Bateman, Matthiessen, Nanri & Zeng (1991*b*), Bateman, Matthiessen, Nanri & Zeng (1991*a*) and Matthiessen, Nanri & Zeng (1991)), and for using such resources for text generation. More generally, the intended purposes of KPML are:

- to offer generation projects large-scale, general linguistic resources which:
  - are well tested and verified in their coverage,
  - possess standardized input and output specifications,
  - and are appropriate for practical generation;
- to offer generation projects a basic engine for using such resources for generation;
- to encourage the development of similarly structured resources for languages where they do not already exist,
- to provide optimal user-support for undertaking such development and refining general resources to specific needs;
- to minimise the overhead (and cost) of providing texts in multiple languages;
- to encourage contrastive functional linguistic work;
- to raise awareness and acceptance of text generation as a useful endeavor.

A fundamental tenet of the approach followed with KPML is that it is often mistaken to simplify the generation task by simplifying or restricting the linguistic resources employed, just because resource development or coverage is not a primary concern. Kpml attempts to simplify the generation task by improving *access* and *handleability* of large-scale resources. This should prompt projects to work with large-scale resources, even when the main aims are elsewhere. The benefits of this are that fragmented solutions that do not scale up are more easily avoided, and that proof-of-concept demonstrations can draw on a more realistic strategic generation capability. KPML seeks to offer a stable development and generation environment that can be used for application-near text generation and demonstration.

1.2 The functionality of the system

The KPML development environment provides a convenient platform for the construction and maintenance of multilingual linguistic resources. Interaction with the system is predominantly by combined mouse-click and graphical/textual information presentation. User commands are offered for loading definitions of (multilingual and monolingual) linguistic resources in systemic form, displaying these resources in a variety of ways useful for development and maintenance, performing static integrity checks on the systemic network defined by the resources loaded, and using the resources for generation. It is also possible to use the system as a blackbox tactical generator. The environment takes over and extends the functionality of the Penman text generation system (Mann 1983*b*, Mann & Matthiessen 1985, Penman Project 1989), going beyond that system in terms of ease of use, development support, and multilingual design.[1] The internal components of KPML and their functionality and communication channels are shown in the block diagram in Figure 1.1.

Particular points of emphasis of the system include:

- an integrated view of examples and linguistic resources: resource maintenance is supported by extensive test suites which are interlinked with the resource definitions providing example-based on-line 'documentation';
- the possibility of combining graphical views of the linguistic resources with particular details of the generation process: generation debugging is graphically driven;
- a very high degree of modularity in the linguistic resource definitions;
- very extensive graphical and textual inspection of all aspects of the linguistic resources and their use;
- automatic resource management, including patch facilities for extending linguistic resources;
- provision of 'fast generation' modes;
- provision for template-based generation,
- provision of structured and annotated 'string' generation to support hyperlinks and other application specific mouse-driven functionalities;
- multilinguality throughout.

[1] The multilingual generation functionality is based on the the multilingual extensions to the Penman system made by Licheng Zeng (University of Sydney) as documented in Zeng (1992). Other extensions in KPML include provision of an integrated systemic morphology, work on higher levels of text organization, such as genre and register, as well as numerous code improvements and bug fixes. Only those aspects of the system relevant to developing and maintaining multilingual grammatical resources are described in this documentation however; for overviews of other aspects, see, for example, Bateman & Teich (1995).

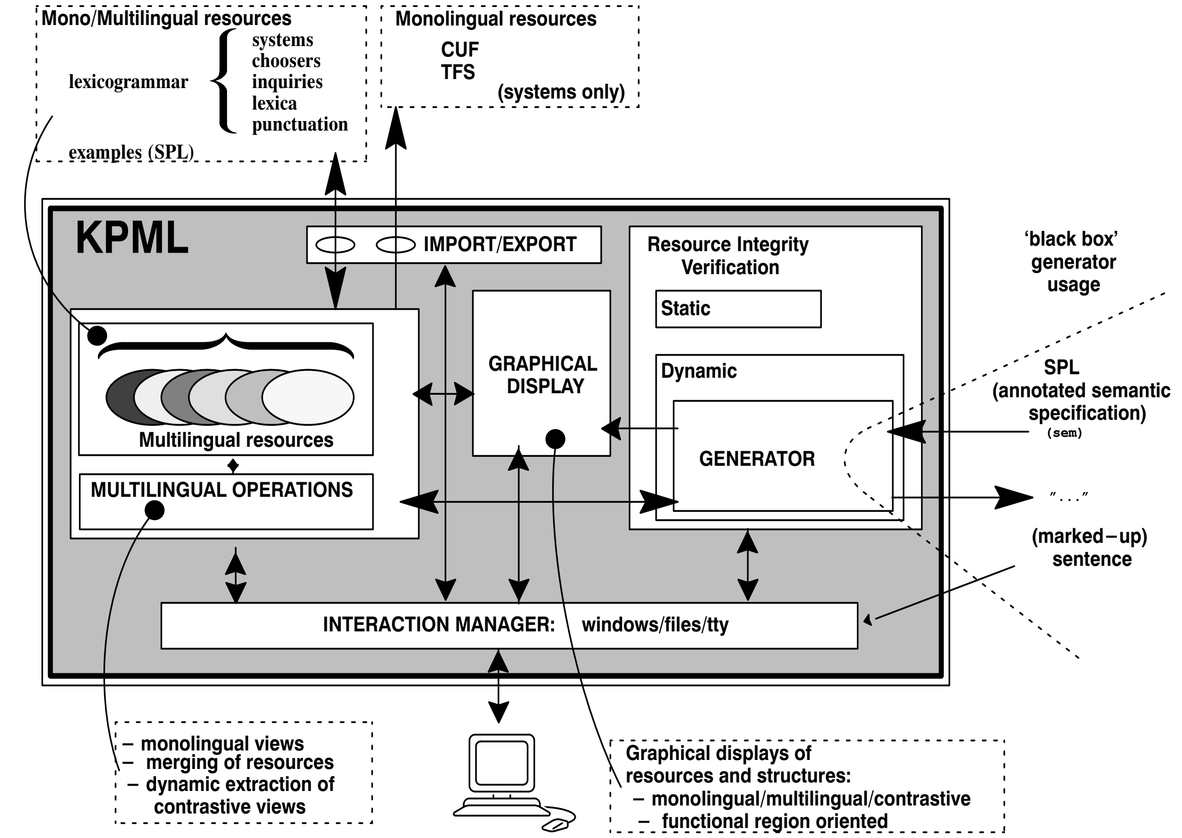


Figure 1.1: Internal KPML components

The view of multilingual resources supported by KPML defines a very fine granularity of language-specific conditionalization. In Bateman, Matthiessen, Nanri & Zeng (1991*a*) and Bateman, Matthiessen & Zeng (1996) we claim that this organization generalizes substantially beyond all previous approaches to multilinguality. The development environment is also released with sizeable examples of multilingual linguistic resources; the most substantial of these being the English grammar Nigel, originally developed within the Penman project , and the KOMET grammars for German and Dutch.[2]

The basic units manipulated by the system are *grammatical systems*, *choosers*, *inquiries*, *lexical units*, *punctuation rules*, and *examples* (the latter including Penman-style SPL input specifications: Kasper 1989). All of these are potentially multilingual in that their contributing elements may be conditionalized to apply in specific languages. Using these object-types as the basic units that the system manages allows KPML to offer fully automatic merging and dynamic extraction of monolingual and contrastive views of those objects. That is, given that a system of a given name is defined as having various forms depending on the language in which it is used, KPML can freely merge such descriptions and subsequently take them apart. As argued in Bateman, Matthiessen, Nanri & Zeng (1991*a*) and elsewhere, this is a useful

[2]For details of these resources, see their respective documentation and descriptions (Matthiessen 1995, Teich 1992, Degand 1993).

approach to managing multilinguality since it constructs multilingual descriptions around the *paradigmatic* unit rather than the structural: functional equivalences are therefore more likely to be preserved.

Building on this functionality, monolingual language descriptions can be freely and automatically merged to produce multilingual specifications and, from these, further monolingual or contrastive resource sets can be automatically extracted. Dynamic contrastive browsing of the resource sets is also supported, as well as contrastive generation and special features for the rapid development of resources for languages not previously handled.

1.3 Overview of the interface organization

The user interface for KPML provides specialized windows for three distinct kinds of work that are typically involved in linguistic resource development and maintenance. These are:

- Loading and saving sets of linguistic resources and determining overall system behavior.
- Developing, maintaining and debugging sets of resources by generation.
- Inspecting linguistic resources and objects.

Each of these activities requires different commands and are conceptually separate. The documentation reflects this separation and describes the functionalities offered to support each activity in distinct chapters.

Interaction with KPML is essentially object-oriented: analogously to, for example, Smalltalk or ALEP (Simpkins 1992), particular objects permit particular operations. For example, a grammatical system allows operations such as inspecting, loading and modifying. Objects from a wide range of linguistic granularity are maintained; there are both large-scale objects (such as a resource set, a grammar, and a functional region of a grammar) and very fine-scaled objects (such as the individual elements of realization statements used within grammatical systems). Operations can be parameterized in various ways: according to linguistic object, according to language, according to grammatical region.

The principal operations supported by KPML are as follows:

Loading: the constitutive objects of a designated class (i.e., all objects for a designated linguistic 'resource', or the systems, choosers, inquiries of a designated 'functional region') are read into the system and become part of the loaded linguistic resources; for most classes of object it is not necessary for a user to know the files of origin for the object definitions affected.

Writing: the constitutive objects of a designated class are written out to the file system; again it is not generally necessary for a user to know which files are created by this operation. The Loading/Writing operations are complementary, creating the required directory structures on writing and assuming these structures on loading.

Graphing: the designated object is shown in a mouse-sensitive graphical form, showing the interrelationships among its constitutive sub-objects. This is most commonly used for subnetworks drawn from the system network definition as a whole (consisting of collections of grammatical systems), choosers and generated syntactic structures. Significantly for debugging and maintenance, most graphing operations also allow *the actual options instantiated during generation* to be highlighted in the graphical display of any graphed object; this is the most common starting point for maintenance and debugging.

Inspecting: the designated object definition is shown in a mouse-sensitive textual form in the KPML inspector window.

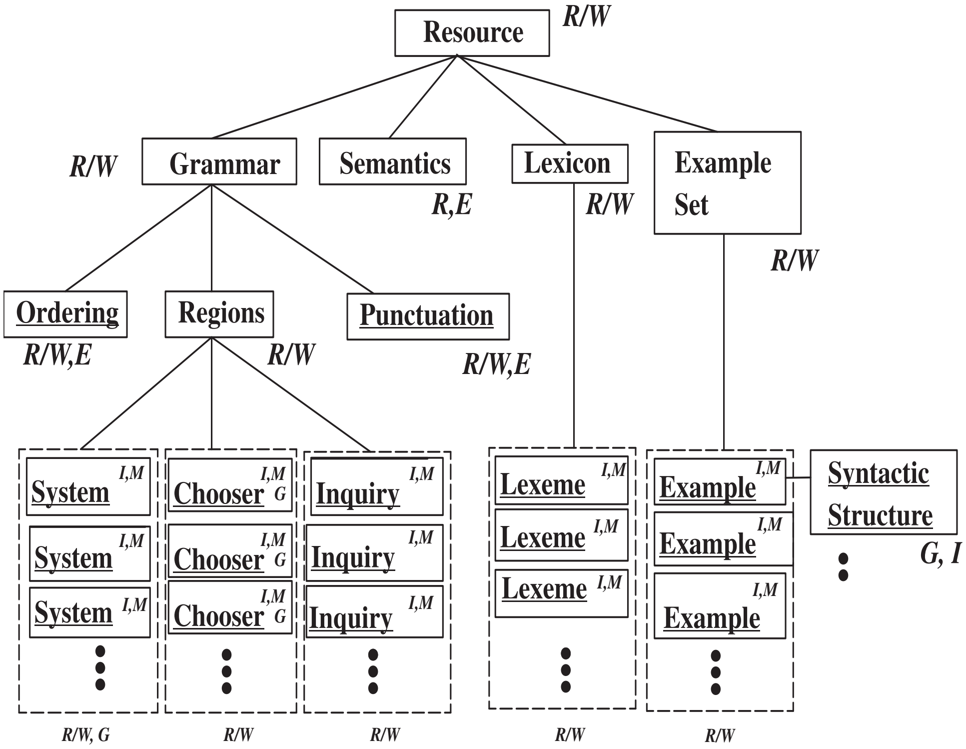
Modifying: the designated object definition is brought into a GNU Emacs editor buffer where it may be modified; on return from the editor buffer the modified definition is automatically added to the loaded definitions active.

Editing: the designated object must be brought by the user from its definition file into an editor and altered.

The objects supported and their hierarchical constituent relationships, as well as a summary view of the operations relevant, is set out in Figure 1.2. The types of objects generally follow closely those of the Penman-style of linguistic resource description introduced above. There are also some additional objects that represent the *results* of generation: for example, the generated string and its underlying constituent structure. These may be inspected, graphed, and, via inclusion in an example set, written and loaded. They cannot, however, be directly modified since they are only meaningful with respect to a particular generation history. Finally, some meta-information concerning a resource set is also maintained—including, for example, declarations of the fonts to be used for a given language allowing development in non-Latin writing systems.

In addition to the constituency relations shown in Figure 1.2, there are also relations between objects. These cross-links follow directly from the use of the linguistic objects during generation (cf. Figure 2.4 and the more detailed description below).

For example, a grammatical system typically has a chooser that is used to determine the appropriate choice of grammatical feature, and a chooser consists of several inquiries organized in a decision tree. Individual inquiries also have *inquiry implementations* that are responsible for querying knowledge sources, classifying concepts, etc. The existence of such object cross-links supports the use of *information chains* by which a resource developer may obtain information concerning aspects both of the resource definitions and of their deployment during generation. This makes it straightforward, for example, to start from a particular generated example and to follow an information chain from a substring to its grammatical constituent to the path of grammatical features through the system network responsible for that constituent to a particular grammatical system to a particular chooser to a particular inquiry which caused a particular inquiry implementation to be triggered with particular actual parameters: each transition is achieved by a single mouse-click. Navigating all components of the resource architecture as set out in Figure 2.4 is thus rendered straightforward regardless of the size of the resources maintained.



Key:
R: load, W: write, E: edit, M: modify, G: graph, I: inspect.
Underlined objects may be explicitly conditionalized for multilinguality; other objects inherit their multilinguality from their contained objects.

Figure 1.2: Linguistic objects and operations supported by KPML

1.4 Overview of the documentation

This user guide is primarily concerned with enabling a user to use KPML for resource development and multilingual text generation. Particular sections provide details on:

- installing and loading the release version of the KPML system (Chapter 3),
- loading released versions of linguistic resources into the system for inspection or generation (Section 5.7),
- inspecting the organization of loaded resources (Chapter 6),
- testing the integrity of resources and using them for generation (Chapters 9 and 10),
- creating new resources (Section 5.9.1).
- using the system in blackbox generation mode as a tactical generator (Section 14.1),
- using the system directly from other Lisp programs (Section 14.4) and from other processes/machines (Chapter 17).

Finally, although this document assumes a general familiarity with systemic-functional linguistic representations, a brief abstract overview is given in Section 2.1, and a set of pointers to further information is given in Section 2.4.

1.5 Availability of the system

KPML is freely available for research purposes; the latest public release can be found on the IPSI ftp-server, `ftp.darmstadt.gmd.de`, under the directory `/pub/komet`. The system is written in Common Lisp, using CLOS and CLIM. The default resource configuration also assumes the presence of the knowledge representation language LOOM, available free from ISI, Los Angeles, for some of their semantic specifications. Use of other knowledge representation systems is straightforward (see Appendix C).

Note: the full functionality of KPML is now dependent on Allegro Common Lisp 4.2 or newer with CLIM 2.0 or newer.

The system will run with reduced functionality (approximately that of KPML 0.8) on other Common Lisp configurations; in particular with:

- Lucid Common Lisp 4.1 with and without CLIM 1 (SunOS 4)
- Lucid Common Lisp 4.2 with and without CLIM 2 (SunOS 5.3/Solaris 2.3)

Note that due to ongoing code changes that are bringing KPML into accordance with *Common Lisp the Language*, second edition (Steele Jr. 1990) and the ANSI standard, KPML is no longer available for any Lisps prior to the versions given above. Moreover, the version for Allegro

Common Lisp/CLIM is the only version that is currently being fully supported.[3] Standalone versions of the system can be made available for Solaris 2.3 and up, and for PCs running Linux (without the user interface) and Windows (with the user interface).

1.6 Known bugs/problems

The following are known to be problematic or missing at the time of the present release of KPML.

- The multilingual interaction of stacked SPL default environments from differing languages simultaneously is not yet supported. Since it appears that virtually no one knows that one can use stacked SPL default environments anyway, this is probably not overly problematic at this time.
- The argument completion facility can be fairly slow if large resource sets have been loaded and the machine being run on is not the fastest. There is accordingly an option for disabling completion (see 5.1).

The following problems can be encountered with non-Allegro use of the system.

- Some CLIM releases (e.g., Lucid CLIM) produce a header for hardcopy postscript files that may not be directly appropriate for printing. Lucid CLIM produces

  ```
  %! nonconforming
  %%Creator: CLIM 1.0
  %%DocumentFonts: (atend)
  ```

  The first line of this should be simply edited to make it palatable for a printer, e.g.:

  ```
  %!PS-Adobe-2.0
  %%Creator: CLIM 1.0
  %%DocumentFonts: (atend)
  ```

- There are some occasional incompatibilities left in the Lucid CLIM-2 version that can result in the window manager throwing control to the Lisp debugger: the `abort` option offered in the Lisp listener generally allows one to continue.
- The entire window interface can freeze under Lucid CLIM-1 (SunOS 4). This behaviour has not been traceable to any particular cause. Use of the system with this somewhat aged configuration is, however, certainly not recommended.

[3] This includes compatible Allegro Common Lisps as available for PCs. Interested parties should contact the author.

1.7 Troubleshooting

If serious errors occur during the loading of Loom, check that Loom has been previously *compiled*. KPML will not try to compile Loom itself, but loading a noncompiled Loom version will fail.

If installation appears to have been completed successfully, but examples do not generate the intended strings, then some component of the linguistic resources has not been loaded appropriately. There are some systematic failures that can be indicative of particular causes. Most typical is that all SPL inputs result only in nominal phrases being generated: this is usually due to the Upper Model not having been loaded during the configuration phase.

If Emacs and Allegro Common Lisp are not being used, then error conditions can cause more than one process to use the originating Lisp listener simultaneously! The user must ensure that the required input makes it way to the appropriate process (e.g., by repeating it until accepted).

If the interface is up and running but after selection of some command it stops reacting, then check:

1. that all of the KPML windows are 'open' or 'expanded'—if a window is 'closed' or 'iconized', menus dependent on that window will not be brought up until the window is open;
2. that no error condition (e.g., a network or X-server fault) has thrown control back to the calling Lisp process.

It is possible that some unfortunately syntactically misformed resource definitions that escape detection on loading can bring the interface to a halt if a request is made to inspect them. Since inspection can only take place in the *Inspector* window (Chapter 6), it is generally only this window that is affected. Should this occur, there is no available option for continuing, and control-Z from the interface fails to abort, then the *Inspector* can be restarted by typing at the originating Lisp listener (cf. Section 14.6):

```
(kpml-i::startup-resource-inspector-frame T)
```

Chapter 2

Computational Systemic-Functional Linguistics

This chapter offers a generic overview of computational systemic functional linguistics. It first presents how the linguistic system as a whole is conceived; this is the model for all aspects of the generic system and so is the foundation on which all decisions, including many 'implementational' decisions, are made. Following this, it introduces an organization for thinking about the relationship between theory, description, and implementation. This should make it easier to see where one should look for details of particular aspects of the generic system.

2.1 The linguistic system

This generic account of computational systemic-functional linguistic (SFL) systems begins with the structure and organization of the *linguistic system*. This is crucial for understanding every aspect of the computational system. We also use it below to more finely articulate what levels of description are available to us computationally.

2.1.1 Depth and Breadth

SFL conceives of language as a resource for making and expressing meanings—a potential for making meaning, or 'meaning potential' for short. This resource is interpreted (i) as a multi-functional system and (ii) as a multi-stratal system of systems; we describe what this entails for a linguistic account in the following two subsections. We then go on to illustrate how linguistic descriptions are represented in SFL and show how this is particularly suited for use as a resource for uncovering the kinds of information and processes that are necessary for controlling linguistic resources.

2.1.1.1 Stratal organization

The context-embedded system of language as a whole is in SFL organized as a resource for making and exchanging meanings. It is organized in such a way that it can construe the symbolic 'gap' between high-level communicative goals in the context of communication and expressions in speech or writing; it construes this 'gap' through organization into a series of *stratified* subsystems — from the most abstract stratum of semantics to the least abstract level of expression, the resources for expressing the grammar's wordings in writing (graphology) or speech (phonology). Each stratal subsystem is organized in such a way that it can relate to its immediate stratal environment: it is organized as inter-related strategic options available to the next higher stratum. Thus, any given stratum is contextualized by the immediately higher stratum — the higher stratum provides the functional motivation for the lower one; and the lower one provides the resources to realize the higher stratum. This is crucial in the design of a multilingual system: languages may have a fair amount of functional commonality at one stratum but diverge with respect to the realization at the stratum next below.

So far three levels of abstraction in the resources that make up language have received extensive computational treatments — a higher level that supports processing global to a text (semantics), a lower level that supports more local text processing (lexicogrammar), and an intermediate level that serves as an interface between the former two (the chooser and inquiry interface). These three levels constitute three strata in a stratal theory of language in context such as systemic functional theory or stratificational theory:

- the highest stratum — the semantic environment: higher-level meanings that provide the semantic environment for any text, and the principal means of relating upwards to context;
- the intermediate stratum — the semantic interface: the semantic interface resources for relating these higher-level meanings to the grammar;
- the lowest stratum — the lexicogrammar: the grammatical resources for wording the meanings, for expressing them lexically and structurally.

We therefore prefer a rather more finely differentiated stratification than that typical in computational linguistics and give full stratal status to the relations defined between semantics and grammar. This is both linguistically necessary and practically useful. Emele, Heid, Momma & Zajac (1992) demonstrate that the sheer diversity of interactions between distinct kinds of linguistic information is guaranteed to defeat any staged approach to generation/understanding that successively maps between levels of representation. A highly differentiated scheme of stratification then simplifies inter-stratal mappings and provides maximal support for the necessarily simultaneous resolution of constraints drawn from multiple levels of representation. This becomes increasingly important the further one moves away from toy research prototypes.

2.1.1.2 Metafunctions

All three of these strata are concerned with meaning. This is reflected in their functional organization in various ways. Most generally, this can be seen in the functional diversity of the resources associated with each stratum. SFL traditionally diversifies the functional spectrum into three highly generalized *metafunctions*, to which any use of language is necessarily simultaneously responsive. They constitute fundamental principles of linguistic organization and are thus embodied in the linguistic system. Each makes its own contribution. The three metafunctions, the *ideational*, the *interpersonal*, and the *textual*, have been described extensively in the SFL literature (e.g., Halliday 1978, Halliday 1985) but, for present purposes, may be simply glossed as follows.

- **Ideational**: the means we have of representing the world to ourselves; it largely corresponds to what has been termed 'propositional content'. It is, as the name suggests, concerned with 'ideation'. It provides the speaker with the resources for interpreting and representing 'reality'. There are two ideational subtypes, the 'experiential' metafunction and the 'logical' one. The former is a mode of ideation that construes experience in terms of particular components and subcomponents. It is the mode of organization of, e.g., the TRANSITIVITY structure of the clause (configurations of transitivity functions, such as Actor (*she*) + Process (*gave*) + Recipient (*to the poor*). The latter is a highly generalized mode of ideation that operates in terms of very general relations such as modification. It is the mode of organization for creating complexes of various kinds, such as coordinate and appositional structures, which are chains of interdependent elements (rather than configurations of constituent components).

- **Interpersonal**: the range of meaning concerned with the expression of social relationships and speakers' attitudes and evaluations. It provides the speaker with the resources for creating and maintaining interpersonal relations with the listener, e.g., by assigning speech roles such as questioner and (intended) answerer and by intruding into the speech situation by giving or demanding comments on what is being said. These resources are represented in the grammar of the clause as MOOD, MODALITY, and other types of interpersonal assessments. For instance, independent clauses in English are organized into Mood (e.g., *he will*) + Residue (e.g., *leave tomorrow*). The Mood element consists of Subject and Finite verbal element and reflects MOOD selections; thus Subject preceding Finite realizes the selection of declarative (*he will*), whereas Finite before Subject realizes yes-no interrogative (*will he*).

- **Textual**: the resources responsible for making language appropriate to its particular context of use, including resources that support the connectivity and coherence of text. It provides the speaker with the resources for contextualizing the ideational and interpersonal information to be presented. We will see extensive illustrations of the particular resources available below.

These three metafunctional components within lexicogrammar and the semantic interface relate to three functionally distinct bases of support within the highest stratum. The semantic environmental manifestations of the metafunctions are the *ideation base*, the *interaction base*, and the *text base*. We use the term 'ideation' base in preference to the traditional 'knowledge base' since it makes the functional position, or 'address' (i.e., the intersection of the semantic stratum with the ideational metafunction, which we shall write as 'semantics×ideational') of the base explicit. The interaction base (semantics×interpersonal) is then concerned, among other things, with the social and epistemic relationship between speaker and listener; it subsumes the notion of 'user model'. The text base (semantics×textual) is concerned with, among other things, rhetorical relations, newsworthiness, identifiability and thematic progression in text; it subsumes the notion of 'discourse model'. In more detail, the semantic environment of the lexicogrammar is organized into:

- **the ideation base**, which is a theory of 'reality' — what one might call a *semanticization* of the world.[1] This is a part of the context that supports 'ideation' and hence the ideational component of the lexicogrammar, i.e., a particular interpretation of the world. The phenomena of the world are ranked and are organized into networks — taxonomies of sequences, process configurations, and simple phenomena. They are interpreted as units with a functional type of structure. For instance, process configurations are configurations of a nuclear process, participants, and circumstances — processes of doing and happening, of sensing, of saying, and of being and having.
- **The interaction base**, which is a theory of (symbolic) interaction and role relationships. This is the part of the context that supports linguistic interaction or exchange and hence the interpersonal component of the grammar, i.e., the speaker's assignment of linguistic role-relationships, the speaker's evaluations, attitudes, and so on. In some respects, we can think of the interaction base as different interpersonal colourings superimposed on the ideation base.
- **The text base**, which is a theory of information as text. This is the part of the context that supports the presentation of information from the ideation base and the interaction base as text in context.

The contents of the linguistic system are thus cross-classified along two dimensions: stratal 'height' and metafunctional 'breadth'; this is summarized in Figure 2.1.

2.1.1.3 Functional Regions

While the metafunctions provide a general division of linguistic resources, this is not sufficiently fine for usefully manipulating large scale linguistic resources. For this reason, within each metafunction, linguistic resources are further divided into *functional regions*. A functional region is a subset of the resources that are concerned with a single 'semantic/functional' area. A lexicogrammar then typically divides into 30 or more functional regions, each of which is responsible for expressing some particular aspect of the functional distinctions made by the stratum above. Organizing a grammar by 'rank' (see below) and 'region' then provides an overall 'map' of the linguistic system which can be used to focus in on areas of interest. The regions can be seen as a kind of meta-network imposed over the base-level network of linguistic resources. KPML strongly encourages orientation to regions as a basic means of finding one's way about in large-scale resources.

2.1.2 Intra-stratal organization: choice and delicacy; structural realization

All stratal subsystems have the same general principles of organization. Since these principles always reoccur, at all the differing scales, or strata, in the system, we refer to them as *fractal*

[1]As a complement to the notion of a conceptualization, if we take the ideation base to be a meaning base rather than a knowledge base — this is described further in Matthiessen & Bateman (1991).

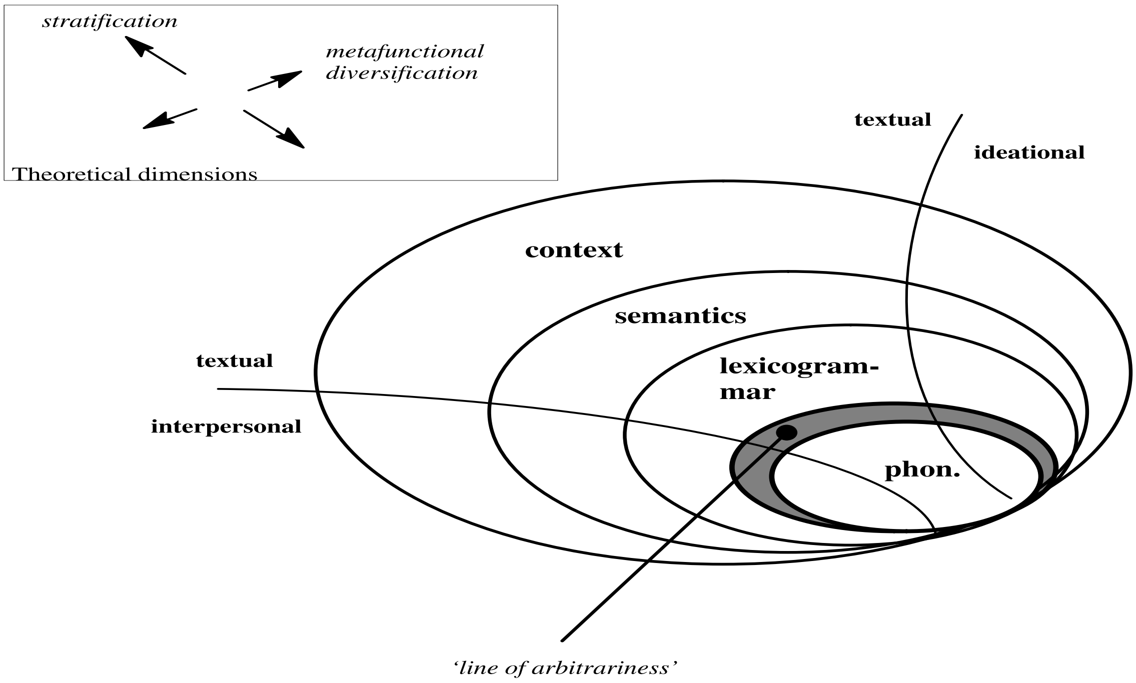


Figure 2.1: Stratal organization of linguistic resources

principles. The fundamental fractal principle is that of taxonomic *choice*. This has often been obscured by the fact that different 'notations' are usually used for the ideation or knowledge base and for the lexicogrammar — for example, frame-based inheritance networks such as those found in KL-ONE and the systemic networks of systemic grammar. We emphasize the similarity in the organization because any solution to the problem of integrating multiple languages in the resources that has been worked out for one stratal subsystem has implications for the others. That is, the issue of how to represent commonality and difference in choice is general across the whole system of language in context, including for example, the ideation base and the grammar. To bring out the commonalities, we will first characterize the common organization principle without committing to any particular notational representation for encoding the information in the ideation base or the grammar.

Both the ideation base and the grammar are organized as a network of types that form a taxonomic hierarchy (known variously as a concept hierarchy, subsumption lattice or system network). These types are related by Boolean operators: a given type may be a subtype of a single type, a conjunction of other types or a disjunction of other types. For our purposes here, types are distinguished in terms of structural properties: each type may have structural consequences — a configuration of roles (slots, attributes, functions). Each role may be restricted as to what type can serve that role (value restriction, type, preselection). The table in Table 2.1 summarizes the manifestation of the general organization just sketched in frame-based inheritance networks used for representing 'knowledge' and in system networks used for representing lexical and grammatical information.

We adopt the system network representation as our general representation medium and have extended the notation for multilingual representation—it now captures what is required of a

	ideation base	**lexicogrammar**
basic notation	frame-based inheritance network	system network
nodes	concept frames (with roles)	grammatical features (with associated realization statements)
network logic	Boolean – classes	Boolean – systems (with input and output grammatical features)
relation between node and structure specification	identical – concept frame	grammatical feature – realized by specifications of structure fragments
unit: structure	concept frame: configuration of roles	grammatical unit: configurations of grammatical functions
role restriction	value restriction of role fillers	preselection of grammatical functions

Table 2.1: Comparison of representation schemes

multilingual representation generally. Our selection of multilingual system networks as the basic representational resource mirrors corresponding attempts to use a single formalism, such as typed feature structures (cf. Carpenter 1992) or 'infons' (cf. Devlin 1990), for all types of linguistic information (as done in, for example, Pollard & Sag (1987), Zajac (1992*a*), Kameyama, Ochitani & Peters (1991), Nerbonne (1992) and others). We draw a distinction, however, between the linguistic theoretical level of description (at which systemic networks appear) and the representation theoretical level (at which typed feature structures or infons appear). We present this in more detail below.

The specification of taxonomically organized choices as a hierarchy of alternative types constitutes a potential. Any type may be instantiated as a token — an actual concept or an actual grammatical unit such as a clause. In the ideation base of a generation system, tokens are stored as 'instantial knowledge' (or assertional knowledge, contrasting with the type of knowledge sometimes called terminological; see, e.g., Brachman & Schmolze (1985) for a standard description) — particular facts about particular individuals at particular times, etc. In the grammar, tokens are not stored — they are only created in the process of generation/understanding: particular paths through the taxonomic hierarchy and instantial wordings. That is, the system stores instantial meaning in the ideation base but not in the grammar.[2]

We can now describe the fractal dimensions of the linguistic theoretical level in more detail. The most important are **axis**, **rank** and **delicacy**.

The dimension of axis separates the strategic, taxonomic organization within a stratum as choice — as represented by system networks — and the tactic realizations of choices — as represented in realization statements. The former gives rise to *paradigmatic* descriptions; the latter to *syntagmatic* descriptions. Within the lexicogrammatical stratum, for example, axis separates the network of strategic options available for realizing meanings as wordings and the tactic realization of particular options as fragments of structure. Thus, in English, if a

[2]This is the basic generalization; we do, of course, store instantial wordings — quotes, proverbs, etc.

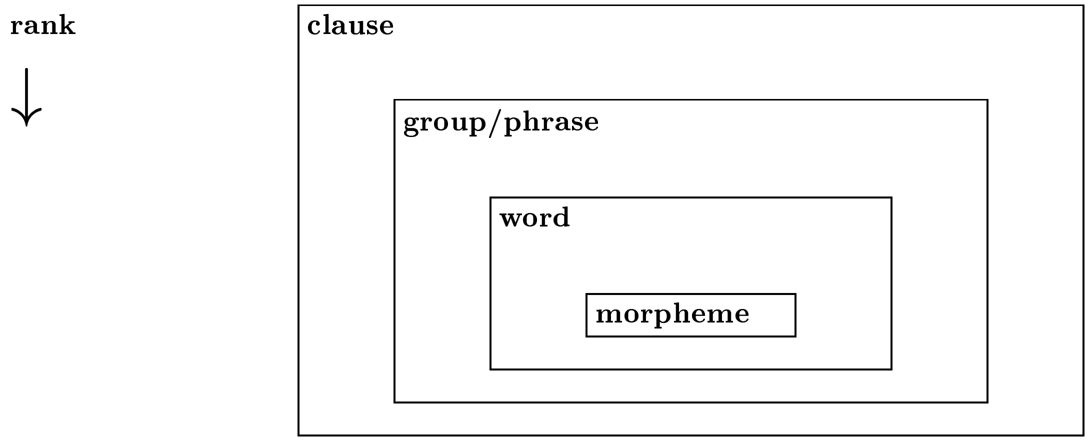


Figure 2.2: Constituency and the rank scale: English lexicogrammar

clause is 'interrogative', there is a further (i.e., more delicate: see below) choice between 'wh-interrogative' and 'yes/no-interrogative'; these latter two systemic options are realized either by the presence of a Wh-element (indicated by the realization statement [+ Wh]), which is ordered before the Finite-element (the finite part of the verb, i.e., that carrying agreement and tense; realization statement: [Wh ^ Finite]), or by ordering the Subject after the Finite-element [Finite ^ Subject] respectively. Crucially, the realization statements are always given in the context of paradigmatic options such as 'wh-interrogative' and 'yes/no-interrogative', and the coherence of the paradigmatic organization is *given preference* over generalizations concerning phrase structure. The paradigmatic orientation is perhaps the central distinctive feature of the architecture overall.

The second dimension of intra-stratal organization, delicacy, orders paradigmatic options with respect to one another. This refers to the dependency between systems in a system network; it corresponds to the subsumption partial ordering in a type lattice representation. The more general options provide the context in which more delicate ones are available.

Finally, the third dimension, rank, refers to the typical constituency potential of a stratum. In English, clauses consist of groups/phrases, which consist of words, which consist of morphemes; thus, the rank scale of the English lexicogrammar is clause, group/phrase, word, and morpheme. Each higher-ranking unit constitutes the context in which units of the rank below serve (see Figure 2.2). Clause, being the highest-ranking unit, is the most transparent gateway to semantics (cf. Halliday 1961).

2.1.3 Inter-stratal organization: interfaces

The basic inter-stratal organization employed is still the framework known as **chooser and inquiry** semantics (Mann 1983*a*). This can be briefly described as follows.

The chooser and inquiry framework for systemic-functional grammar (SFG) arose out of the need to make a text generation system that was modular and re-usable across different contexts and across different computational systems, knowledge representation languages, text planning components, etc. It was necessary to be able to provide semantic control of the grammar component without insisting that a user, or other computational system, be aware of the grammatical distinctions maintained and organized within the grammar. The chooser and inquiry framework provides such a level of semantic control by associating a *chooser* with each grammatical system in the system network. A chooser is a semantic procedure which knows how to make a purposeful choice among the grammatical features of the system with which it is associated. It makes the choice by asking one or more questions, called *inquiries*, concerning parameters that, typically, refer to aspects of the meaning, concepts, etc. that need to be expressed. It is the responsibility of the inquiries to obtain the information relevant for the grammatical decision. As far as the grammar and choosers are concerned, therefore, the inquiries represent oracles which can be relied on to motivate grammatical alternations appropriately for the current communicative goals that need to be accomplished. This is a simpler task than directly requiring a selection of grammatical features, since the choosers and inquiries decompose a single selection among minimal grammatical distinctions into a number of selections among minimal *semantic* distinctions. While the grammatical alternations may not be directly relevant to a component external to the grammar, the semantic distinctions are: this level supplies a situation-independent semantic classification in terms of which a computational system can organize its information for expression in natural language.

The meaning of inquiries can be defined in two ways: either an informal natural language description of the semantic discrimination can be given, or an actual process may be implemented which interrogates a knowledge base, text plan, etc. in order to establish the response appropriate for the particular communicative goal being achieved. In general, the inquiries associated with choosers of systems from the different metafunctions in the grammar need to look to different sources for their responses. Ideational inquiries typically examine the knowledge base or domain model of a computational system (i.e., the ideation base); interpersonal inquiries examine a user model, beliefs, attitudes and intentions (i.e., the interaction base); and textual inquiries examine the text plan and text history (i.e., the text base). Matthiessen (1987) describes the relation between the metafunctions and different kinds of 'support knowledge' that are required in some detail.

One direction of ongoing development is to replace the association of choosers with individual systems by a more general semantic network of inquiries. The arguments for this are presented, with some examples, in Matthiessen & Bateman (1991). Currently released systems still use the individual chooser packages however.

2.2 A generic computational systemic functional system

We now describe the computational instantiation of sets of linguistic resources of the kind described in the previous section. Just as the linguistic system was organized, the organization of a generic computational functional systemic system is also to be described at a number of levels of abstraction. These levels or, to use Christian Matthiessen's suggestive metaphor, *meta-strata*, are motivated by a consideration of semiotic systems as a whole but also relate interestingly to modern practises of software engineering. We separate out the following levels:

- **Theory**: at this level, the theoretical perspective — i.e., the particular questions to be asked, the ways of going about answering them, the view of what kind of phenomenon language is, etc. — is specified. In the present case, we find at this level a statement of what systemic functional linguistics is.

- **Linguistic representation**: here, we find:
  - a **representation** dimension, where we describe the theoretically motivated representational devices available to us for approaching language. In the present case, the most important linguistic representational device is the *systemic network*.
  - a **description** dimension, where we find concrete linguistic descriptions of actual languages and linguistic phenomena, expressed using the representational resources defined.

  Note that this meta-stratum involves *linguists* primarily; there is no necessary connection drawn with computation and the representation adopted is specified only in terms of the theory.

- **Computational representation**: here, we find a re-expression of the previous representational information but in terms which are explicitly computational. The aim would be for *all* information that is expressed at the linguistic representational level to find some corresponding reflex at the computational representational level. This meta-stratum necessary involves *computational linguists* — in fact, we would use the existence of this meta-stratum as a definition of what computational linguistics is. Crucially, as with the relation of theory to linguistic representation, there is assumed to be a *natural realizational relationship* between the meta-strata of linguistic and computational representation.

- **Implementation**: here, we pass a 'line of arbitrariness' in realization and are concerned with how we make the computational representation run as best we can. There is no need at this level to respect modularities defined and used at the higher meta-strata if they do not contribute to desired run-time behavior.

It is crucial to draw the distinction between the linguistic representation meta-stratum and the computational representation meta-stratum since the two are responsive to quite different concerns. Demonstrations such as those of Mellish (1988) or Carpenter (1992, pp27-32) that systemic networks are generally 'equivalent' to type lattice specifications only hold for the representation theoretical level construal of systemic networks. Such interpretations are, however, underconstraining at the linguistic theoretical level and make no criteria available for distinguishing between systemic networks that represent aspects of language and 'arbitrary' networks that appear very unlikely as language descriptions (such as, for example, Brew's (1991) 'systemic' network for 3-SAT). The dimensions of organization that find expression in systemic accounts in general, and in KPML in particular, are all to be construed at the linguistic theoretical level, i.e., at the linguistic representation meta-stratum; it remains for future work to define possible realizations of constructs beyond the basic type lattice organization at the representation theoretical level, although some first steps are presented in Kasper & O'Donnell (1990) and Bateman, Emele & Momma (1992). For more on the differences between a systemic network and, e.g., the subsumption lattices of HPSG at the linguistic theoretical level, see Bateman (1991); for further information about the two levels of theoretical representation considered abstractly, see below. The implementational basis of other levels could similarly be changed without affecting the linguistic representation specifications at all; this reflects a further important principle of modularity.

Although the Penman system (see Penman Project (1989) and Section 2.3) straddled the lower two meta-strata somewhat uncomfortably, future systemic functional systems will move steadily towards respecting this division and so it makes sense to interpret even current systems in its terms. In particular, current considerations for alternative implementations in terms of typed feature structures (cf., e.g., Bateman, Emele & Momma 1992, Henschel 1994) make this division concrete. At present, the grammar definition notation used in Penman-style architectures can be placed at the computational representation meta-stratum, while the Penman code which interprets these and compiles an internal data structure which is 'traversed' as a network, building up internal representations of syntactic descriptions, is best placed at the implementation meta-stratum. We can expect that both the implementation details and the computational representation will change substantially over the next decade, whereas the linguistic representation will probably be extended rather than changed.

It is perhaps useful to bear in mind that the meta-strata are to be considered as being related in a realization relationship, just as the strata of the linguistic system. Thus, each meta-stratum contains a *complete* representation of the linguistic system at the level of abstraction appropriate. A comparison with an idealized view of a generic systemic functional system should clarify the distinctions drawn here. Such a system would consist of:

- theory: systemic-functional linguistics in general,
- linguistic representation: systemic networks used to describe some linguistic phenomena,
- computational representation: statements made in a typed feature structure formalism compiled automatically from the linguistic representation and capable of being executed according to the the abstract semantics of the formalism,
- implementation: efficient implementation of the formalism.

This scheme is shown graphically in Figure 2.3. A representation at the computational representation meta-stratum is intended to correspond to Uszkoreit's (1991) 'declarative specifications' or to Zajac's (1992*b*, p830) 'executable specifications'. They should also be supportive, therefore, of automatic compilation for specific tasks as done, for example, by the compilation of LFG-like grammars in KBMT (Tomita & Carbonell 1986) and in the recent flurry of reports on automatic 'migration' of HPSG resources into various forms (e.g., Götz & Meurers 1995). The main concern is then with the principles of organization of such resources.

2.3 A specific instantiation: the Penman-style architecture

Here we introduce very briefly the Penman-style generation architecture that is also used for the lexicogrammatical and semantic strata supported by KPML.

The approach to generation is *resource-driven*, rather than *instance-driven* (or data-driven). The organization of the systemic network determines the order in which information is gathered and what information is sought. This is managed via the choosers and inquiries as described above.

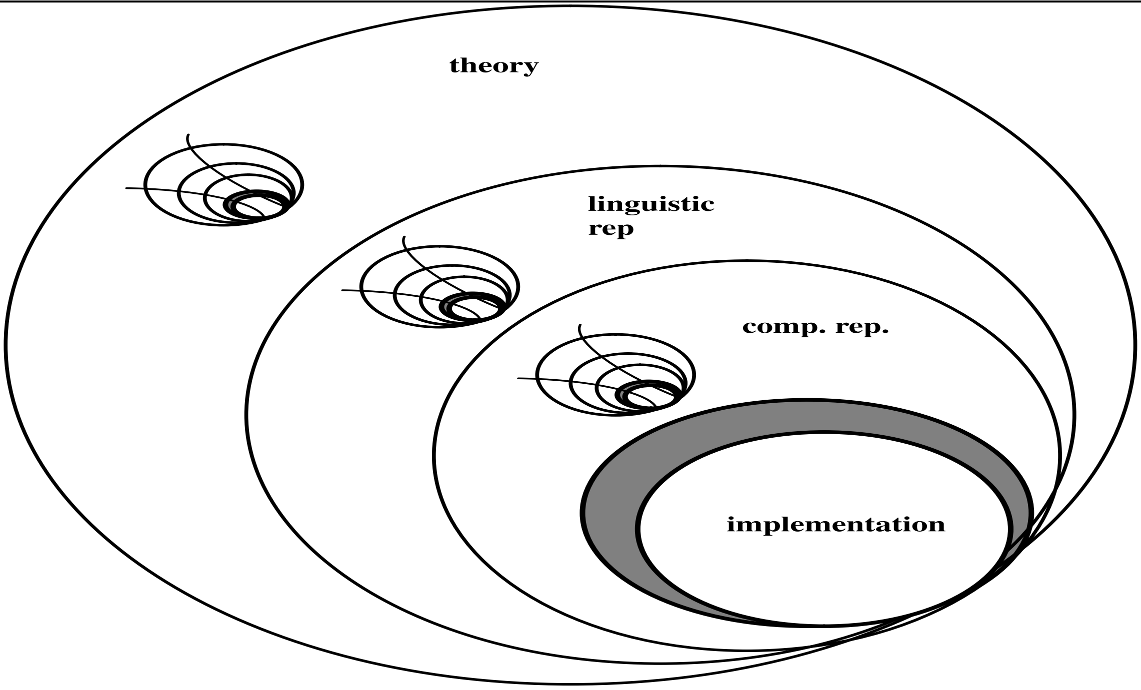


Figure 2.3: Meta-stratal organization of the computational systemic functional account

The architecture is shown in graphical form in Figure 2.4, with the flow of information indicated by broken gray arrows.

2.3.1 The generation process: overview

This section provides a very brief overview of the generation process depicted in Figure 2.4. For more details, see Mann & Matthiessen (1985), Matthiessen & Bateman (1991). Also described here are some particular details of the basic Penman and KPML style generation strategy.

2.3.1.1 Network traversal

The generation process in a Penman-style architecture such as KPML is as follows. Generation proceeds in cycles of traversal through the defined systemic network. Each grammatical unit that is generated is created by one cycle through the network. The result of traversing the network is a set of selected grammatical features (the 'selection expression') and a corresponding grammatical structure. The grammatical structure is created by resolving all the collected grammatical constraints associated with features of the selection expression. Further cycles (for grammatical subconstituents) are created by constraining a grammatical constituent to require realization involving further features selected from the systemic network. More information about the kinds of grammatical constraints that may be employed is given in Section 12.2.5.

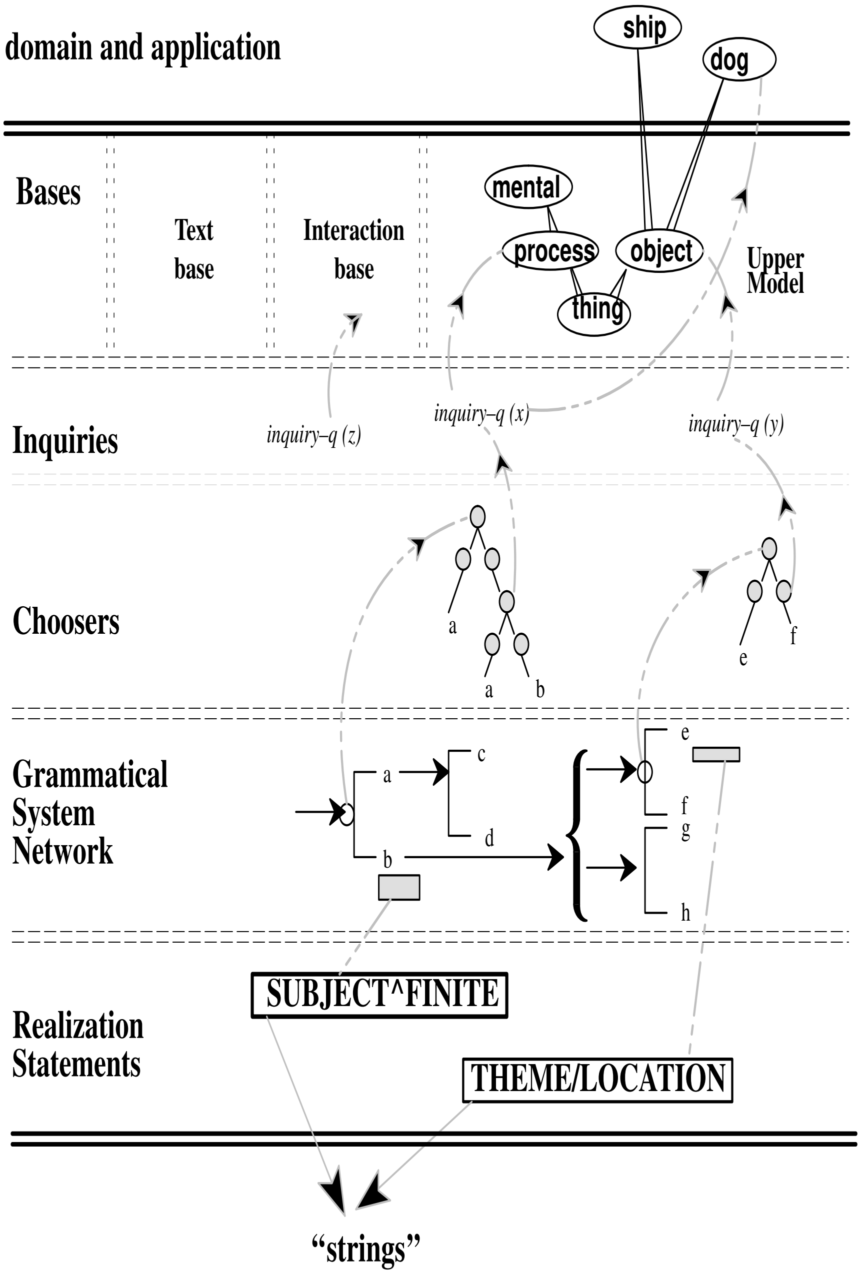


Figure 2.4: Penman-style architecture for lexicogrammar, semantics, and their interrelationships

2.3.1.2 Accessing semantic information

The features that are chosen during traversal of a network are generally selected by virtue of the semantics to be expressed. This is mediated by the chooser and inquiry framework (developed in Mann (1985)). Choosers organize inquiries into 'decision trees', and inquiries are resonsible for (a) inspecting the semantic specification that is being expressed in order to classify that specification along specific semantic dimensions and (b) providing access to particular portions of the semantic specification for triggering further realization. The connection between grammar and semantics is made via a *function association table* that relates grammatical functions (i.e., labels for grammatical constituents defined by the grammar) and semantic 'hubs' (i.e., labels for portions of the semantics to be expressed). Inquiries typically take grammatical functions as arguments, thus providing access to the associated semantic information in a modular fashion. More information is provided in Section 12.2.7.

The usual semantic organization adopted in the Penman-style architecture, and when using KPML, is an *Upper Model*. All of the KPML resources are defined so that generation is possible with respect to a single Upper Model. This provides the concrete instantiation of the *ideation base* introduced above.

2.3.1.3 Stopping traversal: bottoming out

Cycles of generation will continue for all sub-constituents of a grammatical unit until all sub-constituents are filled by some specific linguistic substance—typically lexemes or morphemes. Thus, one possible error is an infinite regression caused by underconstraining some grammatical constituent.

In KPML there are four main ways by which a grammatical constituent may be sufficiently specified as to receive lexical material as its realization and so not to trigger a further cycle through the grammar:

1. an explicit lexical entry can be selected for realization (with the realization statement: `lexify` (Section 12.2.5),
2. a set of lexical features can be associated with a grammatical constituent (by means of the `classify` realization constraint: Section 12.2.5); on completion of a traversal through the grammar, the complete collection of lexical features for a grammatical constituent is used to pick a matching lexical item (i.e., a lexical item whose lexical features unify),
3. an explicit lexicalization on semantic grounds can be asked for by invoking the inquiry `term-resolve-id`.
4. an explicit selection of a morpheme can be made with the morphological realization operators: `preselect-substance`, `preselect-substance-as-stem`, or `preselect-substance-as-property` (Section 12.2.5.4).

Note: if a constituent has been classified, then the selection of a lexical item as described in (2) above will *not respect any additional information*—it is a purely lexicogrammar internal selection. That is, no semantic information or SPL information will be consulted. If the user wants semantic information to be taken into account then option (3) must be taken by including the `term-resolve-id` inquiry in some chooser that is activated at an appropriate point during generation (cf. Section 12.4.1).

2.4 Pointers to further information

Check for the existence of a more extensive WWW-page giving further documentation pointers as well as more general information.

We can now describe the documentation available to a user of a generic systemic-functional computational system in terms of which module of the system is described. This can be done not only for each module of linguistic resources, but also for each meta-stratum at which the module exists. Each level of abstraction and each component with each level has distinct documentation corresponding to its differing concerns. Moreover, any additions and modifications to the framework should position themselves explicitly with respect to this organization, since it is only by doing this that the issues and design criteria can be defined. The dangerous tendency of mixing the linguistic and computational meta-strata should be avoided.

An overview of the documentation and its assignment to modules is given in Figure 2.5. Thus, the following documents together form the basis of a documentation of the generic computational system.[3]

Document Area 1: systemic theory John Bateman. 1992. "Systemic Grammar". *Encyclopedia of AI.*

Christian Matthiessen and M.A.K. Halliday. 1994. "Systemic Functional Grammar: a first step into the theory".

Document Area 2: ideational semantics John Bateman, Bob Kasper, Johanna Moore and Richard Whitney. 1990. "A general organization of knowledge for natural language processing: the Penman upper model." ISI Penman note.

Renate Henschel. 1994. "Merging the English and German Upper Model." *Arbeitspapiere der GMD*, **848**. Sankt Augustin, Germany.

Renate Henschel and John Bateman. 1994. "The merged upper model: a linguistic ontology for German and English". *Proceedings of COLING '94.*

John Bateman, Renate Henschel and Fabio Rinaldi. 1995. "Generalized Upper Model 2.0: documentation". Technical report. GMD/Institut für Integrierte Publikations- und Informationssysteme, Darmstadt, Germany.
URL = `http://www.darmstadt.gmd.de/publish/komet/gen-um/newUM.html`.

Halliday, Michael A.K. and Christian M.I.M. Matthiessen, *Construing experience through meaning: a language-based approach to cognition.* Berlin: de Gruyter, to appear.

[3]There are many more documents covering areas such as grammar and semantics; those listed here are those of particular relevance to the linguistic resources currently available computationally.

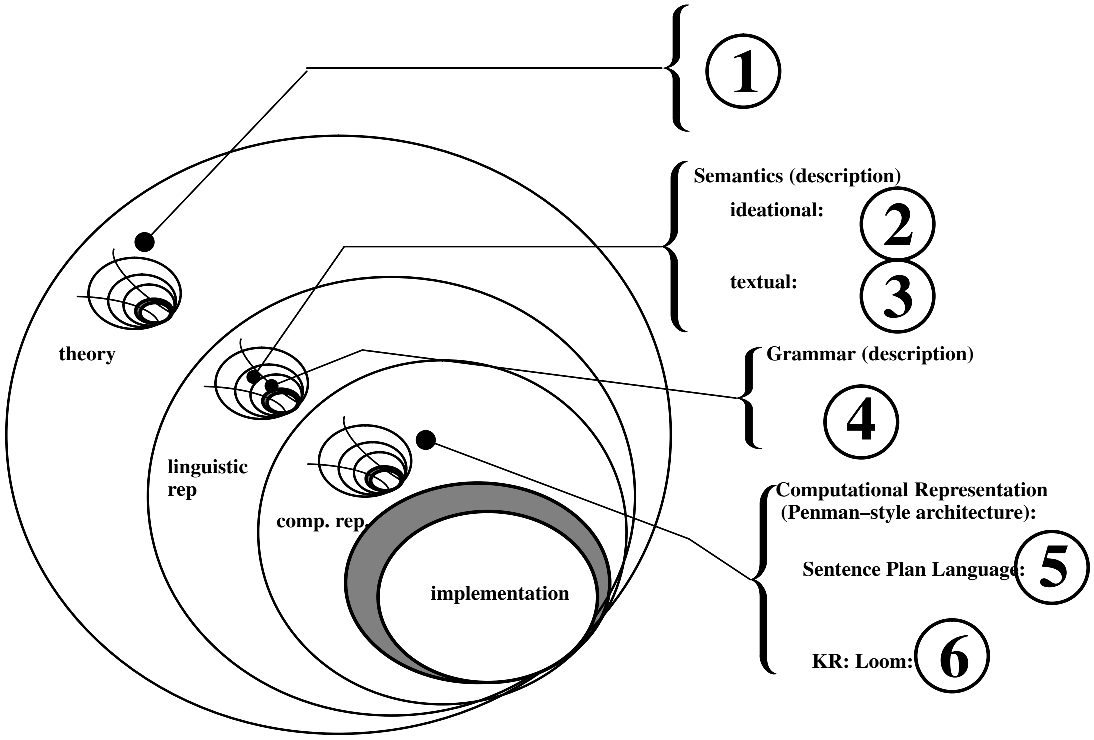


Figure 2.5: Further documentation map

Document area 3: textual semantics John Bateman. 1993. "Nigel: textual semantics documentation". Technical report. GMD/Institut für Integrierte Publikations- und Informationssysteme, Darmstadt, Germany.

John Bateman and Christian Matthiessen. "Uncovering the text base". In: Keqi Hao, Hermann Bluhme and Renzhi Li (eds.), *Proceedings of the International Conference on Texts and Language Research (29-31 March 1989, Xi'an, China)*, pp3–45, Xi'an Jiaotong University Press, 1993.

Christian Matthiessen, "Interpreting the textual metafunction". Linguistics Department, University of Sydney. 1992.

Document area 4: grammar Christian Matthiessen. 1995. *Lexicogrammatical Cartography.* Tokyo, Tapei and Dallas: International Language Sciences Publishers.

Elke Teich. 1992. "Komet grammar of German". Technical report. GMD/Institut für Integrierte Publikations- und Informationssysteme, Darmstadt, Germany.

Liesbeth Degand. 1993. "Dutch Grammar Documentation". Technical report. GMD / Institut für Integrierte Publikations- und Informationssysteme, Darmstadt, Germany.

Bernhard Hauser. 1995. "Multilinguale Textgenerierung am Beispiel des Japanischen". Technische Hochschule Darmstadt, Diplomarbeit.

Document area 5: semantic interface Robert T. Kasper. 1989. "A flexible interface for linking applications to penman's sentence generator". *Proceedings of the DARPA Workshop on Speech and Natural Language.*

The 'Penman Primer', USC/ISI Technical Report (contains several annotated examples of the use of SPL for English generation).

Document area 6: knowledge representation Bob MacGregor. 1995 "The LOOM 2.0 Manual". USC/ISI Technical Report.

In case of difficulties, the unpublished documents can be sent on request. It is, of course, also possible to focus on particular areas of interest by referring to the overall map of documentation concerns set out in Figure 2.5. The documentation is being steadily extended.

Chapter 3

Installation and Startup

The current release version of the KPML system will normally be made available as a single compressed Unix `tar` file containing the software of the system. Sets of linguistic resources for the system are available as separate tar-files.[1] It is possible to update the system without affecting locally developed resources and to obtain linguistic resource updates without having to reinstall the system. The system and the resources should be seen as two conceptually independent components.

Note: the current instructions and released version is compatible for installation with Allegro Common Lisp versions 4.2 and 4.3 with Clim 2.0 and 2.1, and (with reduced eventual functionality: see Chapter 8) Lucid Common Lisp versions 4.1 and 4.2.1 with Clim 1.0 and 2.0. Only the Allegro release is fully supported at the time this document was produced. For *newer* Allegro or Clim releases consult the ftp-directory for a help file that may contain revised instructions for loading.

3.1 Installing the KPML system

The software of the system is to be found in a file '`KPML-`$m.i$`.tar.Z`', where m is the major release number and i is the most recent update within this that is being released.

Restoring the contents of this file will produce a directory structure rooted in the directory `KPML`. One file in that directory—`KPML-INSTALLATION.lisp`—should be edited in order to inform the system of its current placement within the user's directory structure: this is done by changing the pathnames assigned to three global variables. The points to change in '`KPML-INSTALLATION.lisp`' are clearly marked. The first gives the address of the KPML system in its new installation, the second gives the address where LOOM can be found (see below), and the third gives where the linguistic resources are maintained. The latter directory can also be given when KPML is running by using the *Environment Directories* command from the KPML window interface (see Sections 5.4.1 and 12.1).

Note: the KPML system assumes that it is using the Loom knowledge representation system; this should therefore also be installed prior to attempting installation

[1] Separate arrangements will be made for other platforms as necessary.

of KPML.[2] LOOM is available free of charge from USC/ISI; the versions of LOOM currently supported are *both* LOOM 2.0 and 2.1. KPML can then be started with or without a loaded version of Loom present in the Lisp world; if it is not present, the standard KPML startup functions will attempt to load it, assuming that there is already a compiled version to be found.[3] The version of LOOM taken will be either the one in the image, or the one reached by the specified pathname (see next paragraph); no additional information needs to be given to KPML.

Various additional files may be included in the top level `KPML` directory; these should typically be left there. Of these files the following are essential:

- `defsys.lisp`: this contains the system definitions (modules and their component files) for KPML,
- `kpml-package-def.lisp`: this contains the definition of the Lisp package used for most of the KPML source code. The main package is `kpml`, which, for historical reasons, is a nickname of the package `penman`.
- `emacs-side-interaction.el`: this contains Emacs/Mule commands for interacting with KPML from Emacs/Mule (cf. Section 3.2).

The source code of the KPML system is located under the subdirectory `PROCESSES`.

After making the required changes, loading the file `KPML-INSTALLATION.lisp` asks the user whether the system is to be compiled. If the system is being newly installed, the answer to this question should be yes ('`y`'). Subsequently, when the compiled version of the system has been established, this is no longer necessary. Compiling and loading should be done from a Common Lisp process, ideally where both LOOM and CLIM (Version 1.0 or 2.0) have already been loaded. For Lucid Common Lisp it is also necessary to use the Lisp development system rather than the production system.[4]

During compilation of KPML, the user is asked to decide which components of the system overall should be compiled. This takes place in a short dialogue where the user is asked to answer `y` or `n` to several questions. Following compilation, the user is asked if KPML should be loaded.

During loading of KPML, the user is asked to configure the particular instantiation of the system to be constructed. The decisions here concern whether the window interface is to be included, what set of languages are to be expected, etc. The answers required should be clear from the questions posed. The following configuration paths are possible:

- ***Setup window interface?***: if yes, then a version of the KPML window interface will be compiled/loaded.

[2]For some indication of what is involved in using other knowledge representation systems, see Appendix C.

[3]KPML will *not* try to compile LOOM itself. Problems will arise if one attempts to continue loading KPML without a compiled version of LOOM being available—loading will fail ungracefully if one attempts to use the source LOOM files without compilation. Note that if the LOOM pathnames and directory structure have not been properly set up, then the compiled version of LOOM may fail to be found and the system may attempt (incorrectly) to use the uncompiled source files. This can fail with unpleasant messages such as: `>>Error: The function COPY-EQ-SLOTS is undefined` or similar.

[4]It is possible to install the system without CLIM being present; see the configuration step below.

  – ***Load new style interface?***. This is only an option if KPML is being compiled/loaded under Allegro Common Lisp (at least version 4.2) with CLIM 2.0 present. If this is not the case, this question will not be asked and the old-style interface will be loaded.

  Opting *not* to load the window interface means that the system will provide all output and tracing information directly to the standard output stream as if it were a simple teletype. Chapter 14 describes how to use much of the generation functionality of KPML without the window interface.

- ***Load linguistic resources?***: It is possible to call for linguistic resources to be loaded during the loading phase; this means that it then no longer necessary to load the resources directly from the KPML interface as described below. If this question is answered positively, the following two more specific questions are then *not* asked. The resources that are loaded are by default those found under the `ML-BASELINE` multilingual resource set of the currently released and installed resource set.[5] Note that since loading linguistic resources precludes the following two questions, it is generally essential that the baseline resources *also* contain both upper model and inquiry implementations as required. This is the case for the resource sets `ML-BASELINE` and `GENERAL` but may not be the case for others.

- ***Load general upper model?***: Since most users will need the upper model in place regardless of the linguistic resources they are using, it is possible to load the upper model at this configuration stage. The current upper model is usually to be found in the `Semantics` subdirectory of the `GENERAL` language variety of the current release of the KPML resources. This is necessary for interpretation of the semantic input specifications for the grammatical resources unless the user has redefined the interpretation processes in some way.

- ***Load general inquiry implementations?***: The inquiry implementations are also mostly shared across language resources; therefore, it is also possible to load the general inquiry implementations at this stage. The general inquiry implementations will usually be found in the `Inquiry-implementations` subdirectory of the `GENERAL` language variety of the current release of the KPML resources.

- ***What range of languages is to be maintained?***: provides the initial multilinguality configuration for KPML.

  The consequences of this question and its answer are as follows. At any time KPML is only aware of some finite set of named language varieties. This set is used to define the maximal range of applicability for language resources which have no language conditionalization (see Section 12.3 for details of conditionalization). When language conditionalization is not present, a specification is assumed to hold for *all* languages, where 'all' is defined to be the current set of language varieties known. Therefore, if the initial configuration sets up the known varieties to be English and German, and then an unconditionalized language resource is loaded, this resource will be declared applicable for

[5]It is possible to establish another resource set for loading given a positive answer to this question by setting the optional configuration variable `user::*kpml-initial-resources*` in the `KPML-INSTALLATION.lisp` file. The optional configuration variable `user::*kpml-general-resources*` similarly allows the default resource set for the general upper model and general inquiry implementations to be altered.

```
==================================================================
   CONFIGURING THE KPML LOAD IMAGE
==================================================================

Set up the window interface (this requires CLIM)? y
Load new style interface? y
Load linguistic resources? n
Load general upper model? y
Load general inquiry implementations? y

 What is the range of languages to be maintained?
  (This should be a list, e.g., (:english :german), the first language
   is then taken as the current language; nil defaults to (:english).)
: (:english :german :dutch :french :japanese)
Languages to be maintained: (ENGLISH GERMAN DUTCH FRENCH JAPANESE);
 Current language: ENGLISH

Configuration complete...
==================================================================
```

(User input prompted by question marks or colons.)

Figure 3.1: Example configuration dialogue

English and German only. If then the languages known to KPML are extended to include French (for example by creating a new language resource by inheritance—Section 5.9.3), the original unconditionalized resources *will not then be considered applicable to French.* If, however, the original configuration included French, then when the unconditionalized resources are loaded, they would be considered applicable to French.

An example interaction is shown in Figure 3.1.

Following successful loading, the user can make an image of the system as is, or can select particular sets of linguistic resources to be included in such an image using the multilingual operations described below (Sections 5.7 and 5.9.1).

The recommended sequence for a new installation is therefore as follows.

1. Edit pathnames in '`KPML-INSTALLATION.lisp`'.
2. Enter an appropriate Lisp system with CLOS[6] and CLIM already loaded (LOOM may also be pre-loaded without ill effect.)
3. Load `KPML-INSTALLATION.lisp`.
4. Answer 'yes' to the question should the KPML system be compiled.
5. Configure the system compilation as required.

[6]For newer Lisps, such as Allegro 4.2 and newer, CLOS is already present in the standard Lisp release.

6. Answer 'yes' to the question should the KPML system be loaded.[7]

7. Configure the system as required.

8. (For CLIM-1: If the window interface is being loaded, answer when prompted whether the display is monochrome or color.)

9. Set the current Lisp package to `kpml`, with `(in-package :kpml)`.

10. Start up the window interface with `(kpml-i::startup)`.

11. (For CLIM-2: answer when prompted whether the display is monochrome or color.)

12. If no linguistic resources and no *upper model* have been specified in the configuration stage, then an upper model should normally be loaded; most resources released will rely on some version of an upper model being present.

13. Load desired set of linguistic resources.

14. Make an image of the system for subsequent use (KPML provides its own function for this as described below).

3.2 Installing the Emacs/Mule-interface

KPML under Allegro Common Lisp (4.2 and newer) can provide direct editing facilities using GNU Emacs or GNU Mule (cf. Section 11.5). To do this, the appropriate Emacs commands must be defined. The KPML release directory contains an additional file called: `emacs-side-interaction.el`. This must be loaded into Emacs/Mule to provide the necessary commands.

This can either be done explicitly as required with the Emacs command Meta-X load-file, or automatically whenever Emacs is started by placing an appropriate `(load "../emacs-side-interaction")` into the Emacs initialization file (`.emacs`); this is typically found in the user's home directory. No further action on the KPML-side is required.

Alternatively, the commands can be loaded into Emacs/Mule by issuing the KPML function call `(kpml-i::editing-on)` from the KPML Lisp listener (i.e., an Allegro Lisp listener). This also loads the necessary Emacs command.

3.3 Installing the released linguistic resources

The currently released multilingual resources for use with KPML are to be found in a file '`Rn.tar.Z`'. Where `Rn` is `R1`, `R2`, etc. depending on the current release of the KPML resource set.

[7] In order to spare garbage and also for more reliability if a single image of the system is to be used on various machines, it can be advantageous if the compilation and loading phases are carried out separately rather than during a single Lisp session as described here.

Restoring the contents of this file will produce a directory structure rooted in the directory `Rn`. This directory should be placed appropriately with respect to the directory given by the value of the `*root-of-resources*` global variable edited in `KPML-INSTALLATION.lisp`. As long as this constraint is satisfied, no changes are necessary to the resource files themselves.

The `tar` file for the complete resource set is quite large and it may not always be the case that all languages of the resource set are of interest to any particular user. It is possible to select particular language sets and combinations directly from the resource descriptions reachable from the KPML WWW home page:

URL="`http://www.darmstadt.gmd.de/publish/komet/kpml.html`".

3.4 KPML system version maintenance: PATCHES

From time to time patches will be issued that correct bugs that have been found in the system or which make new facilities available prior to a new full release. Patches will be placed in the ftp directory for the appropriate KPML release; they will also be accessible from the World-Wide Web with details of the patches included. In all cases, patches are obtained by retrieving a compressed `tar` file and placing this in the KPML installation top directory. *No further action is required.* When a KPML image is started, it will automatically install and load the latest patch file that it finds in the top directory. The patch file itself is then removed and subsequent restarts of the image simply load the installed patches.

Patch files all have names of the form:

```
kpml-patchesYYYYMMDD.tar.Z
```

The `YYYYMMDD` gives the date of release of the patch file; new patch files *completely replace* previous patch files. The new patch file always includes all previous patches as well as the new ones.[8]

3.5 Making an executable image of the system

The KPML function `make-kpml-image` is available under Allegro and Lucid Common Lisps. In each case, this function makes an appropriate executable image and leaves it in the file whose name is given (as a string) as argument to the function. Images made in this way will automatically load any released patches on start up, will display the configured state of the system, and enter the window interface if present. The image can also be started as a Lisp subprocess under GNU Emacs, which is the recommended way of working with KPML.

Under any other Lisp, the user should ensure that an up to date version of the patches file has been placed in the KPML top directory. These can be installed (if necessary) and loaded by issuing the KPML function call `(kpml::load-kpml-patches)`. This should be done following

[8] Once installed, it is possible for the knowledgeable user to weed out particular patches, but this is not suggested for normal use.

loading of the KPML system and prior to working with it (including bringing up the window interface).

Following loading of the KPML-patches, site-specific patches/additions may be automatically loaded when starting up an image made with `(make-kpml-image)`. For this to occur, the additions must be placed in a file `site-specifics.lisp` in the top-level KPML directory. Finally, *user-specific* additions, customizations, default working environments, etc. can also be automatically loaded by including a file `kpml-user-specifics.lisp` in the user's home directory.

NOTE: it remains the responsibility of the user to ensure that all such additions are compatible with KPML-updates.

Allegro Common Lisp images of KPML made with `make-kpml-image` also support a command line parameter `-nowindow` which suppresses bringing up the window interface on startup.

3.6 KPML resource maintenance: RESOURCE PATCHES

KPML provides methods for maintaining patch levels for a set of linguistic resources and for constructing patch files automatically on the basis of changes made to linguistic resources during a session with the system. The details of this mechanism are described in Chapter 11 below.

Chapter 4

Notational conventions in this document

In the description of operations available under KPML, we will use the following notation for referring to commands and KPML operations throughout this document.

Basic KPML commands are shown as <Load linguistic resources>; these are generally selected by single mouse-clicks from the appropriate menus. Arguments to such commands are shown in the following font: *RANK*. These arguments may either be presented as menu options or by typing when prompted in the *Command Interaction* window. Subcommands reached by further menus of options are shown separated by colons. Several windows offer command menus. Where necessary, the originating window for a command will be given preceding the command.

For example, the command to show all grammatical systems using a realization statement of preselection concerning the grammatical function 'Thing', which is available in the *Inspector* window, will be indicated thus:

INSPECTOR: <Who can ...: ...preselect *thing*>

This means that the command was given by clicking first on the 'Who can...' menu option in the main command menu of the *Inspector* window (cf. Figure 6.1), then on '...preselect' in the secondary option menu that this brought up, and finally by typing 'thing' in at the *Command Interaction* pane of the Inspector window as prompted.

The possible windows from which commands can be issued explicitly are: ROOT, INSPECTOR, DEVELOPMENT, GRAPH, and HISTORY. There are four subtypes of graphing windows: STRUCTURE-GRAPH, RESOURCE-GRAPH, CHOOSER-GRAPH, and DYNAMIC-TRAVERSAL; and two types of history windows: GENERATION-HISTORY and CUMULATIVE-HISTORY. The subtypes will often only be distinguished if the context does not make the intended type of window clear.[1]

We will also occasionally use 'relative' commands, i.e., commands that assume that the user is already in the context of some submenu. These will be specified with a root command

[1] KPML-e versions include a fifth graph subtype: GENRE-STRUCTURE-GRAPH; this is not described here.

'...'; thus the following command description might be used to describe the same command as above, but assuming that we are already in a discussion about the possible options leading on from the <Who can ...> command.

<...: ... preselect>

Finally, there are also often alternative forms of commands that can be given directly by typing within a *Command Interaction* pane. These will be indicated below by preceding the command name with a colon. Thus, an alternative to the above sequence of mouse clicks is:

INSPECTOR: <:Who can preselect *thing*>

This means that the command 'Who can preselect' was typed directly. Command completion (up to the next word in the command) is provided during entry by typing a space.

Generally, all typed input is terminated by typing a carriage return. Inspection commands that expect parameters can also receive these parameters by mouse-selection of objects visible in the window interface (cf., Chapter 6).

Partially typed in or executed commands can be aborted by typing a control-Z.

When descriptions of Lisp functions, macros, etc. are given, the notation of Steele Jr. (1990) will be adopted for their usage patterns.

Chapter 5

The KPML root interface windows

5.1 Introduction

It is assumed that most interaction between the user of the development environment and the KPML system will be via the window interface. If this is not so, or if CLIM is not available, see Chapter 14 for information about interacting with the system without the interface.

Two styles of window interface are provided: the 'new' and the 'old'. The old-style is that familiar to users of the Penman system or KPML 0.8 and before; it is described in Chapter 8. Selection of style (when available) can only be done during the KPML load up and configuration phase. The rationale for the new-style interface is to provide both the quickest access to the information necessary for debugging and maintenance and the ability to maintain that information on screen at all times and in combination with other necessary information. Also provided are more graphical tools for inspecting the results and process of generation. The new-style interface uses color-differentiation extensively for presenting various kinds of information in combination; use of KPML is therefore recommended on color screens, although, of course, the differentiation will still be visible in monochrome.[1]

The recommended way of using KPML is as a subprocess to GNU Emacs; Emacs should be entered in the normal way, and KPML started in an external process Common Lisp buffer. Instructions for starting such a buffer can probably be found in the documentation of the Lisp system being used.[2] Using some of the extensions to Emacs—such as the GNU Mule system—offers here a variety of further possibilities (cf. Section 12.2.2.3). However, it is also, of course, possible to use KPML directly without Emacs being present.

The KPML system uses the original calling Lisp window for outputting results of commands that are not intended for interactive use. Error conditions that arise and which are not caught by KPML may also occasionally result in control being thrown back to the calling Lisp process. In this event, a restart of the KPML interface (usually one of the presented options for continuing from the error) will suffice for continuing work. For these reasons, it is recommended that the user sets up the screen so that the calling Lisp listener (either an

[1]Where this is not the case—for example, with the red/blue differentiation used for contrasting multilingual systemic resources according to language when presented graphically (cf. Section 6.2.5)—alternative representation-styles are selected.

[2]For Allegro 4.2 or 4.3, for example, see Chapter 14 of the Allegro documentation.

Emacs buffer or an interaction shell) can also be seen somewhere in the background while working with KPML. Such error conditions will generally only arise if the user is developing resources and the definitions are seriously incomplete, or if the window system is disturbed in some way extrinsic to KPML (e.g., by network problems, color palette problems, etc.).

Note: if Emacs and Allegro Common Lisp are not being used, then error conditions can cause more than one process to use the originating Lisp listener simultaneously! The user must ensure that the required input makes it way to the appropriate process (e.g., by repeating it until accepted).

Commands are given either by selecting from menus or by being typed within interaction panes. Generally, all typed input is terminated by typing a carriage return. Partially typed in or executed commands can be aborted by typing a control-Z.

The new-style interface also provides argument completion. Control-? produces a list of possible completions of the string already given; control-/ produces a list of all possible completions where the string given occurs as a *substring*. Commands and arguments being input can be edited using the normal input line editing commands (control-f: forwards a character, control-b: backwards a character, control-e to end of line, control-a to beginning of line, etc.).[3]

The remainder of this chapter describes the 'new' style root interface window. Chapters 6 and 7 then describe the other two main windows of the new style interface: the *Inspector* window window and the *Development* window respectively.

5.2 The 'new-style' root window: starting up

Once the KPML system has been loaded, and as long as it has been configured so as to include the new-style window interface (see Chapter 3), the interface can be started by calling the Lisp function `kpml-i::startup` from the selected Lisp listener (i.e., either an Emacs Common Lisp buffer or a shell). The function takes several optional keyword arguments as indicated by the following.

`kpml-i::startup  &key :reset :demo` [*function*]

When non-nil `:reset` indicates that any existing instances of a KPML window interface are to be replaced. `:demo`, when non-nil, brings up the window interface in demonstration mode: here the size of fonts in windows are made very much larger so that they can be easily seen at some distance from the screen or during overhead projection—many of the window and screen images shown in this documentation were made using the KPML demonstration window mode in order to make them more readable.[4]

The straightforward call to:

[3]Since completion can be slow on some platforms, there is an option under the ROOT:<:Options> command that causes completion to be deactivated.

[4]Note that this is a *destructive* operation. Having started up the window interface in demo mode, it is not then possible to revert to non-demo mode. A similar effect can be obtained by changing the allocated fonts—although this requires that particular known fonts are installed and can only work to best effect if the size of some of the window panes is also altered. This is done automatically by using the demo mode.

```
(kpml-i::startup)
```

is equivalent to the call:

```
(kpml-i::startup :reset T :demo nil)
```

Images made with the `make-kpml-image` function (cf. Section 3.1) will automatically bring up the window interface with default parameters when executed.

The first action of the startup function is to ask the user whether the interface is to be brought up in monochrome or in colour. Restarts of the window interface can change their selection here as the user requires. For example, many of the screendumps reproduced in this document were made in monochrome mode since these can look better when printed in black-and-white.

If no linguistic resources are present on startup, the root window alone will be brought up. If linguistic resources have been loaded, then the *Inspector* and *Development* windows described in Chapters 6 and 7 respectively will also be started automatically.

5.3 The root commands: overview

The root window provides commands for those operations that generally precede or follow working with a set of linguistic resources—such as loading and saving linguistic resources—as well as for selecting between general system behaviour options. The root interface window is shown in Figure 5.1.

The root window consists of 5 panes stacked vertically. From top to bottom these are: the root command menu, the root interaction pane, the root messages pane, the <Launch Development Windows> command button, and the documentation line.

Most, but not all, available commands are shown in the command menu. There are also several additional commands that can be typed directly in the middle interaction pane or selected by mouse-click from the completion menu when available (as shown in the documentation line). These latter are less frequently required commands. The prompt shown in the interaction pane includes an indication of the current language; in the example in the figure, this can be seen to be English.

Once resources have been loaded and the system flags have been set as desired, the development and inspection windows can be started by clicking on the <Launch Development Windows> button. This brings up the two windows described in the following two chapters.

Finally, the documentation line shows at all times the options available by clicking the mouse buttons. Options are shown when applicable for the left (L) button, the middle (M) button, and the right (R) button. In the figure, there is only one option available: clicking right would bring up the complete list of commands possible for input at the interaction pane. Clicking on one of these would then insert it as if typed. To activate it, the user must then type a return.

The root commands group into the following categories. Both those commands available directly via the menu and those that need to be entered at the interaction pane are listed here, differentiated according to the notational conventions given in Chapter 4.

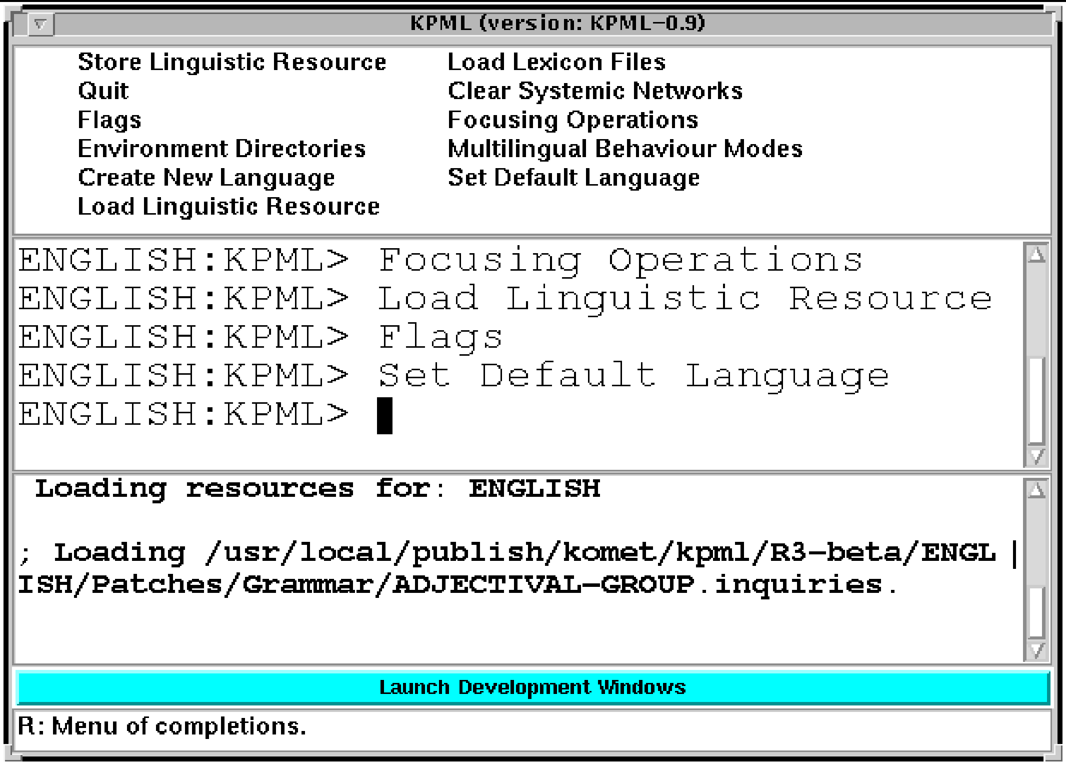


Figure 5.1: The KPML root interface

- General system behaviour (<Flags>, <:Options>, and <Environment Directories>).

- System behaviour particularly concerned with the multilinguality of loaded or stored linguistic resources (<Multilingual Behaviour Modes>, <Set Default Language> , and <Clear Language Varieties>).

- System behaviour during loading or saving particularly concerned with which *types* of linguistic object are to be affected (<Focusing Operations>).

- Resource input/output: including linguistic resource sets as a whole (<Load Linguistic Resource>, <Store Linguistic Resource>, <Create New Language>), lexicons (<Load Lexicon Files>, <:Write Lexicon Files>, <:Clear Lexicons>), and clearing of any loaded systemic linguistic resources (<Clear Systemic Networks>).

- Exit, suspension/activation and clearing of window interface panes (<Quit>, <:Suspend>, <:Activate>, and <:Clear Windows>).

The following sections describe these command groups in detail.

5.4 General System Behaviour

5.4.1 Environment Directories

The <Environment Directories> command brings up a menu for setting/inspecting the environmental file directories that the KPML system uses for various kinds of information access and display. The directories currently maintained here are:

- Root of resources: the directory under which all linguistic resources hang (cf. Section 12.1).
- Hardcopy directory: the directory where postscript versions of graphed information are written when called for—for example, when graphing systemic networks (Section 6.2.1.2), structures (Section 7.9 and 10.2.5), or choosers (Section 6.3.2.2).
- Merging results directory: the directory that records the actions taken when resources are being merged during loading rather than overwritten when the most verbose tracing flags are set (see Section 5.7.2.2).
- Example runner results directory: the directory where the results of attempting to generate selected sets of loaded examples (see Chapter 9) are recorded.

Changing the root directory, for example, is one simple way of creating resources in a new user-specific location—this would be of particular use if different users or developing different resources but using the same installation of KPML.

The starting value for the root directory is that given in the KPML configuration phase. The starting values for the other directories is `/tmp`.

5.4.2 Flags

The command <Flags> brings up a menu containing flags that control general display characteristics of the KPML system. These flags activate or disable:

- display of generated constituency structure: when this flag is set successful generation processes display not only a generated string but also a representation of the grammatical structure underlying that string. The structure is mouse sensitive and can be used for seeking information concerning the generation process. Figure 7.2 shows an example (cf. Section 10.3.2).
- schematic constituency display in generated strings: when set, generated strings are displayed with internal syntactic bracketting enabling selective mouse selection of grammatical constituents (cf. Section 10.3.1). The first output string in Figure 7.1 is an example of the use of this mode.
- restriction of examples offered for generation according to language: when set, the menus of pre-stored examples for generation are restricted only to show those examples defined for the current language (cf. Section 7.4.2).

- automatic acquisition of new lexical items: when set, any new lexical items generated on-the-fly during generation are added to a list of 'new lexemes'. These can then be written out to lexicon files following a session (cf. Section 5.9.4).
- example running results recording to various levels of detail: resource maintenance is generally performed by running test suites. This flag sets the degree of detail in the logs of such test suite runs (cf. Section 10.2.9).
- use of various pop-up windows for showing generation results or for inspecting linguistic objects. In particular, generated strings, selection expressions (i.e., paths of features selected while traversing the systemic networks), and choosers can either be presented in the relevant development or inspector window information panes, or separately in their own pop-up windows. The default behaviour is that selection expressions and choosers are shown in their own windows and generated strings are shown in the *development* window.

5.4.3 Options

The type-in command <:Options> contains several general behavior options that are not typically altered during general use of KPML. These include parameters that may be selected for performance on particular platforms for example. The options list may also include experimental or nonstandard features that are generally disabled but which a user may try out if interested. These include at least the following:

- enabling and disabling font selections (as described in Section 12.2.2.3),
- enabling and disabling command parameter completion,
- enabling and disabling the default for systemic morphology.

For the current most up to date description of the options available, see the World Wide Web KPML documentation.

5.5 General Multilingual Operations and Modes

The first-time user can safely return to this section once familiarity with the default mode operations has been gained.

This section describes the options and effects available under the commands <Multilingual Behaviour Modes>, <Set Default Language>, and <Clear Language Varieties> .

KPML provides some general modes and settings for multilingual operations that apply in some form to almost all operations that the system offers—i.e., to loading, saving, graphing, printing, and generating. These modes extend the flexibility and ease of use of the system particularly when multilingual operations are being performed with any frequency.

All types of multilingual operations on resources can be carried out in three modes:

- **monolingual** mode, where a single monolingual view of a, possibly multilingual, resource is taken,
- **contrastive** mode, where several, usually monolingual, views of a multilingual resource are taken 'side by side', or in parallel,
- **multilingual** mode, where a single multilingual view is taken of some selection of languages (possibly all) drawn from a multilingual resource.

Here, a monolingual resource is understood as one which contains information only about one language variety (whether or not this is indicated by single `in-language` declarations (Section 12.2.1) or by appropriate conditionalization within linguistic unit definitions (Section 12.3)), and a multilingual resource is understood as one which contains information about at least two language varieties.

The modes can be set by selecting the command <Multilingual Behavior Modes>; this brings up a menu of possibilities. The precise consequences of each of the three modes when combined with a given multilingual operation type is set out in the individual sections below. The default behavior on starting up a newly installed instance of KPML is always 'monolingual' in all cases.

In addition, the multilingual modes menu includes options for setting whether linguistic resources are *merged* during loading (see Section 5.7.2.2.2) or not, and for setting whether linguistic resources are by default *cleared* before the loading of a new resource set begins. The defaults are that no merging occurs and that resources are cleared.

As described in the chapter concerning definition formats (Chapter 12), definitions of linguistic objects for loading can contain explicit language conditionalizations. They need not do so, however, in which case the language specification for a linguistic object is taken either from a declaration at the beginning of the file containing the definition or, if no such declaration is present, from the currently known set of languages for which KPML is configured. This behaviour is sometimes not what is required—for example, if a definitions from some resource file are being edited and the changes are being immediately evaluated as typically the case when using KPML combined with Emacs, then these definitions will often *not* contain language conditionalizations because they are relying on the declaration at the beginning of the file. Evaluating the definitions could then place the definition in the wrong language partitions. The <Set Default Language> provides a solution to this problem by setting up a default set of languages for all evaluation contexts where no explicit language conditionalization is given. This is described in more detail in the chapter on resource patching (Chapter 11).

Finally, the command <Clear Language Varieties> offers a way of clearing the accumulated set of language varieties that are currently known to the system. That is, if KPML was initially configured for French and English, then loaded some German and Spanish resources, etc., the currently known list of varieties (in this case at least English, French, German and Spanish) may not be appropriate when loading further resources (perhaps unconditionalized). There may also be a slight processing overhead caused by an unneeded increase in the number of varieties maintained. <Clear Language Varieties> simply sets the known varieties to the empty list. The KPML prompt then reflects this by stating that there is no current language.

5.6 Focusing Operations

The first-time user can safely return to this section once familiarity with the default mode operations has been gained.

This section describes the options and effects available under the command <Focusing Operations>.

5.6.1 Linguistic object focusing

Whereas the default behaviour of the loading and saving operations <Load Linguistic Resource> and <Store Linguistic Resource> is that all linguistic resources of a given language or languages be loaded or saved, this can be more finely controlled by focusing on the types of linguistic object that are of interest.

The command <Focusing Operations: Focus on selected linguistic objects> brings up a menu of the kinds of linguistic objects known to the system. This list contains the following items:

- `:systems`
- `:choosers`
- `:inquiries`
- `:default-orderings`
- `:punctuation`
- `:lexemes`
- `:examples`
- `:inquiry-implementations`
- `:inquiry-defaults`
- `:domains`
- `:properties`
- `:resource-patches`
- `:kpml-lg-specific-patches`

All or any of these items may be selected. Subsequent loading or saving operations will then concern only the linguistic objects of the types selected.

The command <Focusing Operations: Release linguistic object focus> undoes the effect of object focusing, by setting the default list of object considered back to the full set. The full set consists of all the linguistic objects *except* the two patch options.[5]

[5] Patching is not activated as the default behaviour since it changes the operation of several commands and the user needs to be aware of this—cf. Chapter 11.

5.6.2 Language focusing

When one is working with some subset of the languages for which resources are available, it is possible to fix attention to that subset so as to avoid repetitive queries (during, for example, contrastive saving, graphing, generation, etc.) as to which languages are required.

The command <Focusing Operations: Focus on selected languages> brings up a menu of the languages that are known to the system. The user should then select some subset (or all) of the languages offered. These then become the languages that are used in any contrastive or multilingual operations without further user queries.

The effect of language focusing is removed by the command <Focusing Operations: Release language focus>. Giving this command when no languages are focused has no effect.

5.6.3 Region focusing

Region focusing provides a finer selection of particular functional regions (cf. Section 2.1.1.3) within languages. When a set of regions is focused, then only these regions will be effected by loading and saving operations. Region focusing works entirely analogously to language focusing.

5.7 Loading existent linguistic resources

Loading refers to the reading of resource definitions (according to the specifications set out in Chapter 12) maintained in files into the KPML system. The assumed directory organization of these resource files is as described in Section 12.1. Normally, the first operation that will be done when starting up KPML will be to load some set of resources. The default startup loading behavior is monolingual behavior.

5.7.1 Simple resource set loading

Once KPML is installed, loaded, and running, the first operation that typically needs to be performed is to load some already existing set of linguistic resources.

The resources desired need also to have been installed and made accessible to KPML. KPML can access resources when the directory in which the resources are kept has been placed in the global variable `*root-of-resources*`. This can either be done during installation of KPML (see Section 3.3), or at any time by issuing the command <Environment Directories> (see Section 5.4.1).

Following this, the simplest way to load a set of linguistic resources is with the command:

<Load Linguistic Resources>

The languages offered will be those for which KPML has been configured.[6]. This command will

[6] Note that configuring KPML for a given language (Chapter 3) is no guarantee that resources for that language exist!

then cause all available resources for the designated resource set to be loaded. This includes the grammar definition (systems, choosers, and inquiries), any lexica that are defined for the resource set, any examples that are defined for the resource set, punctuation rules, and SPL-defaults/macros for the language variety; for descriptions of all these resource types, see Chapter 12.

Note that explicit language conditionalization given in an input specification *always* takes precedence over any default assumptions or options. That is, if a resource set is called `:english`, but contains explicit conditionalizations for `:german`, then it is these explicit conditionalizations that prevail.

Resource set loading relies on the resources having the organization and internal form also described in Chapter 12. This organization is automatically created and conformed to by any of the KPML commands for saving linguistic resources (Section 5.9.1).

If the resource set is complete (as any of the standardly released resource sets will be), it is then possible to generate with the loaded resources—either from the provided examples or from new semantic specifications given by the user. Generation of an example sentence provided in the resource set is started by the command DEVELOPMENT:<Generate Sentence *EG-n*> where *EG-n* is an example name selected from an offered menu. The first time that a sentence is generated, it will probably be the case that some internal bookkeeping is triggered; this does not then occur again until new resources are loaded. For the details of the generation process see Section 7.4, and for test suite maintenance Chapter 10.

5.7.2 General commands for loading linguistic resources

While the above loading command usage is often sufficient for using KPML, the system provides considerably more functionality for loading linguistic resource sets.

5.7.2.1 Loading particular kinds of linguistic objects

It is possible to indicate exactly what kinds of linguistic objects are to be loaded from any resource set by issuing the command <Multilingual Behaviour Modes: Focus on selected linguistic objects> (Section 5.6.1). When some subset of linguistic objects are focused, any load operation initiated before the focused object set is released is automatically restricted to just those objects that are focused.

Therefore, if a grammar, for example, that of French, was to be kept intact, and it was simply required to load an updated version of, for example, the punctuation rules for that language, then this could be achieved with the following command sequence (making use also of the language focusing commands mentioned in Sections 5.6.2) issued from the ROOT KPML window interface.

<Multilingual Behaviour Modes: Focus on selected language *French*>
<...: Focus on selected linguistic objects *punctuation*>
<Load linguistic resources>
<Multilingual Behaviour Modes: Release Object Focusing>

Note that the command DEVELOPMENT:<Operations on examples: Load examples> is also available for loading examples (Section 10.2.1). This allows the user to select a given example set from the Example directory of the current language, should not all the available example sets be required. Example sets offered in the menu consist of those files with extension `.spl` or `.ex` in the appropriate language directory.

5.7.2.2 Loading modes: overwriting and merging

Two loading modes are provided: *overwriting* and *merging*.

5.7.2.2.1 Overwriting mode When systems, choosers, inquiries, examples and lexical items are loaded for which definitions of identically named entities already exist, these previous definitions are fully replaced by the new ones. No trace of the older ones will survive. When a newly defined entity has a smaller language scope than the entity replaced, then a warning to this effect is given since it means that the previous language resources relying on the old definition may no longer be complete.

Similarly, for punctuation rules, nonsystemic dependencies, and inquiry implementations, the newly loaded resources for a language will replace *all existing definitions for any language.*

Although potentially deleterious for the loaded versions of existing resources, this option can be sensibly used for working on new language development without regard for previous resources. Subsequently, merging can be undertaken using the merging mode for reimporting the debugged resources into the general multilingual potential.

5.7.2.2.2 Merging mode When systems, choosers, inquires, examples and lexical items are loaded for which definitions of identically named entities already exist, these previous definitions are *merged* with the new definitions. The result is a multilingual entity which is equivalent to a set of monolingual definitions. The entity can then be used or inspected from the perspective of any of the languages for which KPML is configured.

Similarly, for punctuation rules, nonsystemic dependencies, and inquiry implementations, the new definitions are added to existing definitions (replacing only any such definitions already existing for the newly loaded language), and definitions of other languages are not affected.

5.7.2.3 Loading and the multilingual modes

The multilingual modes (Section 5.5) intersect with loading to produce the following behaviors.

5.7.2.3.1 Monolingual loading When the monolingual mode for loading is activated, a resource set from a single identified language variety is loaded. For example, issuing a <Load linguistic resources> command prompts for a single language and the resources found under the corresponding directory will be loaded. Monolingual loading takes place in **overwriting** mode; that is, any new definitions possessing the same name as existing definitions cause the existing definitions to be overwritten—*regardless of whether this causes information to be lost* *by removing definitions relevant for other languages!* That is, if there is an existing definition of a grammatical system PROCESS-TYPE that is relevant for the languages English, German and Japanese, and a new system of the same name is loaded monolingually for German, then the previously accessible views of English and Japanese will be lost. If a new system of the same name is loaded monolingually for, for example, Dutch, then the previous views of English, German and Japanese will all be lost. This behavior comes closest to that of the Penman system when loading new definitions.

5.7.2.3.2 Contrastive loading When the contrastive mode for loading is activated, resource sets from several identified language varieties are loaded. For example, issuing a <Load linguistic resources> command in this case prompts not for one but for several languages: resources found under each of the corresponding directories will then be loaded. The order of loading is not specified and should not be significant. Also, although it will generally be the case that the individual resource sets are monolingual, this need not be the case and is not enforced. Contrastive loading provides a convenient way of loading an entire set of distinct resources in one go.

Although resources are cleared before loading commences—as in the case with monolingual loading—contrastive loading takes place in **merging** mode. Here, for any of the selected languages, definitions sharing names with existing definitions will be *merged* with the views of the existing definitions that correspond to languages distinct to the one currently being loaded. Indeed, with this option, overwrite mode would make little sense since it would usually make it the case that information would be lost when each additional resource set were loaded. Thus, asking for contrastive loading of, for example, English, German and French results in a single three-language multilingual resource consisting of the merged monolingual descriptions of each of those languages.

Note that, since the resources loaded are cleared prior to a contrastive load, asking for the contrastive loading of a single language is equivalent to monolingual loading. In order to load a single language into the KPML system in *merging* mode, this mode has to be selected explicitly. This can be done under the <Multilingual Behaviour Modes> command described in Section 5.5. Then, using the example for *monolingual* loading outlined above: in the first case the definitions for English and Japanese will be maintained and only that for German will be replaced; in the second case, all of the information is maintained, the incoming definition for Dutch is simply merged with the existing definitions for English, German and Japanese resulting in a definition for PROCESS-TYPE that allows four distinct language views. This could be of use in successive testing of resources.

Note also that when a system has been disabled for some language during previous loading, then that status remains unchanged unless the system is explicitly reloaded for that language.[7]

Similarly, if a linguistic object belongs to a patch, then that patch status remains unchanged when linguistic objects of the same name but from other languages and possibly of different patch status are loaded.

[7] I.e., loading a system of the same name but for another language will have no effect on the status of the existing definition.

5.7.2.3.3 Multilingual loading When the multilingual mode for loading is activated, a single multilingual resource set from a specified directory is loaded into the KPML system. The normal behavior during multilingual loading is that loading proceeds in **merge** mode; i.e., new definitions replace old definitions just for those languages which are common between the new and the existing resources. If potential 'interference' with existing resources is to be ruled out, then those resources should first be cleared. The directory used for loading multilingual resources can be inspected and set using the <Environment Directories> command (Section 5.4.1).

The multilingual loading option is quite powerful. It makes it possible for a multilingual grammar for, for example, English, German and Dutch developed by one research group to be merged directly with another multilingual grammar for, say, Japanese, Chinese and Thai developed by a distinct research group. The result is then in this case a single six-language multilingual resource from which contrastive views can be extracted as required—for example by inspecting (Chapter 6) or saving (Section 5.9.1) operations. An alternative way of producing the same result would be for each group to extract three monolingual resource sets (contrastive saving, see below) and then to load the resulting six descriptions contrastively. The multilingual option is, however, much faster since the necessary merging operations have already been carried out in the multilingually written files.

Note: there is no guarantee that an 'optimal', or even a 'canonical', merged form is created in any of the options involving merges. All that is guaranteed is functional equivalence of the resources created.

The language varieties used as conditionalizations in a multilingual resource set should all be made known to the system before loading; that is, if the resource set uses conditionalizations for `:french`, `:japanese`, and `:dutch`, then these values *must* be declared as expected language varieties to KPML.[8] Multilingual resources created with KPML will automatically include a declaration of the languages they include (cf. Section 12.2.3).

5.8 Resource clearing

If it is necessary to clear already loaded resources before loading new resource sets, this can be carried out by the <Clear Systemic Network> command. This clears *all* systemic networks and their corresponding choosers and inquiries. Language and region focusing have no effect here.

The command <:Clear Lexicons> similarly clears all lexical items defined; while the command DEVELOPMENT:<Example Operations: Clear Examples> clears all example definitions.

5.9 Saving and Creating linguistic resources

Note that only systemic resources are included in the KPML saving operations: i.e., only systems, choosers, inquiries, lexemes, examples, punctuation, and default orderings. Other definitions (domains, etc.) have to be prepared separately since

[8]From Lisp, pushing the values onto the value of the global variable `kpml::all_languages` is sufficient.

they are typically not systemically defined.

5.9.1 Simple resource set saving

Saving is the operation of exporting the resource definitions held at any time in the KPML system to sets of external files. Saving is the normal operation to be performed after working on the resources or after creating new resources. Saving can be carried out in any of the three multilingual operation modes (Section 5.5). The default startup saving behavior is monolingual saving.

All information concerning systems, choosers and inquiries that is known to the system will be saved to their respective regions, regardless of any originating file structure used in loading that information. In contrast, lexicons and examples are saved back into a directory structure isomorphic to their originating definitions although not necessary in the same directory.

All requests for saving linguistic resources initiated with the <Store linguistic resources> command obey any constraints that may have been set under language, region, and linguistic object focusing as described above.

The following additional commands provide specialized saving commands:

- ROOT:<:Write Lexicon File> – this will pick out the lexical items defined in an identified file and write these and these only back to that file.
- DEVELOPMENT:<Example Operations: Write Examples> – this will write out the current examples as defined (cf. Section 10.2.2).

Note that the saving commands never clear their target directories before saving: the user should therefore exercise care that old and new definitions are not mixed involuntarily. To aid this, the save menu contains an additional flag asking whether a new directory is to be created (regardless of the existence of a previous directory for the language variety at issue) or not. When a new directory is to be created and there is already an existing directory of the same name for that language variety, then the existing directory is copied into a backup directory. The name of the backup directory is the same as the original with the date of creation of the new directory appended.

5.9.2 General commands for saving linguistic resources

5.9.2.1 Monolingual saving

In monolingual saving mode, the user is prompted for a single language selected from those for which KPML is currently configured. A single set of monolingual resources for that language will then be written to files in a directory corresponding to the name of the language variety. The directory will be located underneath the `*root-of-resources*` directory as specified during KPML initialization (Chapter 3) or as subsequently modified by the <Environment Directories> command (Section 5.4.1).

For example, if KPML is working with a loaded multilingual resource including views for English and German, issuing a monolingual *Store linguistic resources* command for German will write out a set of files (three for each functional region, providing the systems, choosers, and inquiries, plus ordering and punctuation information: see Chapter 12) under a directory called `GERMAN`. All of the resource files will be conditionalized exclusively for the single language German.

Resource sets of this kind can then naturally be reloaded separately at any time using the monolingual <Load linguistic resources> command, or as a contributor to a multilingual set using the contrastive load option.

5.9.2.2 Contrastive saving

In contrastive saving mode, the user is prompted for a set of languages selected from those for which KPML is currently configured. The system then performs a monolingual save for each of these languages.

For example, issuing a contrastive <Store linguistic resources> command for English and German results in two directories (called `ENGLISH` and `GERMAN` respectively) being written, each containing a complete *monolingual* conditionalized set of resource definitions.

Resource sets of this kind can then naturally be reloaded in their entirety at any time using the contrastive <Load linguistic resources> command, or as single languages using the monolingual load option.

Note that monolingual conditionalized resource sets are *explicitly marked as being relevant for a given language*. This contrasts with monolingual resource sets which have no language affiliation: see Section 5.9.5.

5.9.2.3 Multilingual saving

When the multilingual mode for loading is activated, a single multilingual resource set is written to a specified directory. The user is prompted for the languages (which can be any subset of the set of languages for which KPML is configured at that time) to be included in that resource set. The resource set contains the appropriate language specific conditionalizations to enable the individual language views to be recovered when required.

For example, if KPML is configured for English, German, Dutch, French and Japanese, then issuing a multilingual <Store linguistic resources> for English, Dutch and Japanese will result in a single three-way multilingual resource set being written to the specified directory. The directory used for saving multilingual resources can be inspected and set using the <Environment Directories> command (Section 5.4.1).

Multilingual resource sets of this kind can be reloaded at any time using the multilingual <Load linguistic resources> command.

5.9.3 Inheriting language definitions

KPML provides one way of creating new linguistic resource sets: a resource set is constructed that is the exact copy of some existing linguistic resources apart from the language conditionalization being altered to refer to some new language. This is triggered by the <Create New Language> command.

This command brings up a menu dialogue which asks the name of the new language variety to be created and the existing language variety from which it is to be created. Following the operation a complete new set of resources for the new language variety based on the definitions of the selected old language is written under the current root of resources, and this language is added to the list of available languages. If the original resource set was complete, then it should be possible to issue a load linguistic resource command on the new language and obtain the same generation results as were obtained with the originating language.

This command can be used as the first stage of creating resources for a new language.

The command is fully sensitive to region and linguistic object focusing.

As always, only systemic resources are included in this saving operation: i.e., only systems, choosers, inquiries, punctuation, and default orderings. Other definitions (domains, etc.) have to be prepared separately.

5.9.4 Automatic lexical item acquisition and saving

When the ROOT:<Flags> option 'automatic acquisition of new lexical items' is set, any undefined lexical items mentioned in `:lex` or `:name` slots in SPL expressions are created with lexical features appropriate for their place of occurence in the realized sentence and with a spelling drawn from the label appearing in the SPL. All such lexical items are placed on the list `*new-lexical-items*` which can then be output to a file of new lexical definitions (presumably after running through a complete batch of examples with the example runner, for example) using the function `make-new-lexical-items-file`.

5.9.5 Creating unconditionalized linguistic resources

From time to time, it may be required to create sets of linguistic resources that are not conditionalized in any way—such as, for example, the resources under GENERAL in the KPML resource releases. Such resources can then be used as seeds for growing further multilingual resources for different languages.

In order to create a set of unconditionalized resources the following steps are necessary.

1. Establish a loaded set of linguistic resources that has the desired behavior when some particular language variety is current.

   That is, we need to specify some language variety that is to serve as the basis for the unconditionalized resources. Since these resources will not specify any language variety, they are equivalent to the resources for one particular language. The first task, therefore, is to create some language variety that has the desired effect.

2. Load this established language variety as the only loaded language and with `all_languages` set to only that language (as a singleton list).

3. Specify that the resource saving mode is `:multilingual`.

4. Issue *in a Lisp listener* the following save command:

   (`save-unconditionalized-linguistic-resources` *:new-resources*
   `:root-directory` *root-directory*
   `:inquiry-package` *new-inquiry-package*)

   Only the first parameter is obligatory; this defines the name of the directory under which the new, unconditionalized resources will appear. The remaining keyword parameters are optional and as for the `save-linguistic-resources` function (Section 14.4.4).

Following this sequence of operations, a new unconditionalized resource set that has the behavior of the originally selected language variety will have been left under the directory `NEW-RESOURCES`, which itself will be under either the default root of resources directory or the directory given in the call to `save-unconditionalized-linguistic-resources`.

Note: currently the creation of unconditionalized default orderings, punctuation, etc. is not done correctly. For the present, simply edit the values found there to remove the conditionalization (which will be for the original language variety specified).

Thus, the lists following the language condition should replace the values given as a whole; i.e., all lists of the form:

```
(:multilingual
  (:english    X))
```

should be replaced simply by `X`. More elegant support for unconditionalized information of this kind will be provided at some stage. The `properties.lisp` file should also be treated with caution; the file of the same name to be found in the GENERAL resources can be taken as a model.

Finally, as is usually the case with resource creation, all definitions not covered by the automatic resource saving operations (e.g., SPL default definitions, domain models, etc.) should be copied as required.

Loading such general unconditionalized resources should, of course, be carried out in *multilingual* loading mode, since the resource is not restricted to any single language but takes on the scope of applicability defined by the range of languages for which KPML is configured at the time of resource loading or the current defaults as created by <Set Default Language>.

5.9.6 Changing the Lisp package of inquiry implementations

This subsection is only relevant for users who intend to be writing extensive inquiry implementations of their own.

For increased modularity of resource development, it may occasionally be appropriate that different bodies of inquiry implementations are maintained in different Lisp packages. This is caused by the fact that such implementations, as simple Lisp functions, lie outside the comprehensive language conditionalization facilities offered by KPML.

Writing of resources is thus extended so that it is possible to set the package from which inquiry implementation codes will be expected. To do this, the variable `*package-for-inquiry-implementations*` must be set to either a string denoting the package or a (dotted pair) association list of languages (as specified in `all_languages`) and such strings. An example of the latter would be:

```
((:english . "ENGLISH-INQUIRIES")(:german . "GI")(:japanese . "J"))
```

If a symbol representing an inquiry implementation in an inquiry definition (see Section 12.2.7) is already from a package that is not the `kpml` package, then this information is preserved *unless* the flag `*force-inquiry-implementation-package?*` is additionally set true. This package setting behavior can by summarized as follows:

if a new package for inquiry implementations (target) is given
then
    if the old package was `kpml`
    then the new target is taken
    else the package stays as it is, unless forced.
else
    package stays as it was.

Use of distinct packages for inquiry implementations is further supported by a new package, called `kpml-kb`, which already includes the normal Penman/KPML-defined SPL, knowledge representation system, and lexicon interface functions as external symbols. It is therefore sufficient for an inquiry implementation file to begin with the declaration:

1. for CLtL1:

   ```
   (in-package "INQS" :use '("LISP" "KPML-KB"))
   ```

2. for CLtL2 (e.g., Allegro 4.2 and later):

   ```
   (defpackage "INQS" (:use "COMMON-LISP" "KPML-KB"))
   (in-package "INQS")
   ```

in order to use both Lisp and the Penman/KPML functions without package specifiers.

Care should be exercised if these inquiries refer to symbols that are maintained in the `kpml` package. In KPML, there is no support for systemic resources (including lexical item definitions) being in distinct packages. Therefore all lexical features, system name, system features, etc. will be in the `kpml` package and so must be referenced appropriately. Use of the functions defined in Section 13.2 whenever reference is to be made to lexical information guarantees that the appropriate package is enforced. Further, all responses to inquiries returned from inquiry implementations will be automatically interned in the `kpml` package regardless of their package of origin.[9]

[9] Actually they are interned in the package identified by the value of `kpml::*current-language-package*`, but in KPML this is always `kpml`.

An example of these package definitions and mechanisms is given by the following inquiry implementation (assuming CLtL2):

```
(defpackage "INQS" (:use "COMMON-LISP" "KPML-KB"))
(in-package "INQS")

(defun Accompaniment-Modification-Q-Code (process)
  "Look for an accompaniment feature."
  (if (fetch-subc-feature 'accompaniment process)
      'accompanying))
```

Here, the function definition is, of course, for the function `komet-inqs::accompaniment-modification-q-code`. However, the symbol `fetch-subc-feature` refers to the SPL interface function of the same name described in Appendix B. All of these interface functions are usable in this way. In KPML, these functions operate on their arguments so that regardless of package of origin, symbols are searched for in the appropriate place (e.g., in the upper model package `penman-kb` , or in the package used for SPL specifications—which is normally `kpml`). Finally, the symbol returned by the function is automatically converted to one belonging to the appropriate package for the inquiry response—since this is the symbol that occurs in the inquiry definition in the `:answerset` slot and in the use of the inquiry in any choosers and so must be made accessible in the package for those definitions.

The full list of Penman/KPML functions and variables (all in the `kpml` package) which are accessible on use of the `kpml-kb` package are as follows.

- The following functions are used by inquiry implementations to access components of SPL expressions and to interrogate the subsumption relations of the knowledge base. Most of them are described in Appendix B and C.

```
FETCH-FEATURE
FETCH-FEATURE-SYMBOL
FETCH-MINIMAL-RELATION
FETCH-NON-MINIMAL-REIFIED-RELATION
FETCH-REIFIED-RELATION
FETCH-RELATION
FETCH-RELATION-RANGE
FETCH-RELATION-SPEC
FETCH-SUBC-FEATURE
FETCH-SUBC-FEATURE-NAME
GET-GLOBAL-TERMS
GET-SYMBOL-TERM
GET-TOP-LEVEL-TERMS
GLOBAL-FETCH-FEATURE
INVERT-REIFIED-RELATION-SPEC
KB-ENTITY?
KB-SUPERP
MAKE-TERM-GRAPH-ID
TERM-EQ-P
TERM-GRAPH-FEATURES
```

```
TERM-GRAPH-ID
TERM-GRAPH-PARENT
TERM-GRAPH-SYMBOL
TERM-GRAPH-TYPE
TERM-ROLE-P
TERM-TO-GRAPH
TERM-TYPE-P
TERM-TYPE
```

- In addition, some of the SPL functions are actually macros and expand to involve other symbols, which, of course, must also be accessible.

```
TERM-GRAPH
ONUS
```

- The following functions are provided by KPML for accessing lexical information during generation. This can be done from inquiry implementations, although this should be understood as standing in for a more appropriate way of transporting features internally to the lexicogrammar. It should also be borne in mind that the lexical features will normally be in the `kpml` package, and so this will need to be explicitly specified for lexical/morphological inquiries residing in a different package, *unless* the built-in KPML access functions are used (e.g., `lexical-feature-present-p`, etc.: Section 13.2); these latter functions ensure that symbols of the appropriate package are used regardless of their originating package.

```
FIND-ASSOCIATION
FUNCTIONAL-ROLE-P
ACCESS-LEXICAL-INFORMATION
LEXICAL-FEATURE-PRESENT-P
LEXICAL-CLASS-ASCERTAINER
LEXICAL-FEATURE-PRESENT-IN-ASSOCIATION-P
LEXICAL-CLASS-OF-ASSOCIATION-ASCERTAINER
LEXICAL-ITEM-FEATURES
LEXICAL-ITEM-SPELLING
LEXICAL-TERM-RESOLUTION
```

- The following functions are former Penman functions that are also used occasionally in inquiry implementations.

```
NP-SPECIFY
OPERATOR-RUN
PLEDGED-P
EXPRESSED-P
WARNING
```

- The following former Penman variables hold necessary information for making internal information accessible concerning the history of the generation process for supporting textual inquiries.

```
*CONSUMED-TERMS*
*PLAN*
*PLAN-GRAPHS*
```

- The following KPML variable conditionalizes some of the morphology inquiry implementations.

```
*ACTIVATE-WORD-RANK-AND-BELOW*
```

- The following is needed for lexicalized information.

```
LEXICON
```

- The following book-keeping inquiries should always be accessible.

```
WHEREAMIIDCODE
TRIVIALDEFAULTCODE
```

5.10 Interface suspension, exiting, etc.

5.10.1 Quiting the interface

The command <Quit> causes all open KPML windows to be destroyed and then exits the interface. Interaction with the system is then still possible in the calling Lisp listener. There a new interface instance can be started or Lisp can be exited in the standard manner (e.g., for Allegro `:exit`, or for Lucid `(quit)`).

5.10.2 Suspending the interface

When the KPML interface is running it normally intercepts all output and messages from the generation process in order to present this information in an appropriate form (e.g., in a particular window pane, in some pop-up window, or as a dialogue menu). Very occasionally, it might be desired to have this information sent to the calling Lisp listener as if the interface were not running. This might be one way, for example, of obtaining a trace of generation that can be edited or printed in hardcopy and studied at length. The command <:Suspend> has the effect of disabling the interface's interception of messages. These messages are then presented as if the interface were not present—i.e., in a pretty printed teletype form.

Note that suspending the interface does *not disable* the interface: it is still possible to issue commands in the normal way. All that is suspended is the presentation of information.

Thus, selecting to trace the system and chooser activity during generation (with the command DEVELOPMENT:<Generation Display Modes>), suspending the interface, and then generating an example would result in output of the form shown in Figure 5.2 being sent to the Lisp listener. From there it can be printed, edited, etc. more readily than its appearance in the KPML interface.

```
⋮

--------------------------------------------------------------------------------
>> Triggering Gate MOOD-UNIT
--------------------------------------------------------------------------------
>> Entering System DEPENDENCE
      Entering Chooser DEPENDENCE-CHOOSER
          ENTIRENESS-Q:   Does DEFAULT-ASSERT-356929   represent the entire
          speech act or is it part of a larger one?
       Environment's answer to the system network is ENTIRE
       Chooser DEPENDENCE-CHOOSER chooses feature INDEPENDENT-CLAUSE.
--------------------------------------------------------------------------------
>> Triggering Gate INDEPENDENT-CLAUSE-SIMPLEX
       Chooser INDEPENDENT-CLAUSE-SIMPLEX-CHOOSER chooses feature
       INDEPENDENT-CLAUSE-SIMPLEX.
--------------------------------------------------------------------------------
>> Entering System INTERNAL-SUBJECT-MATTER
      Entering Chooser INTERNAL-MATTER-CHOOSER
          THEME-MATTER-Q:   In addition to specification of a set of
          processes, a set of participants, and a set of circumstances, does
          S-368638   contain a distinct specification of some conceptual
          context or topic with respect to which these processes,
          participants, and circumstances are to be interpreted?
       Environment's answer to the system network is NOTHEMEMATTER
       Chooser INTERNAL-MATTER-CHOOSER chooses feature
       NONINTERNAL-SUBJECT-MATTER.
--------------------------------------------------------------------------------
>> Entering System MOOD-TYPE
      Entering Chooser MOOD-TYPE-CHOOSER
          COMMAND-Q:   Is the illocutionary point of the surface level speech
          act represented by DEFAULT-ASSERT-356929   a command, i.e. a request
          of an action by the hearer?
       Environment's answer to the system network is NOCOMMAND
       Chooser MOOD-TYPE-CHOOSER chooses feature INDICATIVE.
--------------------------------------------------------------------------------
>> Triggering Gate FINITE-CLAUSE
--------------------------------------------------------------------------------
>> Triggering Gate FINITE-INSERT
       Chooser FINITE-INSERT-CHOOSER chooses feature FINITE-INSERTED.
--------------------------------------------------------------------------------

⋮
```

Figure 5.2: Example non-interface trace of generation

5.10.3 (Re-)Activating the interface

Issuing the command ROOT:<:Activate> undoes the effect of a <:Suspend> command, and information returns to being presented in the interface.

5.10.4 Clearing the interface windows

The command ROOT:<:Clear windows> clears the KPML development, inspection, and any dependent display windows.

Chapter 6

The KPML Inspector Window

KPML offers a wide variety of modes for inspecting the contents of loaded linguistic resources. The most effective way to inspect resources is by a mixture of graphical and textual displays. Larger-scale views are usually obtained graphically, allowing the user to quickly focus in on particular details that are presented textually. The KPML inspector window provides convenient overall access to these inspection methods.

Information inspection within KPML generally involves following ***information chains***: that is, one might know that a particular grammatical system exists, but wants also to know how the grammatical features of that system are actually chosen and under what conditions. This involves following the information chain from system name to corresponding chooser, and from corresponding chooser to corresponding inquiries and their definitions. This might then be followed further to particular knowledge base concepts, which might lead back to lexical items and other points in the grammar. An overview of the information chains possible when examining linguistic resources—particularly the linguistic *potential*—is given in Figure 6.12 below.

The graphical presentation modes can also be used in conjunction with the generation modes described in the following chapter in order to graphically display generation paths: i.e., traversal paths through the systemic networks or choosers, and associations of grammatical and semantic units. These additional capabilities are described under generation tracing (Section 7.5.2). This means that some of the menus reached by clicking as described in this chapter will contain additional options to those described here.

An example of the inspector window is shown in Figure 6.1. There are four panes, from top to bottom: the *Inspector Command* menu, the *Interaction* pane, the *Information* pane, and the mouse documentation line. As usual, for the commands that are entered in the interaction pane, input is terminated by a carriage return. Partially typed in or executed commands can be aborted by typing a control-Z; argument completion is provided by control-? (for string completion) and by control-/ (for completion where the string given occurs as a *substring*). Commands and arguments being input can be edited using the normal input line editing commands (control-f: forwards a character, control-b: backwards a character, control-e to end of line, control-a to beginning of line, etc.).

It is also possible to provide parameters to commands by selecting objects with the mouse. Whenever the inspector window is waiting for parameters to be input to a command (as indicated by the prompt in the *Interaction* pane), most symbols representing linguistic objects (e.g., system names, feature names, lexeme names, etc.) that are displayed in the *Information* pane can be selected by clicking on them. The symbols that can be used are highlighted when the mouse points at them (see, for example, Figure 6.11). They can be included as parameters *by mouse-clicking right* and then selecting the 'pickup object' option. This inserts the symbol as a parameter in the *Interaction* pane, where it can be further edited or simply accepted as the entire parameter by typing return. Note that clicking *left* on a symbol in the Inspector window has its normal effect of presenting the object clicked upon (as described below).

6.1 Overview of Commands

There are four main groups of commands available under the Inspector window. These concern:

- networks and graphical overviews of resources,
- presentations of individual object definitions,
- linguistic object selection according to specific criteria,
- direct inspection of linguistic objects.

The first three groups provide initial steps in information chains where the starting point for the chain is given explicitly by the user by typing in linguistic object names or by selecting from a menu of possible objects. While this mode of information seeking enables all components of the loaded resources to be examined, it is more usually the case during resource maintenance or debugging that very particular information is being sought—for example, information concerned with particular decisions made during the generation process at some point in the grammar. The fourth group of commands therefore provides ready access to information on the basis of descriptions that have already been presented. Here information chains are followed by mouse clicks rather than giving the name of some linguistic object explicitly. It is possible, in this way, to obtain most information necessary for resource debugging simply by clicking along information chains. This, combined with KPML's extensive options for selecting which information starting points are to be presented, speeds up such resource debugging considerably.

The remaining sections of this chapter describe the individual commands under these groups in detail.

6.2 Graphing systemic networks

Numerous options are provided for browsing the linguistic resources in graphical form. This is probably the best way of navigating the large-scale resources available. In order to ease that navigation, graphing is strongly oriented towards *functional regions*. As described in Section 2.1.1.3, a functional region is a subset of the resources that are concerned with a single 'semantic/functional' area. KPML offers commands for graphing regions in their entirety

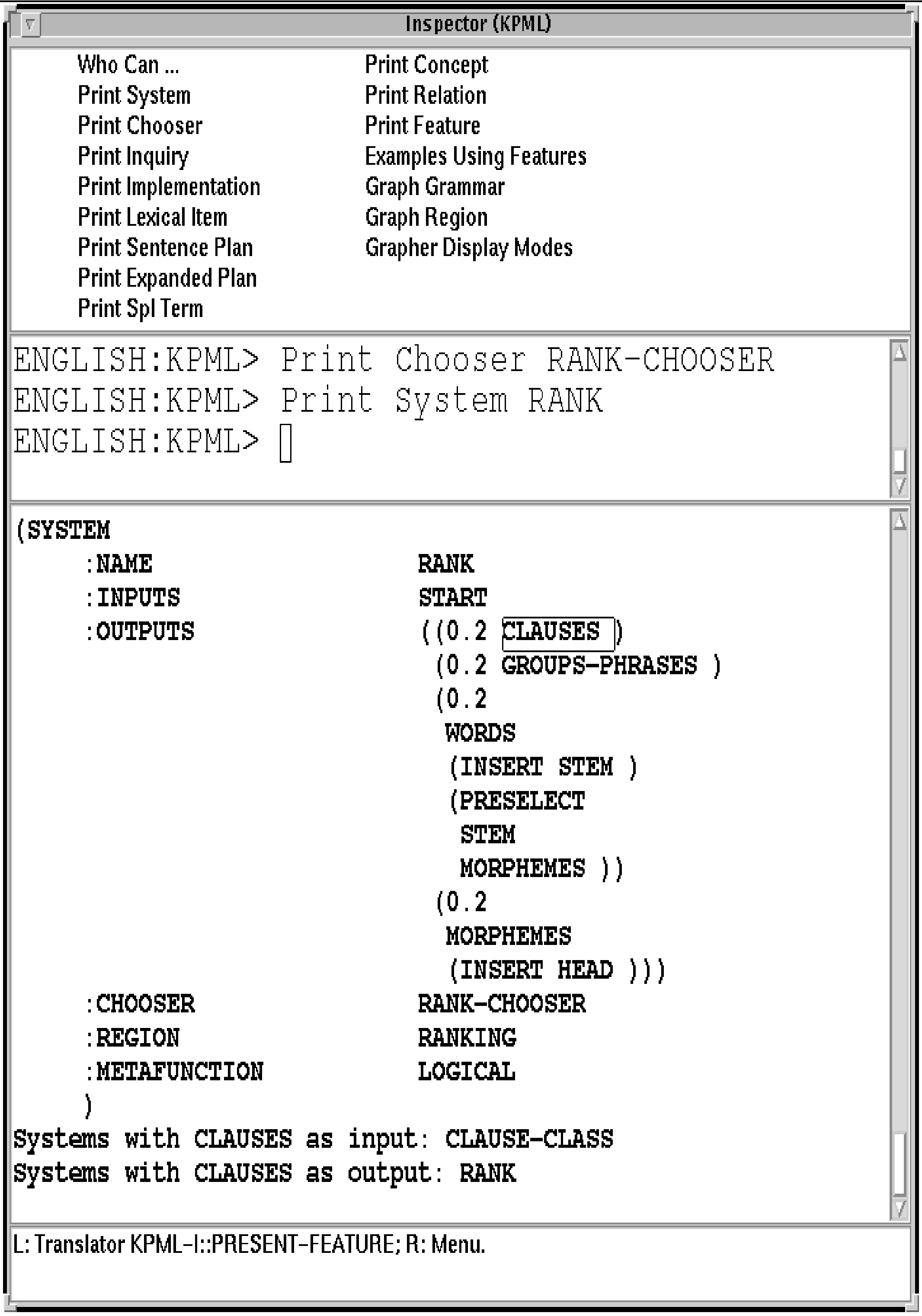


Figure 6.1: The KPML inspector window

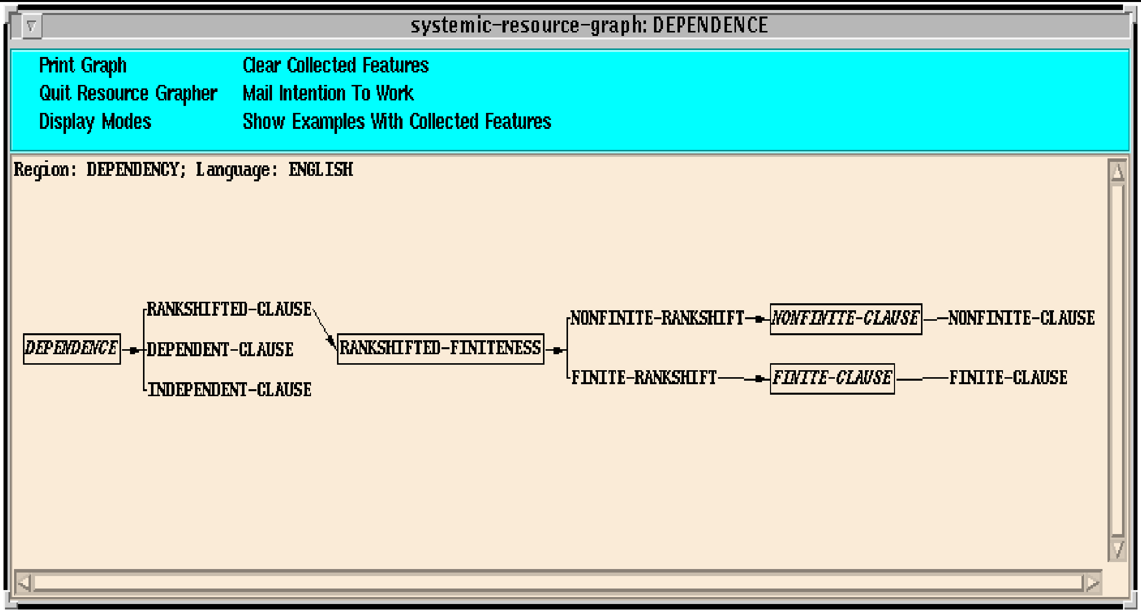


Figure 6.2: Dependency region (extract)

(<Graph Region>) and for graphing network portions starting from any specified grammatical system (<Graph Grammar>). The default graphing behaviour in the latter case is still, however, that only systems from a single grammatical region are selected for graphing. This avoids overly large graphs and maintains some functional coherence in the area of systemic network examined. This default behaviour can, of course, be overriden if desired.

An example of an extract from a region is shown graphed in Figure 6.2. Here we can see part of the region DEPENDENCY, which is partially responsible in the grammars of several languages for determining what kind of linguistic unit is to be generated/described. In the graphed representation, boxed elements denote 'systems' of the systemic network and unboxed elements denote the 'features' of those systems. 'Gates'—i.e., systems with only one output feature–can be recognized in that there is only one dependent feature. These are often terminal and are used for grouping together realization statements; an option explained below allows these to be filtered out of the graph.

Figure 6.2 therefore represents two grammatical systems proper (DEPENDENCE and RANKSHIFTED-FINITENESS) and two gates (NONFINITE-CLAUSE and FINITE-CLAUSE). The first system has three grammatical features ('rankshifted-clause', 'dependent-clause', and 'independent-clause'); the second has two ('nonfinite-rankshift' and 'finite-rankshift'). With respect to the region shown, the system RANKSHIFTED-FINITENESS can only be entered if the feature 'rankshifted-clause' is selected in the system DEPENDENCE.

The default options for graphing combine several different kinds of information over and above that given in Figure 6.2. In the examples given in this section, we start with this simplest case and move towards and beyond the default setting showing the additional information that may be presented.

Systemic network graph windows have their own set of commands and options and remain available until explicitly quit by the user. The immediately following subsections describe the command and mouse-activated options available. These apply to all network graphs.

The command INSPECTOR:<Grapher Display Modes> can also be issued from the graph windows. The options it provides are therefore described below under the GRAPH:<Display Modes> option.

6.2.1 Basic graphing options and commands

This section describes the commands available from the GRAPH window.

6.2.1.1 Quit Resource Grapher

The command GRAPH:<Quit Resource Grapher> exits from, and removes, the associated resource grapher window.

6.2.1.2 Printgraph

The command GRAPH:<Print Graph> places a postscript file of the contents of the graph (as produced under effect of the various display modes: see below) in the default 'hardcopy directory'. The present hardcopy directory can be inspected and changed with the ROOT:<Environment Directories> command (Section 5.4.1) and with the GRAPH:<Display Modes> or INSPECTOR:<Grapher Display Modes> commands described below.

Note: the user is still responsible for sending the created postscript file to an appropriate printer; this is not done automatically.[1]

6.2.1.3 Show examples with collected features

It is possible to 'collect' lists of grammatical features. The simplest way to collect a feature is to click right on a feature shown in a graph and to select the appropriate subcommand as described in Section 6.2.3. The grapher command <Show examples with collected features> then prints in the *Inspector* information pane a list of stored examples where the complete set of features on the collected list occur. This is a quick way of finding examples representing the distinctions drawn in the grammar.

The examples using any single selected feature can also be shown by right-mouse clicking on any grammatical feature shown in the graph and selecting the appropriate command presented.

Note, this will only select from examples where the selection expression is already present in the example record: see Section 10.1 for a description of how and when this occurs.

[1] Unless the flag `kpml-i::*auto-print*` is set.

6.2.1.4 Clear Collected Features

This command clears the list of collected features. Note that it is possible to collect features from several graphs, from the textual displays, and from feature lists produced during generation (e.g., selection expressions) simultaneously; clearing is therefore necessary before collecting when collecting is intended to begin afresh.

It is also possible to clear the collected features by right-clicking on any empty space in the graph and selecting the appropriate command from the menu that appears.

6.2.1.5 Display Modes

Giving the command GRAPH:<Display Modes> (or, almost equivalently, from the *Inspector* main command menu directly with INSPECTOR:<Grapher Display Modes>) allows the user to change the view of the graph in various ways, both in terms of layout and content. The options provided differ slightly depending on whether the user is already graphing an area of the linguistic resources or is setting the grapher display options from the inspector window.

6.2.1.5.1 Content-oriented resource graph options The content-oriented options are as follows.

- ***bounded by region***: this is the normal way of delimiting the amount of network that is shown. Only systems and features that are reachable from the given start system without passing outside the functional region of that start system will be displayed.
- ***maximal depth***: this is another way of delimiting the amount of information shown. Here the restriction is in terms of depth reached. This option can be combined with the previous one in the case of large functional regions.
- ***show previous generation path***: this folds in network traversal information from generation (described in Section 7.7).
- ***terminal gates visible***: this determines whether terminal gates, which make no contribution to the connectivity of the network, will be shown in the graph. If realizations are being sought, then terminal gates will be useful; otherwise, they may merely clutter the network display. Thus, if the apparent end of the network has been reached but, curiously, realization statements appear to be missing, then it is probably due to this flag being turned off.
- ***region external links visible***: this determines the display behaviour at the boundaries of functional regions. Since information about inter-region connectivity can be very useful, and gives a sense of the incompleteness of any information shown, this is the default graphing behaviour. With this flag set all systems lying on the boundary of a functional region also display the *region-external* systems, i.e., systems of other regions, into which they feed. These region-external systems are shown in a smaller, bold typeface with the name of their region of origin attached.

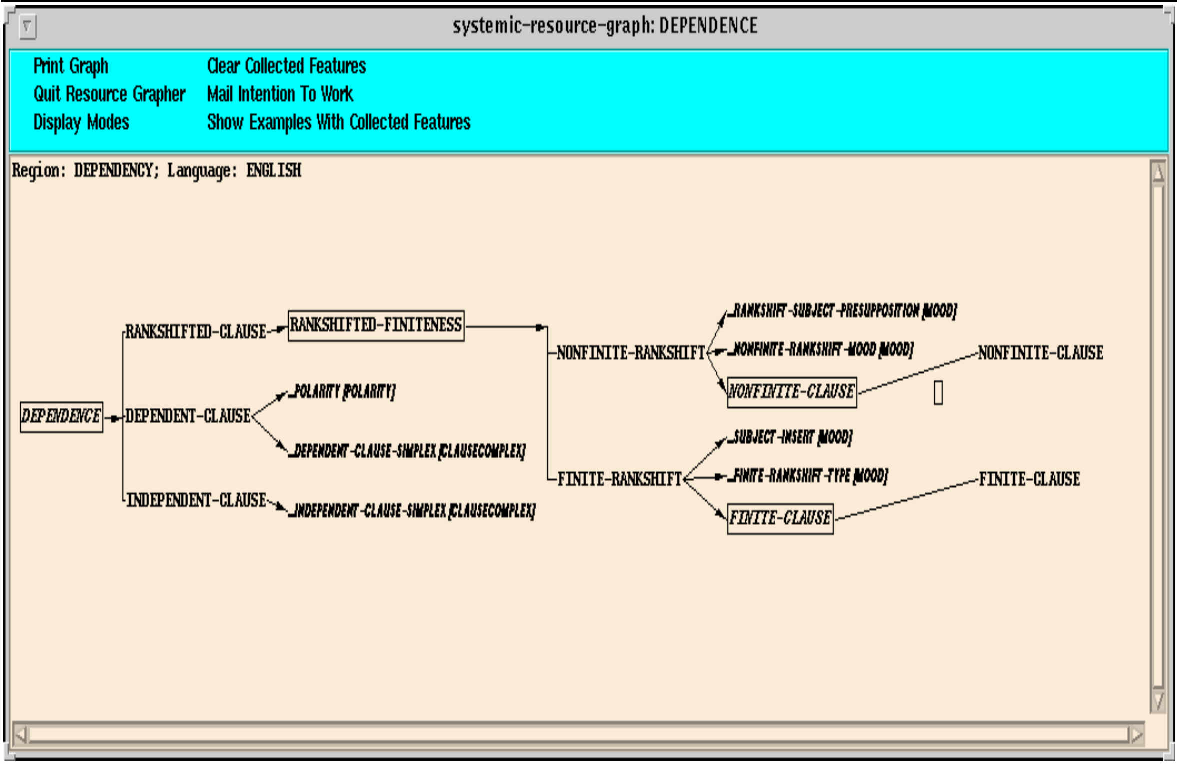


Figure 6.3: Extract from Dependency region with links to other regions shown

The consequence of using this option for the segment of the DEPENDENCY region graphed in Figure 6.2 above is shown in Figure 6.3. Here we can see not only that, for example, feature [independent-clause] lies on the boundary of the DEPENDENCY region (since no further features or systems issue from it), but also that it leads onto the systems INDEPENDENT-CLAUSE-SIMPLEX and INDEPENDENT-PARATACTICS in the functional region CLAUSECOMPLEX—hence we know that it is non-terminal in the network as a whole.

In addition, systems which are part of the region but whose entry conditions include features from *outside* of the region are shown in italics. Systems whose entry conditions are drawn entirely from within the region are shown in a normal typeface. This is also useful for getting a sense of the connectivity of the network, since only those paths from within the region will actually be shown in the graph. Thus, in the example of Figure 6.3, we can see that both FINITE-CLAUSE and NONFINITE-CLAUSE have input conditions additional to those of the region lying outside of the graphed region, whereas RANKSHIFTED-FINITENESS does not.

Clicking on a region-external system brings up a further graph starting from that system and obeying the active graph settings.

Middle-clicking on any system brings up the *entry condition* of that system. This is useful since the graphical notation currently supported within KPML only shows the presence of a feature in an entry condition—it does not show the logical connections between those features.

- ***current language***: this determines the language whose resources are being displayed. Picking a functional region and varying the value of the current language is thus one (very awkward!) way of setting up contrastive views of the multilingual resources; this behavior is provided properly by contrastive graphing (Section 6.2.5).

- ***Systemic notation***: this flag, when set, causes the graphed network to be displayed using standard systemic notation rather than a form more reminiscent of the definition format described in Section 12.2.5. This is particularly intended for use in combination with the show realization statements flag described next.

- ***Show realization statements***: this flag, when set, causes the graphed network to display the realization statements that are associated with particular grammatical features. Realization statements can be shown either in their definitional format (Section 12.2.5) or in standard systemic notation if the previously given flag is also set. Since it is usual for systemic networks to contain their realization statements, this option is the default when KPML is newly configured. Examples of how this appears can be seen in Figures 6.4 and 6.6.

  As well as showing realization statements for all features in a network (the default), it is possible to select to see only those realization statements that were last used when a system was activated. This option goes together with the 'show previous generation path' flag (Section 7.7).[2]

6.2.1.5.2 Layout and hardcopy oriented resource graph options The layout-oriented options are:

- ***vertical scaling***: the distance between elements vertically.

- ***hardcopy vertical scaling***: the distance between elements that will be used in postscript files for hardcopying.

- ***hardcopy directory***: the directory where postscript files for hardcopying will be stored (when the *Print Graph* menu option is used).

- ***header present***: this flag determines whether header information (containing the region name, the current language, and, if hardcopy, the date of production of the graph) is shown in the graph or not.

- ***suitable for figures***: when set, this flag causes hardcopy versions of graphs to be produced in 'single page' mode. Postscript files for inclusion in text documents should normally be produced with this flag set, otherwise the encapsulated postscript produced will not be appropriate (cf. Section 6.2.2).

6.2.1.5.3 Continuation options When called from a grapher window, the grapher modes menu also contains options for specifying whether the current graph is to be replaced by a similar graph respecting the newly set options, whether a new graph is to be produced in addition to the old graph, whether a hardcopy version is to be produced, or whether no action is to follow.

[2]This functionality is based on the description of Kumano, Tokunaga, Inui & Tanaka (1994).

6.2.1.6 Mail Intention to Work

Sends an e-mail message describing the Region that is being graphed expressing the intention to work on that region. This is to provide an improved flow of information between distributed developers of linguistic resources; those who wish to receive such messages should send a note to `bateman@gmd.de`.[3]

6.2.2 Producing graphs for inclusion as figures in documents

The options described in the previous section can be used straightforwardly to produce encapsulated postscript files suitable for inclusion as figures in text documents. The steps are as follows:

1. produce a postscript file by issuing a <Print Graph> or equivalent command with the GRAPH:<Display Modes> flags '*suitable for figures*' and '*header present*' set and unset respectively. (Unless there is some particular reason, the 'show previous generation path' flag should probably also be left unset.)
2. if necessary adjust the bounding box size in the produced file to include only the actual contents of the figure[4]
3. include the generated postscript file in the document in the normal way recommended for encapsulated postscript (e.g., for LaTeX with a 'psfig' style, etc.).

Note that for resource diagrams, better results are often achieved if the hardcopy vertical scaling is also reduced.

An example of a figure produced in this way is shown in Figure 6.4. Note that the figure makes use of several of the further layouting options described below (particularly those of Section 6.2.3.5). The figure shows the first few steps in delicacy of the grammar of English.

6.2.3 Mouse activated resource graph options

In addition to the explicitly shown graphing commands and options described above, the systemic resource graphs also offer an extensive range of operations by mouse clicking on various parts of the displayed graph. These mouse activated options are described here.

6.2.3.1 Showing a full system definition

Left-mouse clicking on a *region-internal* system node in a resource graph (e.g., on DEPENDENCE in Figures 6.2 or 6.3) causes the full textual display of the system's definition to be printed in the *Inspector* window. This is therefore equivalent to issuing the command INSPECTOR:<Print System> and typing in the name of the clicked upon system.

[3] At present, only the most recently activated resource graph determines the region about which a message is sent.

[4] There are several utilities for this: it appears that some older versions of CLIM do not produce an appropriate bounding box size for figures.

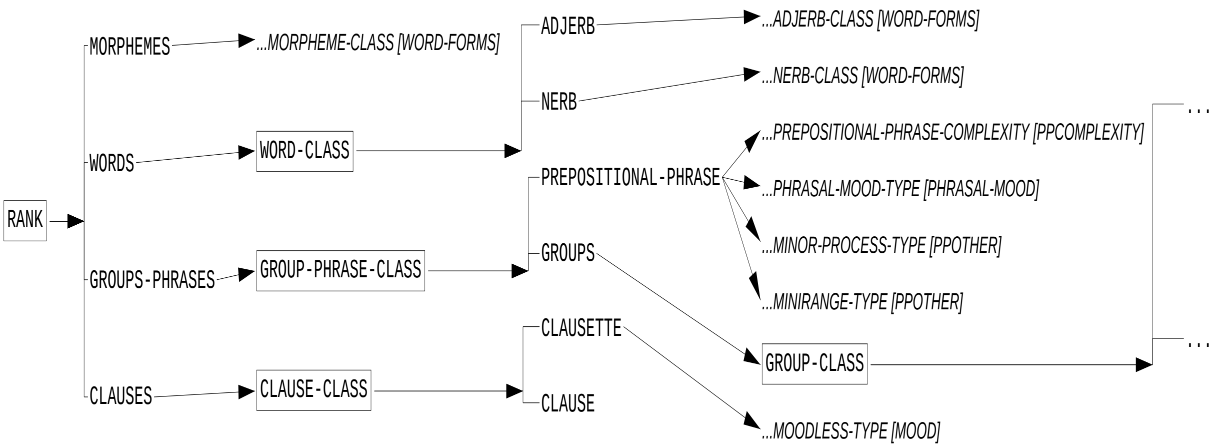


Figure 6.4: Example of EPS figure showing systemic resources

6.2.3.2 Showing the realization statements of a feature

If realization statements (i.e., the structural constraints associated with any grammatical feature: see Section 12.2.5) are not being shown in the graph automatically (by selecting the appropriate flag as described in Section 6.2.1.5 above), then left-clicking on a grammatical feature will pop-up a window containing the realization statements for that feature.

6.2.3.3 Showing the chooser associated with a system

Right-clicking on any node also brings up a menu including *show associated chooser* as an option. This has the same effect as issuing the command INSPECTOR:<Print Chooser> for the chooser associated with the grammatical system of which the clicked upon grammatical feature is an output. In other words, the chooser responsible for choosing the feature clicked upon to be chosen is presented. The mode of chooser display (i.e., textually in the *Inspector* window or in graphical form) is as selected for the <Print Chooser> command.

In addition, if realization statements are being shown in the graph (as caused by setting the appropriate content-oriented flag described in Section 6.2.1.5 above), then left-mouse-clicking on a grammatical feature is a short-cut for the above.

6.2.3.4 Collecting/Discollecting features

It is possible to 'collect' lists of grammatical features. These collected features can then be used in a variety of further operations.

Grammatical features can be either added to, or removed from, the current collection by the corresponding menu options reached by right-clicking on any node in the resource graph. The right-click menu also includes a command for showing the examples which use the selected grammatical feature and for clearing the features collected so far.

Once features have been collected, resource graphs split into two panes, the lower of which

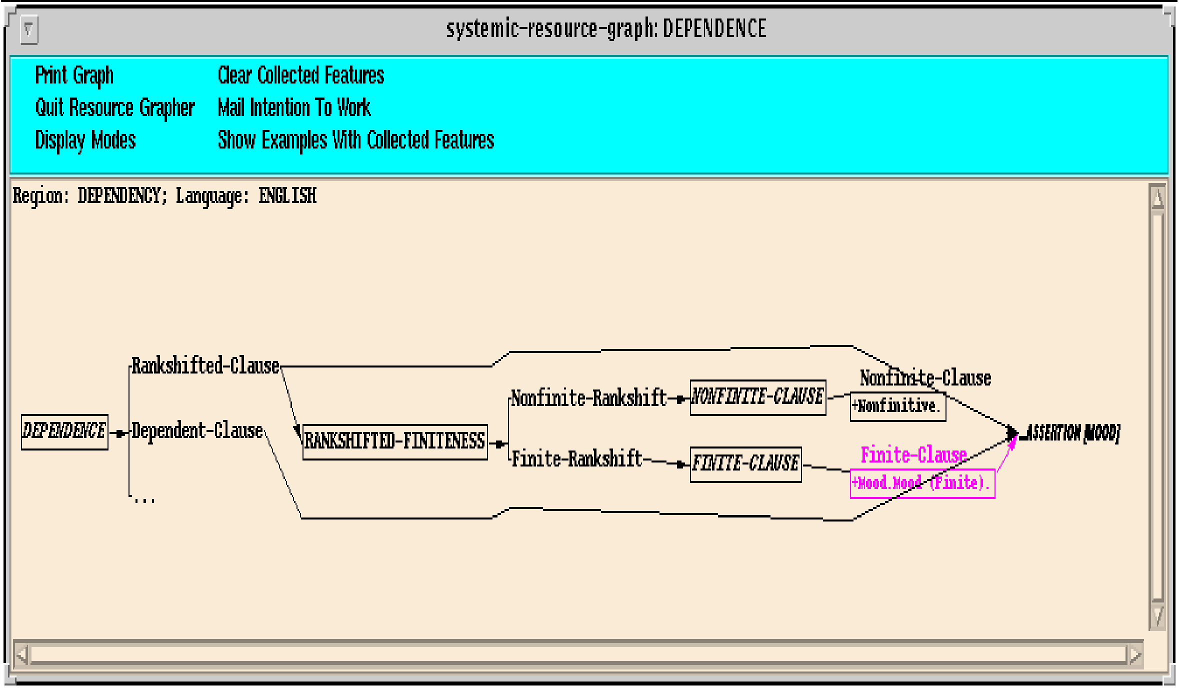


Figure 6.5: Pruned extract from the Dependency region

shows the list of features currently collected. Clicking on any feature on this list will *remove* it from the collected feature list.[5]

6.2.3.5 Pruning the displayed graph

It is possible to select particular portions of the systemic network graph that are not to be shown. The *stop graph here* command under the right-mouse click menu causes subsequent graphing to stop at the clicked upon node. When grammatical *features* are removed from a graph, their absence is marked by '. . . '. When a grammatical *system* is removed, however, there is no indication of this in the graphed network at all. Care should therefore be exercised with this facility in order not to obtain a false view of the resources that are in fact defined. Graph pruning can probably be used to best effect for preparing teaching materials; for grammar maintenance it may be misleading and so should be used with care.

The pruning option can also be used, for example, to remove confusing detail from cluttered graphs. Figure 6.5 shows again the extract from the DEPENDENCY region shown in Figure 6.3 but focusing this time on the connectivity leading to the ASSERTION system in the MOOD region.

[5]This behaviour can be changed by means of the flag `kpml-i::*show-collecteds*` (Section A.3).

6.2.3.6 Redisplaying a graph

In order to make new layout or content options for network graphing take effect, it is necessary to *regraph* the graph. This can be most easily achieved by *left*-clicking on any portion of the graph that is not occupied. This brings up a menu of general options, one of which is redislaying the graph.

6.2.3.7 Spawning further graphs

Left-clicking on a *region-external* grammatical system (for example, the systems INDEPENDENT-CLAUSE-COMPLEX, POLARITY, DEPENDENT-CLAUSE-COMPLEX, etc. in Figure 6.3) brings up a further graph rooted in the clicked upon system and concerning the region of that system. The subgraph is *only* produced when the system selected is shown as being outside of the region with which the graph is currently concerned. This provides a means of growing a graph interactively when it is necessary to follow paths through more than one region.

6.2.4 Graphing regions

The command INSPECTOR:<Graph region> brings up a menu of defined functional regions from which one must be selected, and graphs all the systems that fall within this region. This will be more or less effective depending on the integrity of the region. If a region is poorly defined with many points of contact with other regions, then the graph will look correspondingly complex. 'Holes' in the region—that is, paths through the network that lead out of a region only to lead back into it further downstream—create extra, usually spurious, 'starting' points that are collected together at the left of the graph. Region starting points are defined as those systems for which *all of their* entry conditions lie outside of the region. It is still possible that systems shown within a region have additional entry conditions from outside of the region; this is indicated by printing their names in italics as described above.

Figure 6.6 displays a very small functional region in the normal default style that is active when KPML is freshly configured. Here we can see that realization statements in systemic notation are present (the notation is explained in Table 12.1) and that the region has two 'points of entry'.

The options for changing the appearance and content of a region graph are identical to those described above for resource graphs in general.

6.2.5 Contrastive and multilingual graphing

The three modes of multilingual operations on resources (Section 5.5) apply also to graphing and have the consequences described here.

6.2.5.1 Monolingual graphing

The monolingual graphing option corresponds to the behavior for resource graphing commands described above, i.e., single graphs for the currently selected language are produced.

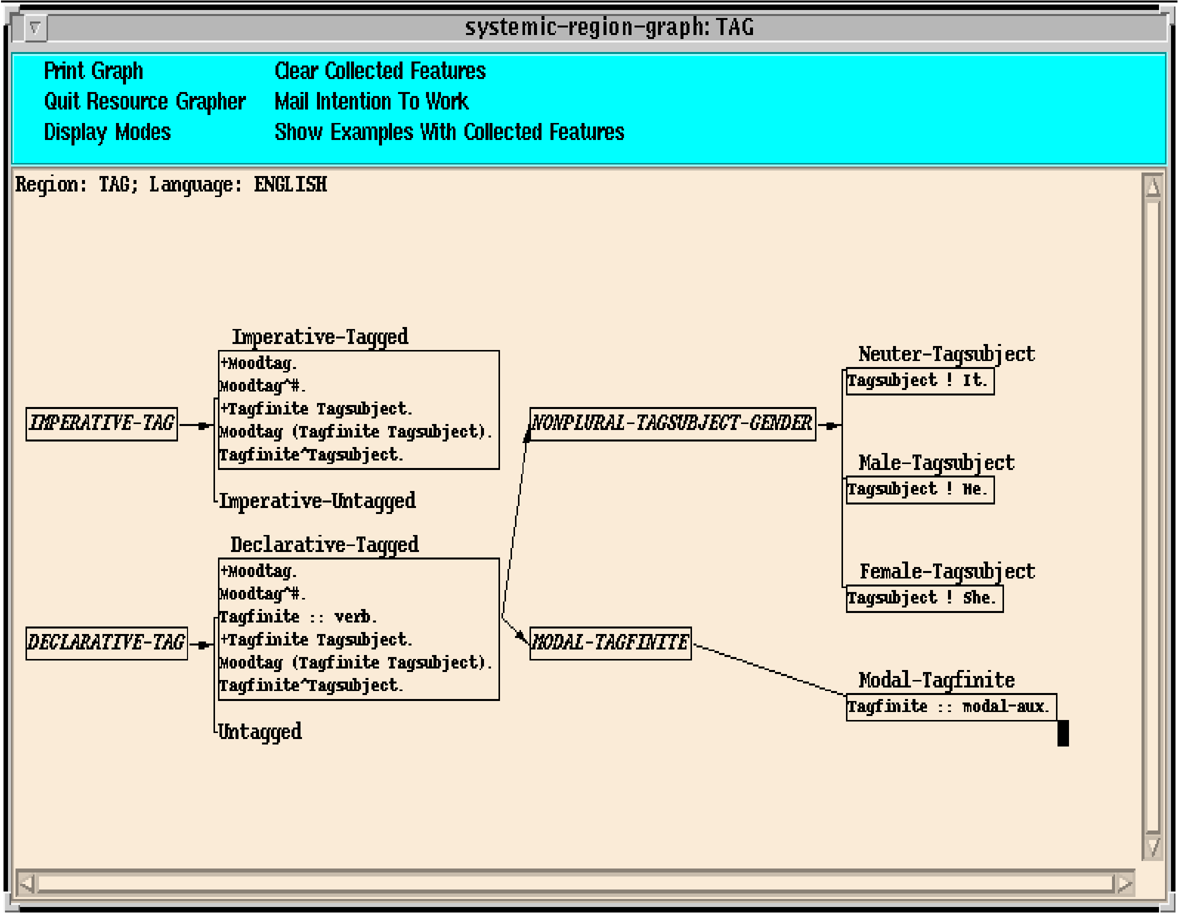


Figure 6.6: Example of region graphing: the region TAG

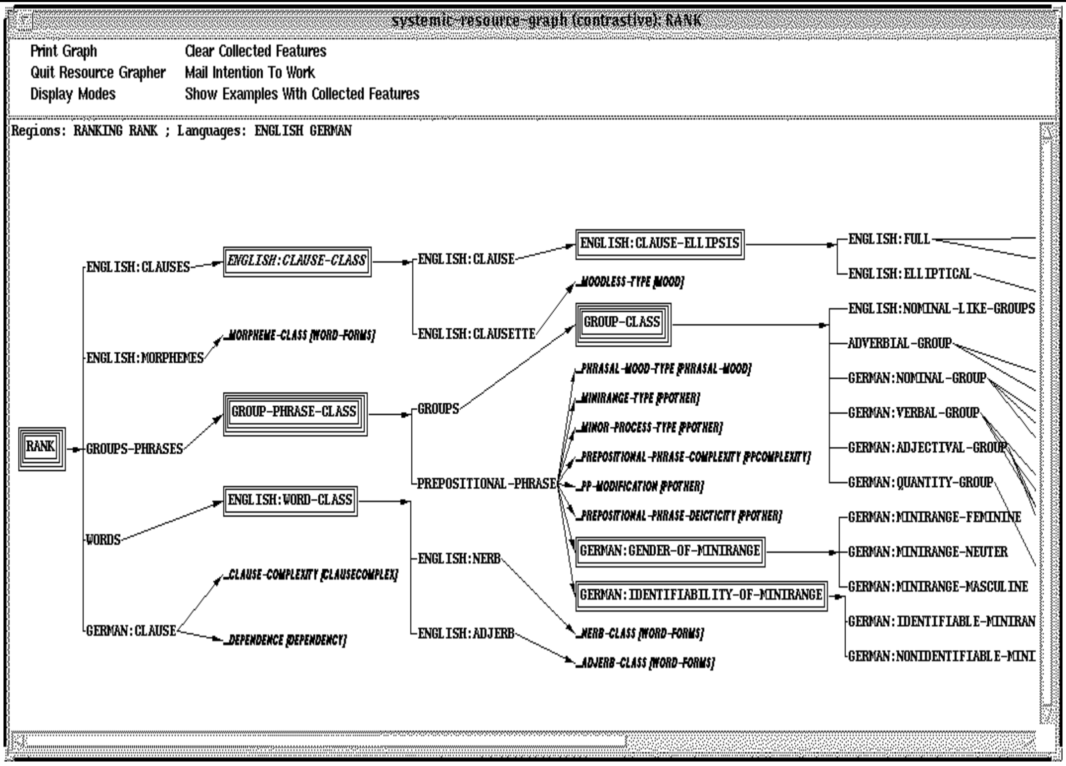


Figure 6.7: Example of multilingual (monochrome) graphing

6.2.5.2 Contrastive graphing

In contrastive graphing mode, the user is additionally prompted for a selection of languages of interest. Then individual graphs are produced in parallel for each language specified allowing the languages to be compared. An example can be seen in Figure 12.7.

6.2.5.3 Multilingual graphing

In multilingual graphing mode, the user is prompted for a selection of languages of interest (currently limited to two). Then a single graph is produced that contains the multilingual view of the resources of those languages. This option can be used to best effect when KPML is configured for color monitors since one of the languages is presented in red, the other in blue, and their common overlap in black. If color is not available, the divergent resources are marked explicitly according to language and congruences are highlighted. An example is shown in Figure 6.7. Here we can see that systems are shown double-boxed if they only hold for a single language, and multiply boxed if they hold for both languages. Features of systems are prefixed by the language they hold for when they do not hold for both languages. Thus in the example we can see that only the systems RANK, GROUP-PHRASE-CLASS, and GROUP-CLASS are common to both English and German among the systems shown.

A further graphed example is given in Figure 12.8.

When graphing networks in this mode, an additional option appears under the Grapher Display Modes (Section 6.2.1.5). This asks whether the 'integrity' of grammatical systems is to be preserved in the multilingual graphs: that is, since in a multilingual set of resources a single feature may belong to more than one system (one for each language), it can be meaningful to graph features with more than one parent. This cannot occur in monolingual graphs, since features are uniquely assigned to systems.[6] When the integrity of systems is to be maintained during graphing, then each feature only has one parent system—*even if this means duplicating features.* The duplicated features will in any case belong to different language varieties. When integrity is not maintained, then the graph may combine similarly named features from different languages. The default display style on newly configuring KPML is that integrity should *not* be preserved. This allows distinct portions of a multilingual grammar to be merged wherever it is possible to do so; preserving integrity guarantees that language graphs diverge as soon as a feature diverges.

The consequences of this graph display choice is shown in the two graphs of Figure 6.8. In the upper graph, system integrity has been maintained; in the lower graph, it has not. The difference in the display modes can be seen by examining the position of the grammatical feature 'clause' in the two graphs. For German, this feature is a output feature of the grammatical system RANK; for English, however, the feature is an output feature of the grammatical system CLAUSE-CLASS, which is itself reached via the 'clauses' feature of the English RANK system. The lower graph, preserving system integrity, therefore shows two 'clause' features, one for each system. In contrast, the upper graph shows only one 'clause' feature, belonging both to the English system CLAUSE-CLASS and the German system RANK. Which view is more appropriate depends on the particular resources and purposes of inspection.

6.3 Inspecting individual object definitions

6.3.1 Introduction

The definitions of any linguistic object can be displayed directly. The commands for displaying objects are found at the top level in the Inspection window command menu. The linguistic objects addressed include: systems, choosers, inquiries, inquiry implementations, grammatical features, lexical items, SPL-terms, sentence plans (both prior to, and following, expansion of macros), and LOOM concepts and relations. Each of these can be typed directly at the Interaction window using the command <:Print ...>; i.e., clicking on 'Print Lexical Item' has the same effect as typing 'Print Lexical Item' in directly at the Interaction window. Similarly, each of the options under the menu option '*Who can ...*' can be obtained also by typing in the equivalent command at the Interaction window.

The full list of these commands, and the description of their function whether reached by typing in or by selecting menu options, is as follows. All commands are described as if the multilingual mode setting were 'monolingual', which is the default (see Sections 5.5 and 6.3.3); examples of different settings are also given in Section 6.3.3.

[6]Definitions can include more than one system possessing a given feature, but all but the last such definition are disabled during loading: cf. Section 7.5.2.4.

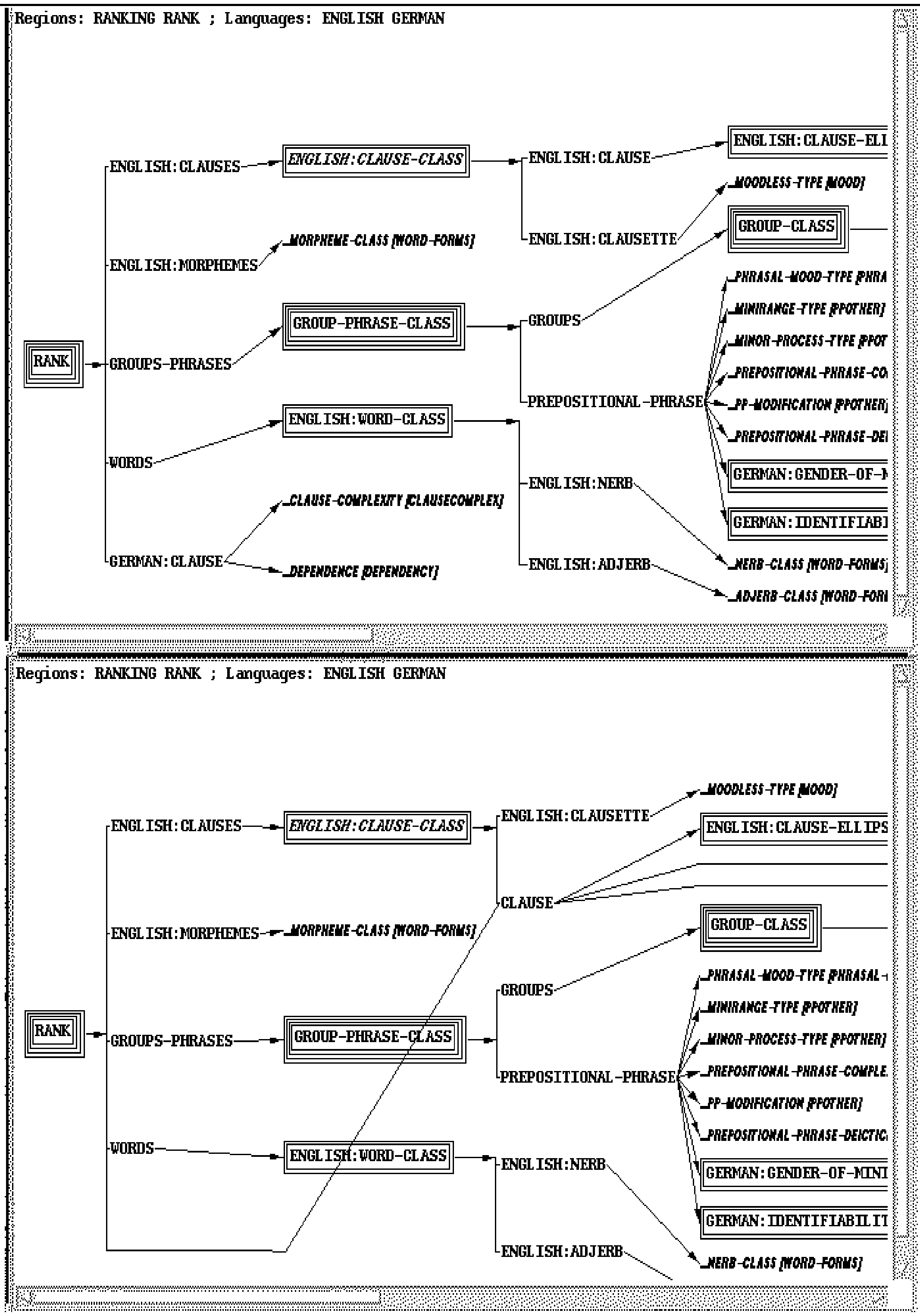


Figure 6.8: Multilingual graphs with and without preservation of grammatical system integrity

6.3.2 Display commands

6.3.2.1 Print System

Requires that a grammatical system name be entered in the interaction window and prints the definition of the system (see Section 12.2.4) in the interaction results window. The definition shown is that which holds for the currently active language. For example, this was the last command performed in the interaction pane shown in Figure 6.1, producing the RANK system definition shown there.

6.3.2.2 Print Chooser

Requires that a chooser name be entered in the interaction window and prints the definition of the chooser (see Section 12.2.6) in the interaction results window. The definition shown is that which holds for the currently active language.

If `:choosers` are on the list of activated pop-up displays (settable from the ROOT:<Flags> command), then the command <Print Chooser> produces a graphical representation of the specified chooser instead of a textual form. Figure 6.9, for example, shows the corresponding graph of the chooser definition given in Section 12.2.6. The layout-oriented grapher options described for system network graphing also apply for chooser graphs and may be set with the command CHOOSER-GRAPH:<Chooser Display Modes>. The linguistic objects present in the graph are, as with the textually presented version, mouse-sensitive, providing links back to grammatical features and inquiries (cf. Section 6.5). In addition, clicking within the chooser graph on any blank space brings up the general commands available for the chooser as a whole.

As with the RESOURCE-GRAPH:<Print Graph> command (Section 6.2.1.2), the CHOOSER-GRAPH:<Hardcopy> command produces a postscript file in the current hardcopy directory. This can then be sent to a printer, or given the appropriate flag settings and modifications (cf. Section 6.2.2) included in text documents. An example of an included encapsulated postscript version of a chooser (that for PRIMARY TENSE in English) is shown in Figure 6.10.[7]

6.3.2.3 Print Inquiry

Requires that an inquiry name be entered in the interaction window and prints the definition of the inquiry (see Section 12.2.7) in the interaction results window. The definition shown is that which holds for the currently active language.

[7]The chooser graphs are the only kinds of graph produced by KPML which are displayed vertically: note that although the display modes options still calls this 'vertical spacing' although in this case the effect is more one of changing the *horizontal* spacing. The EPS example in Figure 6.10 was produced with 'vertical' spacing of 5.

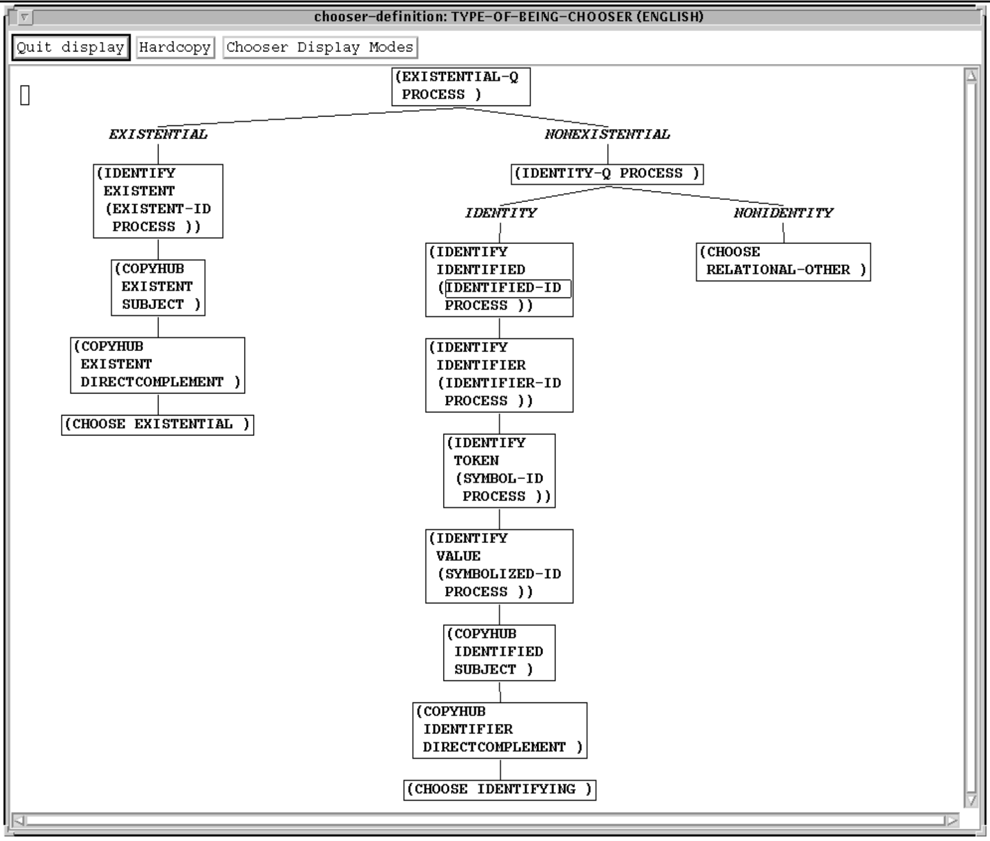


Figure 6.9: Graphical display of a chooser

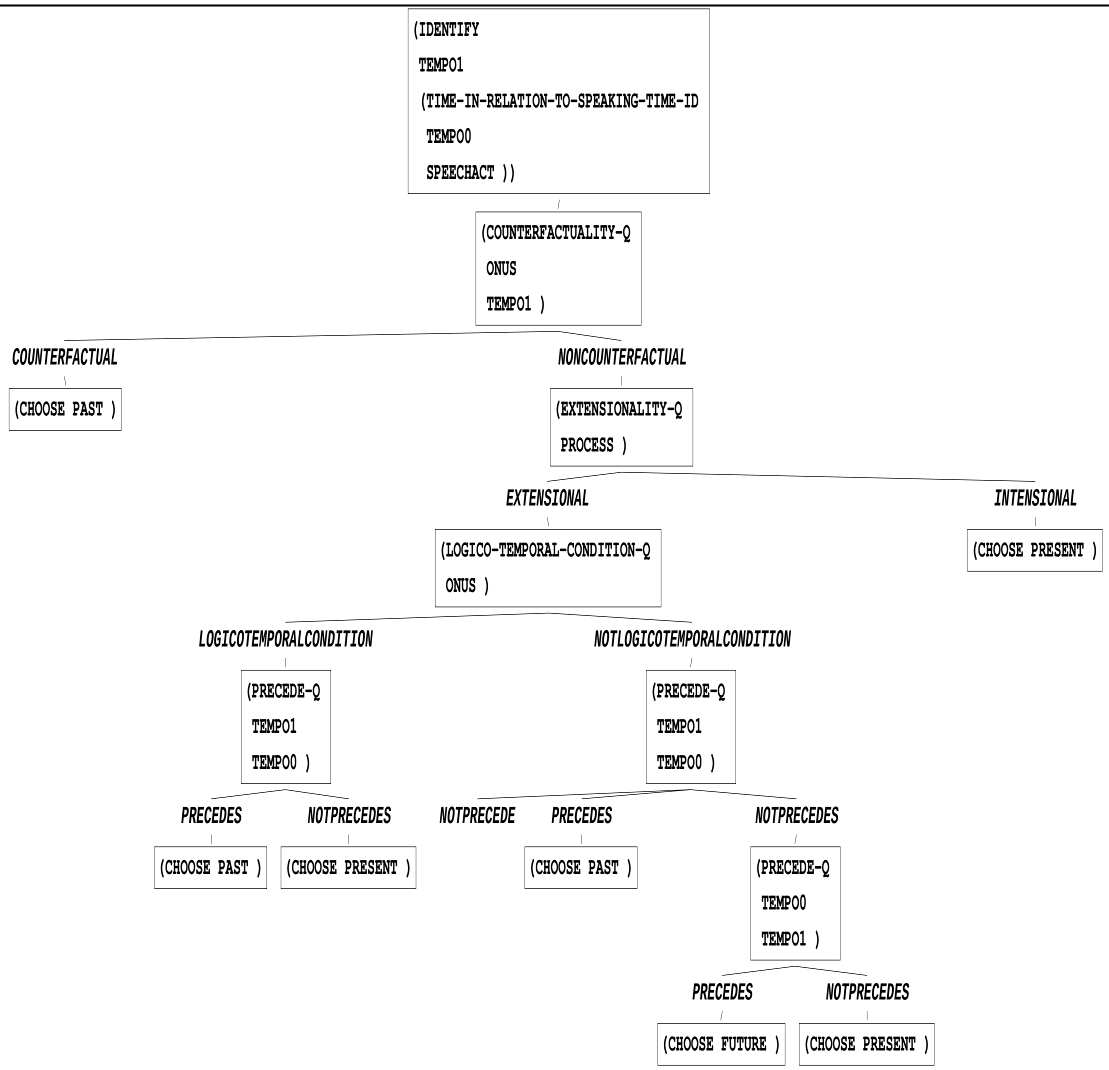


Figure 6.10: Graphical chooser display included in this document as an EPS file

6.3.2.4 Print Inquiry Implementation

Requires that an inquiry name be entered in the interaction window and prints the definition of the *implementation* of that inquiry if it exists and is accessible in uncompiled form. This definition will normally be a Lisp function.

6.3.2.5 Print Lexical Item

Requires that a lexical item name be entered in the interaction window and prints the definition of the lexical item (see Section 12.2.8) in the interaction results window. The definition shown is that which holds for the currently active language.

6.3.2.6 Print Concept

Requires that a LOOM concept name be entered in the interaction window and prints the definition of that concept in the interaction results window.

Note: this is currently specific to Loom, and will need updating when a different knowledge representation language is used. The change is straightforward.

6.3.2.7 Print Relation

Requires that a LOOM relation name be entered in the interaction window and prints the definition of that concept in the interaction results window.

Note: this is currently specific to Loom, and will need updating when a different knowledge representation language is used. The change is straightforward.

6.3.3 Definition displaying and the multilingual modes

All the display commands described above are also further parameterized by the monolingual, contrastive, and multilingual modes. The effects are described as follows. The default, startup mode is monolingual definition printing. The description of the definition form is given in Section 12.2.4.

6.3.3.1 Monolingual definition printing

In monolingual definition printing mode, the definition corresponding to that for the currently selected language is displayed in one of the KPML windows.

```
(SYSTEM
     :NAME                  RANK
     :INPUTS                START
     :OUTPUTS               ((0.333 CLAUSE) (0.333 GROUPS-PHRASES)
                             (0.333 WORDS))
```

```
     :CHOOSER               RANK-CHOOSER
     :REGION                RANK
     :METAFUNCTION          LOGICAL
     )
```

6.3.3.2 Contrastive definition printing

In contrastive definition printing mode, displays are given of the individual definitions corresponding to views from either all of the known language varieties or, alternatively, if a set of languages has been 'focused' (Section 5.6.2), from this focused set. For example, if German and English have been focused (or, if KPML has only been configured to expect German and English), then <Print System *Rank*> produces the following.

```
Language: GERMAN
(SYSTEM
     :NAME                  RANK
     :INPUTS                START
     :OUTPUTS               ((0.333 CLAUSE) (0.333 GROUPS-PHRASES)
                             (0.333 WORDS))
     :CHOOSER               RANK-CHOOSER
     :REGION                RANK
     :METAFUNCTION          LOGICAL
     )
Language: ENGLISH
(SYSTEM
     :NAME                  RANK
     :INPUTS                START
     :OUTPUTS               ((0.2 CLAUSES) (0.2 GROUPS-PHRASES)
                             (0.2 WORDS (INSERT STEM)
                              (PRESELECT STEM MORPHEMES))
                             (0.2 MORPHEMES (INSERT HEAD)))
     :CHOOSER               RANK-CHOOSER
     :REGION                RANKING
     :METAFUNCTION          LOGICAL
     )
```

6.3.3.3 Multilingual definition printing

In multilingual definition printing mode, a combined view of all of known or focused languages is presented. For example, in the same situation as the the previous example, multilingual mode printing would produce the following.

```
(SYSTEM
     :NAME                  (:GERMAN :ENGLISH RANK)
     :INPUTS                (:GERMAN :ENGLISH START)
     :OUTPUTS               ((:ENGLISH 0.2 :GERMAN 0.333 :ENGLISH CLAUSES
                              :GERMAN CLAUSE)
                             (:ENGLISH 0.2 :GERMAN 0.333 :GERMAN :ENGLISH
                              GROUPS-PHRASES)
```

```
                          (:ENGLISH 0.2 :GERMAN 0.333 :GERMAN :ENGLISH
                           WORDS :ENGLISH (INSERT STEM) :ENGLISH
                           (PRESELECT STEM MORPHEMES))
                          :ENGLISH (0.2 MORPHEMES (INSERT HEAD)))
:CHOOSER                 (:GERMAN :ENGLISH RANK-CHOOSER)
:REGION                  (:ENGLISH RANKING :GERMAN RANK)
:METAFUNCTION            (:GERMAN :ENGLISH LOGICAL)
)
```

6.4 Object selection according to specified criteria

The KPML inspector window includes commands for accessing linguistic objects on the basis of specific resource-centered properties. The full list of commands is as follows.

6.4.1 'Who has' selections

6.4.1.1 Who has as input

The command <:Who has as input> requires that a grammatical feature be entered in the interaction window and prints a list of those grammatical systems that have the feature as input in the interaction results window. The connectivity shown is that relevant for the currently active language.

6.4.1.2 Who has as output

The command <:Who has as output> requires that a grammatical feature be entered in the interaction window and prints a list of those grammatical systems that have the feature as output in the interaction results window. The connectivity shown is that relevant for the currently active language.

6.4.2 'Who can' selections

The INSPECTOR:<Who Can...> command brings up a wide range of 'who-can' type queries described as follows. All of them can also be given directly as type-in commands at the *Inspector* interaction pane.

6.4.2.1 Who can lexify

Requires that either a grammatical function or a lexical item name be entered in the interaction window and prints in the interaction results window a list of those grammatical systems that lexify the grammatical function to have the lexical item as realization. The unspecified argument (i.e., either the grammatical function or the lexical item) is filled by any such unit in the loaded resources where the specified kind of realization statement holds. This option is reached from the menu option <Who can ...>, followed by the selection <... lexify>. The systems selected are those relevant for the currently active language.

6.4.2.2 Who can inflectify

Requires that either a grammatical function or a morphological lexical feature name be entered in the interaction window and prints in the interaction results window a list of those grammatical systems where the grammatical function is inflectified to have the morphological feature. The unspecified argument (i.e., either the grammatical function or the morphological feature) is filled by any such unit in the loaded resources where the specified kind of realization statement holds. This option is reached from the menu option <Who can ...>, followed by the selection <... lexify>. The systems selected are those relevant for the currently active language.

6.4.2.3 Who can classify

Requires that either a grammatical function or a lexical feature name be entered in the interaction window and prints in the interaction results window a list of those grammatical systems where the grammatical function is classified to have the lexical feature. The unspecified argument (i.e., either the grammatical function or the lexical feature) is filled by any such unit in the loaded resources where the specified kind of realization statement holds. This option is reached from the menu option <Who can ...>, followed by the selection <... classify>. The systems selected are those relevant for the currently active language.

6.4.2.4 Who can insert

Requires that a grammatical function be given in the interaction window and prints in the interaction results window a list of those systems where the specified function is inserted. This option is reached from the menu option <Who can ...>, followed by the selection <... insert>. The systems selected are those relevant for the currently active language.

6.4.2.5 Who can order

Requires that a grammatical function be given in the interaction window and prints in the interaction results window a list of those systems where the specified function is ordered with respect to some other function. This option is reached from the menu option <Who can ...>, followed by the selection <... order>. The systems selected are those relevant for the currently active language.

6.4.2.6 Who can partition

Requires that a grammatical function be given in the interaction window and prints in the interaction results window a list of those systems where the specified function is partitioned with respect to some other function. This option is reached from the menu option <Who can

...>, followed by the selection <... partition>. The systems selected are those relevant for the currently active language.

6.4.2.7 Who can preselect

Requires that either a grammatical function or a grammatical feature name be entered in the interaction window and prints in the interaction results window a list of those grammatical systems where the grammatical function is preselected to have the grammatical feature. The unspecified argument (i.e., either the grammatical function or the grammatical feature) is filled by any such unit in the loaded resources where the specified kind of realization statement holds. This option is reached from the menu option <Who can ...>, followed by the selection <... preselect>. The systems selected are those relevant for the currently active language.

6.4.2.8 Who can ask

Requires that a branching inquiry be given in the interaction window and prints a list of those choosers where the inquiry is asked in the interaction results window. This option is reached from the menu option <Who can ...>, followed by the selection <... ask>. The choosers thus shown are those relevant for the currently active language.

6.4.2.9 Who can identify

Requires that a grammatical function be given in the interaction window and prints in the interaction results window the identifying inquiries that can provide a value for this function. This option is reached from the menu option <Who can ...>, followed by the selection <... identify>. The inquiries selected are those relevant for the currently active language.

6.4.2.10 Who can pose identifying inquiry

Requires that an identifying inquiry be given in the interaction window and prints a list of those choosers where the inquiry is asked in the interaction results window. This option is reached from the menu option <Who can ...>, followed by the selection <... pose identifying inquiry>. The choosers thus shown are those relevant for the currently active language.

6.4.3 Examples Using Features

The command INSPECTOR:<Examples Using Features> is similar to the command that is available from the resource graph window described above. However, if there are no collected features, it prompts the user for a feature that is to be sought in the stored example records. A list of examples that have the feature specified somewhere in one of their selection expressions is given in the *Inspector* window

As always, this will only select from examples where the selection expression is already present in the example record: see Section 10.1 for a description of how and when this occurs.

Features are normally collected directly from graph of the systemic networks (Section 6.2.3.4) or from selection expressions produced during generation (Section 7.4).

6.5 Direct inspection and information chains

6.5.1 Introduction

In addition to the kinds of commands given above where explicit names of linguistic units must be entered, it is also possible (and more usual) to use direct mouse driven commands that operate on particular linguistic objects visible in the KPML display windows—such as the network graphs or the inspection window. The mouse-sensitivity of the nodes in systemic graphs was described above (Section 6.2.3); this section discusses the possibilities that textual representations offer for supporting the direct following of information chains also.

As in the descriptions above, it should be noted that there are also options for supporting various tracing operations during generation. These are not described here, but will be returned to in Chapter 7.

Normally all objects that are displayed in any KPML window are to a greater or lesser extent mouse sensitive. The kinds of operations that can be performed on these objects depends on their type. Thus, different types of linguistic objects support different kinds of operations. The descriptions given here will be organized according to linguistic object type.

Linguistic objects are generally referred to by name. In the textual displays, it is therefore the *names* of various linguistic objects that occur that are mouse sensitive.

As an example, Figure 6.11 shows the textual display of a grammatical system with all the mouse sensitive objects present artificially highlighted. Each highlighted object here also shows the *type* of the object that is mouse sensitive. Moving the mouse around a window quickly reveals the mouse sensitive objects in that window; the type of an object can usually be seen in the mouse documentation line.

6.5.2 Inspection operations on grammatical systems

6.5.2.1 Printing system definition

Clicking left on a system name will print a definition of that system in the *Inspection* information pane. This is also the first option on the menu produced by clicking right on a system name. In both cases this is fully equivalent to the command <Print System>, but saves typing in the desired system name.

6.5.2.2 Print associated chooser

Right-clicking on a system name brings up a menu including *show associated chooser* as an option. This has the same effect as issuing the command INSPECTOR:<Print Chooser> and typing in the clicked upon name.

(SYSTEM
    :NAME $\boxed{\texttt{RANK}}_{system}$
    :INPUTS $\boxed{\texttt{START}}_{feature}$
    :OUTPUTS ((0.2 $\boxed{\texttt{CLAUSES}}_{feature}$)
        (0.2 $\boxed{\texttt{GROUPS-PHRASES}}_{feature}$)
        (0.2 $\boxed{\texttt{WORDS}}_{feature}$ (INSERT STEM)
            (PRESELECT STEM $\boxed{\texttt{MORPHEMES}}_{feature}$))
        (0.2 $\boxed{\texttt{MORPHEMES}}_{feature}$ (INSERT HEAD)))
    :CHOOSER $\boxed{\texttt{RANK-CHOOSER}}_{chooser}$
    :REGION RANKING
    :METAFUNCTION LOGICAL
)

Figure 6.11: Mouse sensitive objects within a textual display

6.5.2.3 Graph Grammar starting from system

Right-clicking on a system name brings up a menu including *Graph Grammar* as an option. This has the same effect as issuing the command INSPECTOR:<Graph Grammar> and typing in the clicked upon name.

6.5.3 Inspection operations on grammatical features

The first command described in this section is that reached by left-clicking on a grammatical feature. The remainder are all reached by selecting the corresponding command from the menu produced by right-clicking on a grammatical feature.

Note that all the commands here can also be typed in full in the INSPECTOR *interactor* pane; in this case, the arguments required must be typed directly as with all such commands.

All information given is for the current language *only*.

6.5.3.1 Displaying usage of grammatical features

Clicking left on a grammatical feature presents lists of all those systems that use that feature in their inputs and their outputs. This has the same effect as issuing the command INSPECTOR:<Print Feature> and typing in the clicked upon name. The same command can be reached under the right-click menu.

6.5.3.2 Who has as input

Selecting this option from the right-click menu for grammatical features prints a mouse-sensitive list of the systems which have the clicked upon feature in their entry conditions.

6.5.3.3 Who has as output

Selecting this option from the right-click menu for grammatical features prints a mouse-sensitive list of the systems which have the clicked upon feature as one of their output features. Note that a well-formed systemic network can only have one such system: however, it is quite possible to have multiple definitions that share an output condition known to KPML at the same time. All but one of these will, however, be *disabled*—i.e., will not be used in generation (cf. Section 7.5.2.4).

6.5.3.4 Show path to

Selecting this option from the right-click menu for grammatical features prints a mouse-sensitive list of the features which lie on the traversal path leading to the clicked upon system. For example, selecting this option for the feature 'finite-rankshift' (cf. Figure 6.3) produces the following for English:

```
Path to feature: finite-rankshift is
(FINITE-RANKSHIFT FULL CLAUSES CLAUSE RANKSHIFTED-CLAUSES)
```

The paths given are calculated starting from the selected feature and working leftwards in the network along all connections that do not participate in disjunctions. Giving a full path description including disjunctions is naturally somewhat more expensive and so is avoided.

6.5.3.5 Show chooser of feature

Selecting this option from the right-click menu for grammatical features is equivalent to issuing a <Print Chooser> command for the systems who have the clicked upon feature as an output. That is, all choosers that could be responsible for the clicked upon feature are displayed.

6.5.3.6 Graph from feature

Selecting this option from the right-click menu for grammatical features brings up a systemic network graph with the clicked upon feature as root. All layout and content options are as described in Section 6.2.

6.5.3.7 Collect feature

As described in Section 6.2.1.3, features can be 'collected' for various purposes. Selecting this option allows the collection of any feature that is displayed in the *interaction result* pane.

6.5.3.8 Uncollect feature

As described in Section 6.2.1.3, features can be 'collected' for various purposes. Selecting this option removes the clicked upon feature from the current list. This allows any feature that is displayed in the *interaction result* pane to be removed from the current collection.

6.5.3.9 Clear collected features

Selecting this option clears *all* the collected features (cf. Section 6.2.1.3) regardless of which feature happened to be clicked upon.

6.5.4 Inspection operations on choosers

All of the following commands can also be typed direction in the INSPECTOR window's *interactor* pane. In this case, the arguments required must be typed directly as with all such commands, instead of being the object clicked upon.

6.5.4.1 Print chooser

Clicking left on a chooser presents presents the definition of that chooser shown according to current defaults (i.e., a textual view in the INSPECTOR window or as a separate graphical view as determined by the appropriate flag under the ROOT:<Flags> command. This has the same effect as issuing the command INSPECTOR:<Print Chooser> and typing in the clicked upon name. The same command can be reached under the right-click menu.

6.5.4.2 Show inquiries of chooser

Selecting this option from the right-click menu for a chooser displays in the INSPECTOR's *interaction results* pane a mouse-sensitive list of all the inquiries that are used in the clicked upon chooser.

6.5.4.3 Systems of chooser

Selecting this option from the right-click menu for a chooser displays in the INSPECTOR's *interaction results* pane a mouse-sensitive list of all the systems that use a particular chooser. Normally there is only one such system, since each system has its own chooser. But if there are multiple versions of systems then it is possible that a single chooser would be used by these distinct versions. The main rationale of this command is to provide a mouse-driven means of following an information chain from choosers back to systems.

6.5.5 Inspection operations on inquiries

All of the following commands can also be typed direction in the INSPECTOR window's *interactor* pane. In this case, the arguments required must be typed directly as with all such commands, instead of being the object clicked upon.

6.5.5.1 Print inquiry

Clicking left on an inquiry presents presents the textual definition of that inquiry in the INSPECTOR's *interaction results* pane. This has the same effect as issuing the command INSPECTOR:<Print Inquiry> and typing in the clicked upon name. The same command can be reached under the right-click menu.

6.5.5.2 Print implementation

Selecting this option from the right-click menu for an inquiry displays in the INSPECTOR's *interaction results* pane the inquiry implementation associated with an inquiry: this is normally a Lisp function. The implementation can normally only be printed if the corresponding function has not been compiled.

6.5.5.3 Who can ask

Selecting this option from the right-click menu for an inquiry displays in the INSPECTOR's *interaction results* pane a mouse-sensitive list of those choosers that ask the inquiry as a branching inquiry. As well as being typed directly as INSPECTOR:<:Who can ask>, this command can also be reached as INSPECTOR:<Who can ... : ... ask>.

6.5.5.4 Who can pose identifying inquiry

Selecting this option from the right-click menu for an inquiry displays in the INSPECTOR's *interaction results* pane a mouse-sensitive list of those choosers that pose the inquiry as an identifying inquiry. As well as being typed directly as INSPECTOR:<:Who can pose identifying inquiry>, this command can also be reached as INSPECTOR:<Who can ... : ... pose identifying inquiry>.

6.5.6 Inspection operations on lexical items

The only inspection operation for a lexical item is to print its definition. This may include mouseable morphological features.

6.5.7 Inspection operations on SPL terms

The only inspection operation for an SPL term is to print its definition.

6.5.8 Inspection operations on examples

A wide range of further inspection operations are supported on the basis of 'example records'. These are stored as test suites, or *example sets*. The role and maintenance of examples is

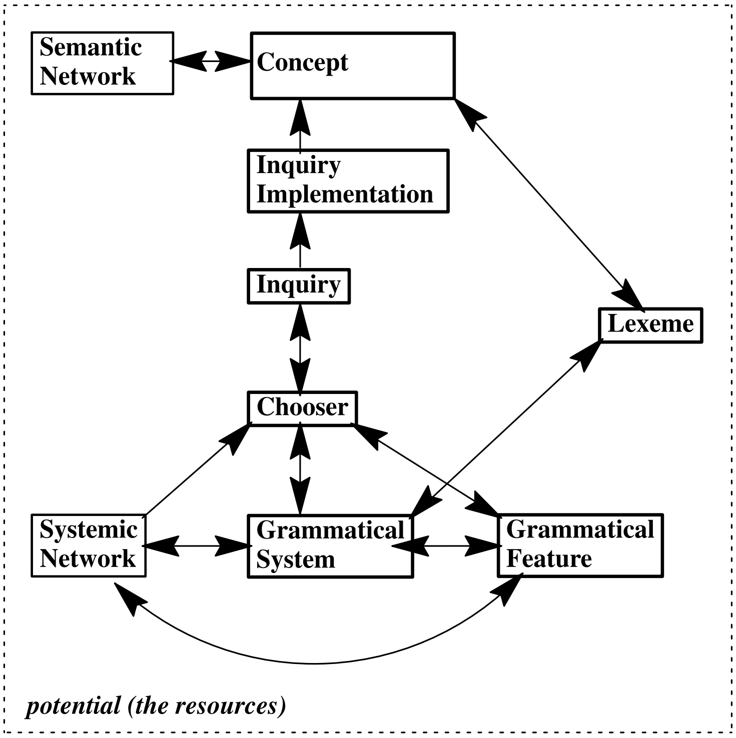


Figure 6.12: Summary of information chain possibilities: resources

described in detail in Chapter 10. The description of inspection (and other) operations on examples is therefore given there.

6.6 Overview of information inspection chains

The possibilities for following information chains for particular types of linguistic objects are summarized in Figure 6.12. Each box in the diagram represents a particular type of linguistic object supported by KPML. An arrow between a pair of boxes indicates that it is possible, by means of a mouse-click combination, to go from an object of the indicated source type to the associated object of the target type.

For example, there are menu commands available which make it possible to go from the textual or graphical representation of a 'grammatical feature' to:

(i) the system for which that feature is an output,

(ii) the chooser which is responsible for selecting that feature,

(iii) a graph of the systemic network having that feature as root.

Right mouse-clicking on an object will generally bring up a menu presenting the indicated options (plus options for generation tracing as described in Chapter 7).

Note that the options presented here are considerably extended by the *example sets*; this is summarized in Section 10.4.

Chapter 7

The KPML Development Window

7.1 Introduction

The KPML development window is used for maintaining and developing linguistic resources by means of generation—either single instances of generation or by running through example sets. The development window includes commands for controlling the amount of information shown or gathered during generation and for inspecting the results of generation.

7.2 Window Layout

An example of the *development* window is shown in Figure 7.1. It is divided into 4 panes stacked vertically:

- the menu commands for resource development,
- the 'target' sentence pane which shows the intended generation target when working with examples,
- the interaction results display pane,
- the mouse documentation line.

7.3 Overview of commands

The development commands can be divided into the following categories:

- commands for starting generation (<Generate Sentence>, <Generate Again>, and <Example Operations: Example Runner>),
- commands for controlling the information displayed or collected during generation (<Generation Modes>, <Reset Generation Modes>, and <Clear Tracing Options>),

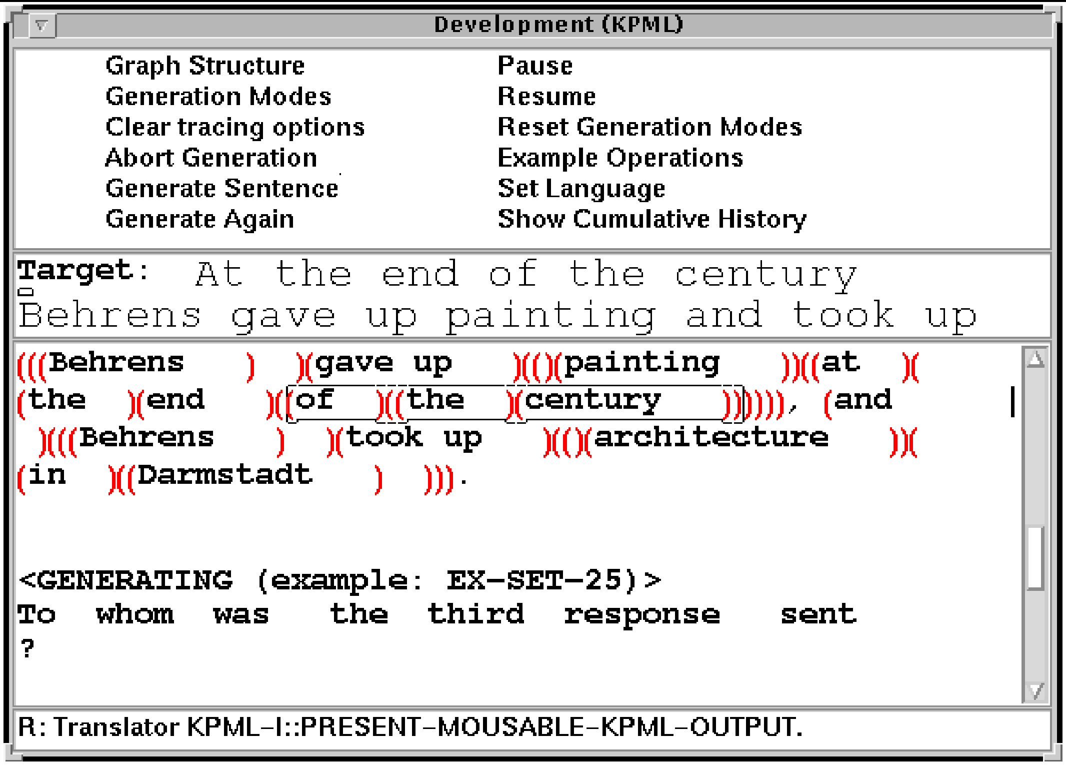


Figure 7.1: KPML development environment window

- commands for interrupting or resuming the generation process (<Pause>, <Abort Generation>, and <Resume>,
- commands for showing the results of generation (<Graph Structure> and <Show Cumulative History>),
- commands for operating on examples and example sets (those under <Example Operations>).

In addition, the single command <Set Language> can be used to explicitly indicate the language for which generation is to proceed and for which display and graphical information is to be given.

These command groups, with the exception of those for example sets which are described separately in Chapter 10, are described in detail in the following sections. Since the main function of the development window is concerned with *generation*, our discussion of the commands take generation as its starting point.

7.4 Generation: basics

7.4.1 Introduction to generation with KPML

Generation with KPML has two main functions. First, generation is motivated externally in that some application, or user, wishes to automatically create natural language strings expressing some given semantic content. Second, generation is motivated internally in that a set of linguistic resources can be demonstrated to be adequate for successfully generating some pre-specified set of test sentences: a test suite, or *example set*. In both cases it may be necessary to inspect both the linguistic resources defined and the generation process in order to ascertain why a particular semantic specification did not produce an acceptable string, or to discover what kind of semantic specification would have been appropriate. Because generation plays a crucial role as a mode of resource 'debugging' or maintenance, an extensive range of commands are provided for finding out exactly what happened during generation and why.

The simplest mode of generation in KPML is that some semantic specification is provided as input (expressed in the SPL, sentence plan language, notation) and a string is produced that expresses that semantic specification in the natural language determined by the resources loaded and the language desired.

Although it is possible to give one-off semantic inputs (cf. Section 14.1), it is more common for work to proceed on some set of selected examples—these might be a test suite for the resources, or some set of representative sentences that a particular application needs to have generated. The task is generally to ensure that the defined resources do the expected thing with the inputs given and, when they do not, to repair or extend them. For this reason, generation with KPML is *example-driven.* Semantic inputs are loaded as a set of examples and then selected for generation—either singly or as a collection. The creation and maintenance of example sets is described in detail in Chapter 10.

7.4.2 Starting generation

The command DEVELOPMENT:<Generate Sentence> brings up a menu of examples, selection of one of which initiates generation by appeal to the semantic specification stored in that example (rather than by user interaction or by rote from an example file). Generation is normally undertaken in 'implemented' mode (Section 7.4.7), which means that inquiry implementations, where they exist, are used to interrogate the environment (knowledge base, upper model, etc.) rather than having a user intervene in the generation process.

The menu showing available examples can be set to show either:

- all examples,
- the examples relevant for a particular language or set of languages.

In addition, the example selection menu can show examples identifying them either by the *target* string (i.e., a pre-specified desired result for reference during debugging), or by the *generated* string (i.e., the string that was actually produced last time the example was generated). These are controlled by the appropriate flags under the ROOT:<Flags> command.

Restriction to current language(s) works prior to any further restriction of example displays. The default settings are for language restricted display of the example targets.

The command <Generate Again> restarts generation for the previous example generated.[1] If no previous example was generated, then this command is equivalent to <Generate Sentence>.

KPML offers a variety of ways of inspecting the results of generation.

The simplest is to display the string produced (or strings, if the input was not specific enough) directly in the *interaction results* pane. This is what has happened in Figure 7.1 above. Here two options are provided, also shown in the figure: either the string can be printed as is (the second string shown), or it can be printed with an explicit marking of constituency structure (the first string shown). Explicitly marking the constituency has the advantage that it is easier to see the underlying structure in order to directly select constituents by mouse for further information gathering. For this reason, this display is the default when KPML is newly configured; this can be changed under the ROOT:<Flags> command.[2]

An alternative presentation form is to produce a summary of the structure generated in the *Interaction Results* pane. The display of this structure can be activated by setting the appropriate flag by means of the <Flags> command. An example of this structure is shown in Figure 7.2. The generated string is also displayed along with a summary of any warnings that may have occured during generation.[3]

Both the generated string and the display structure are mouse sensitive and allow for several further resource maintenance and debugging commands as described in Section 10.3.

7.4.3 Generation and the multilingual modes

An appropriate use of the multilingual modes can in many cases remove the need for explicit language switching. The interaction of these modes with generation is described in this section. It should be noted, that *whenever a language switch is carried out automatically as a consequence of the multilingual mode settings, the full side-effects of explicit language switching as set out in Section 7.11 will occur.*

7.4.3.1 Monolingual generation

When the monolingual generation mode is selected, sentence generation (as long as it is started by <Generate Sentence> or <Example Operations: Example Runner>) proceeds for the currently active language only. This is the behavior closest to that of Penman (with the current language obligatorily set to `:english`).

[1]Regardless of how input. Thus an SPL input specification could be given as an argument to the function `say` and subsequently regenerated with <Generate Again>.

[2]The displayed versions of the generated strings are in fact produced with the example record operation <Display Generated String> (Section 10.2.5.1). The mouse-sensitive structure can therefore be fine-tuned to differing granularities—it need *not* be a direct representation of the syntactic structure. KPML uses the same structure for this presentation as the 'rich mouseable structure' that can be passed back to applications for further processing (e.g., adding hyperlinks, defining their own mouse sensitivity, etc.). Section 14.5 describes these facilities in detail.

[3]All warnings can be suppressed by setting the flag `*demo-mode*` to true: **not recommended for everyday use!**

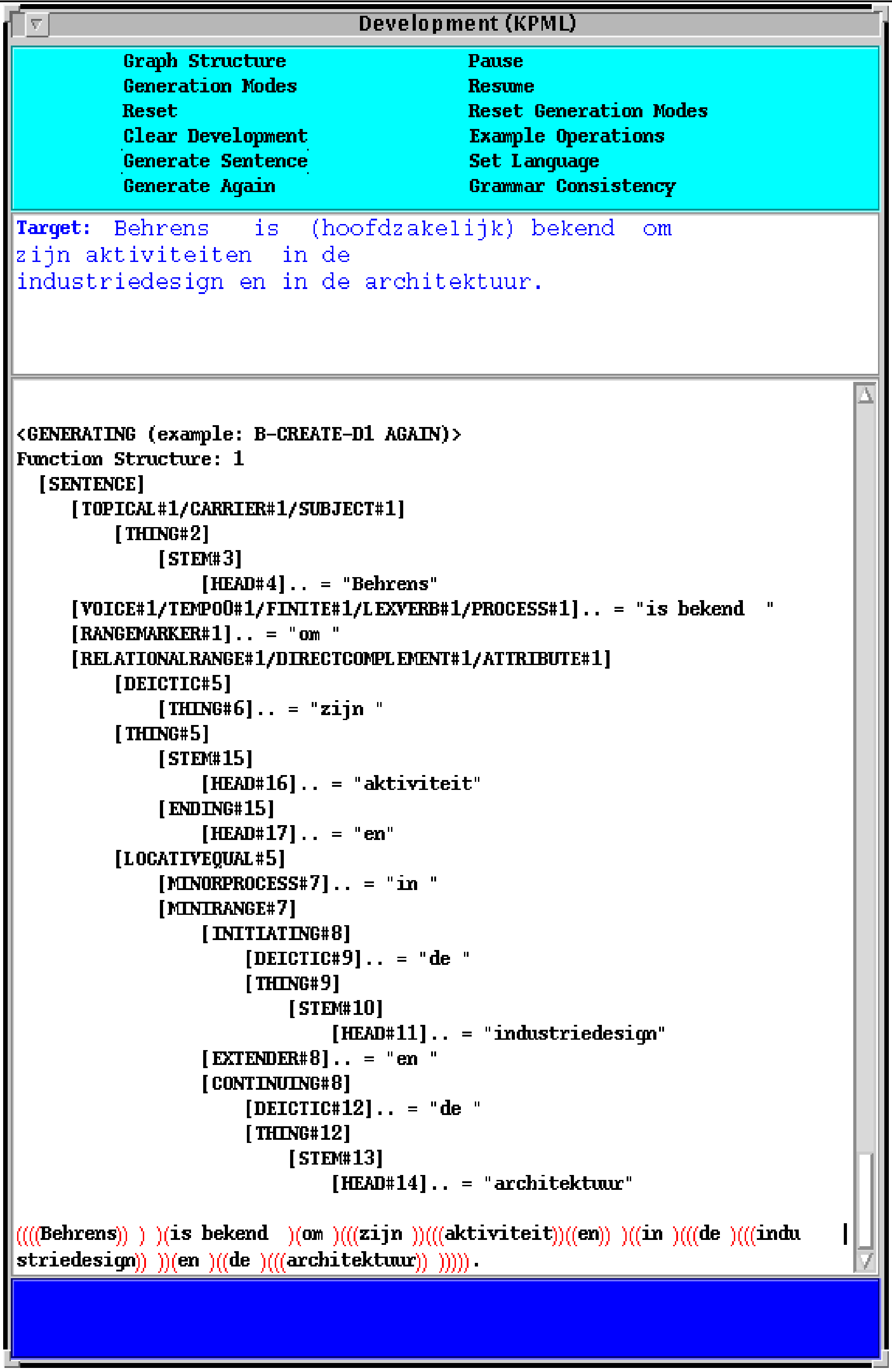


Figure 7.2: Example structural result of generation

7.4.3.2 Contrastive generation

When the contrastive generation mode is selected, sentence generation (as long as it is started by <Generate Sentence> or <Example Operations: Example Runner>) proceeds for all the languages for which KPML is currently configured (or the subset of those languages declared to be in focus via language focusing: Section 5.6.2). A given SPL specification is attempted for each language variety for which it is declared relevant.

7.4.4 Semantic defaults and macros

Various semantic defaults may need to be initialized before using provided linguistic resources for generation. Moreover, individual language resources may define their own SPL macros. Failing to correctly set up the required default environments can be a cause of completely mystifying generation failures. It is therefore worthwhile being aware of the default mechanisms, although KPML attempts to make the loading and activating of defaults largely transparent to the user.

In any realistically sized resources there are a large number of inquiries defined for each language (typically around 600) and many of these control aspects of generation variation that are not directly derivable from a barebones 'propositional content'—i.e., they do not belong to the ideational metafunction. To simplify the use of such resources for generation, applications may define sets of defaults that are to be used for providing the responses to selected inquiries. A set of defaults, called a *default environment*, can be defined and then activated and deactivated at will. The default environment mechanism was implemented for Penman by Bob Kasper in 1988-89 and is inherited and made partially multilingual in KPML code. For more information about the possibilities for default definitions, therefore, see the SPL descriptions in the Penman documentation (Penman Project 1989). Their definition forms are described in Section 12.2.2.2.

Definitions of SPL macros and default environment definitions are automatically loaded as part of the default <load linguistic resources> command. The files concerning these definitions are as follows (see Section 12.2.14):

- a file `basic-spl-macros.lisp`: which holds the SPL macro definitions,
- a file `basic-spl-defaults.lisp`: which holds the SPL default environment definitions.

When a set of inquiries are being used for generation, each inquiry may have an *active default*. The initial value for the active default comes most straightforwardly from the `trivialdefault` slot of the inquiry definition (see Section 12.2.7). Subsequent activation of default environments may, however, alter the active default. Subsequent deactivations of default environments restore the previously present active default.

When a set of linguistic resources, particularly of inquiries, is freshly loaded, the active defaults are undefined. If generation is attempted in this state, it will usually fail since insufficient information is present in the input semantic specifications. (Of course, if the semantic specification is complete, then no defaults will be required.) Setting the desired defaults is then a two step process:

1. the base case defaults present as `trivialdefaults` are made current (i.e., are copied into the `activedefault` slots);

2. further defaults are activated by issuing 'begin default environment' commands. A set of default environments that are to be standardly assumed for a resource set is typically held in a file: `properties.lisp`.

If KPML can establish that defaults have not been activated, then the above two steps will be triggered automatically when generation is attempted. A message to this effect will be given to the user. This will only affect the defaults of the current language.

7.4.5 Run-time cautions

Cautions may be produced during run time. They indicate that, although there is not necessarily anything wrong yet, a possible problem has been recognized. e.g. if the realization operation `(conflate Subject Agent)` is specified before one of the operators has been inserted, a caution to that effect will be given. If, by the end of the pass through the grammar, this function has still not been inserted, then a warning will be given.

7.4.6 Run-time warnings

Warnings can often be produced during generation, especially if the resources are not fully debugged. Some of the most common warnings (and the actions that are taken) are:

- lexical item not known – an appropriate lexical item with the required grammatical features is created,

- association changed – the new association replaces the old (this is a warning since it indicates that a non-monotonic operation is involved that would have failed in a purely declarative rendering of the resources),

The most serious warning is the following:

- no hub specified for grammatical function.

This indicates that a pointer to semantic information necessary for continuing generation has not been made available by means of an appropriate identification in an identifying inquiry (cf. Section 12.2.7). Without such information the generation process cannot continue and so the user is asked whether an association is to be provided by hand. Normally, however, resources should not get themselves into this situation and so the problem should be dealt with in the resource definitions rather than being worked around. Failing to give an association can easily bring the generation process to an ungraceful halt, even landing in the calling Lisp process. The option for continuing the process of the *development* window should then be taken.

7.4.7 Running modes

The inquiries defined by linguistic resources usually include both a 'formal' and an 'informal' form. As described in Section 12.2.7, the informal form is a predefined natural language question which represents the semantic characterization presented by the inquiry (for ask operators) or a description of the type of semantic entity demanded (for identify operators).

The informal form can be used to guide computational implementations of the inquiries, since they provide a high-level gloss of what each inquiry is intended to do, or they can be treated as questions that a user can answer directly during generation. This is the basis of the mode of using a grammar where the user simulates the 'environment', that is, everything that lies outside a given set of resources, manually; this mode is called *deimplemented* mode.[4] Deimplemented mode is useful for gaining familiarity with the linguistic resources themselves, without needing to attend to any knowledge distinctions or text planning issues. Also available under deimplemented mode is the possibility of having the inquiry responses taken from a pre-stored *example record* (see Chapter 10 and Section 12.2.9). This facility, combined with sets of pre-run example records (such as the *exercise set* for the Nigel grammar of English, originally compiled by Lynn Poulton for the Penman system), provides a very useful resource in its own right for coming to understand what the grammar can do and how it does it, as well as assisting in resource maintenance and development. KPML provides a significantly extended set of operations on such example records compared to those of the Penman system. These are described separately in Chapter 10.

The formal version of an inquiry generally consists of just the inquiry's name and certain additional information concerning the inquiry's function; the complete definition specification is illustrated in Section 12.2.7. Relevant here, however, is the notion of *implementing* an inquiry. Code for answering inquiry questions automatically is called the *implementation* of the inquiry. Each inquiry that is implemented includes the name of a Lisp function that is the code that actually runs when the grammar needs to determine which selection of grammatical feature is appropriate. Accordingly, the mode of using the linguistic resources where the inquiries operate automatically without user intervention is termed *implemented* mode. This is the normal mode of use that is started under the <Generate Sentence> main command menu option.

7.4.8 Boundary conditions

There are a number of circumstances that can arise which cause the normal flow of generation to be interrupted. This is, of course, far more likely to happen while linguistic resources are being developed and debugged. Most foreseeable boundary conditions are caught by the KPML window interface and are handled by presenting menu options to the user. It is, however, possible that unforeseen kinds of errors might throw the user back into the Lisp debugger. This will occur in the *calling Lisp process*, not the KPML window interface. For this reason, it is best to maintain access to the calling Lisp process, so that an appropriate restart can be initiated from the debugger in the normal way.

[4] This mode played a more important role in the early days of the Penman system before the inquiry interface and semantic representations had become stable. It is still potentially useful for getting to understand in detail how the architecture works and the kind of modularities that it achieves. A detailed example of a mock-up deimplemented generation traversal is given by Mann & Matthiessen (1985).

Whenever a choice boundary condition is reached, whether genuinely or by virtue of a forcing flag (see Section 7.5.2 below), the following set of options appears:

1. Force a choice and continue: shows the user the available grammatical features that the system offers and asks the user to select one. This becomes the feature chosen in that system and any chooser information is ignored.

2. Make no choice and continue: the generation process continues with no choice being made in that grammatical system. This removes an expected grammatical feature from the selection expression and so downstream systems depending on any of the features of that system will not be entered. The final result of the generation will therefore be incomplete to a degree dependent upon the number of systems whose entry was forbidden.

3. Run chooser again: re-executes the chooser that is associated with the system at that time. This could lead to a different result if

   - the example record has been edited, so changing the responses that the inquiries receive (in deimplemented mode),
   - the chooser itself has been edited,
   - the inquiries or inquiry implementations have been edited, or
   - the environment has altered (in implemented mode).

4. Run chooser again in manual mode: re-executes the chooser that is associated with the system at that time but insists that the user supply the necessary responses to the inquiries that are asked.

5. Associate a new chooser with this system: asks the user to supply a new chooser that replaces the existing one; not recommended for normal grammar use.

6. Enter debugger: simply enters the normal Lisp debugger; not recommended for normal grammar use, but a fairly harmless way of suspending generation for a period while information on the state of generation is inspected.

7.5 Tracing and debugging during generation

7.5.1 Introduction to generation debugging under KPML

Whereas the sole mode of generation debugging supported by the Penman system can be described as one of 'tracing', KPML supports an additional mode best described as 'result focusing'.

'Tracing' refers to displaying more or less detail of the generation process as it occurs and attempting to intervene when things go wrong. This mode of interaction could sometimes be suitable for getting to know how the system operates. In this mode, the user can, in the extreme case, single step through the generation operations observing each decision made.

'Result focusing' refers to selectively indicating information that could be relevant for the debugging process, allowing generation to proceed, and then focusing in on the selected 'results'. This is a very much faster way of debugging resources since it provides (i) pinpoint inspection of the aspects of the generation process requested rather than less fine 'tracing', (ii) systematic overviews of some selected aspect of the resources during generation, and (iii) the ability to check up on selected decisions as and when they show themselves in need of checking, rather than during the generation process. The success of the method in general relies, of course, on how effectively one can determine the appropriate places to inspect: here also, therefore, KPML provides a significant set of tools.

It is also possible within KPML to set the information to be gathered during generation sufficiently broadly that the result approximates that of tracing. The only difference is that the 'result focusing' is not available interactively: it is a *collection* of the information used during generation and not, as with tracing, a step-by-step report on what the generator is doing.

The two modes can also be mixed, in that generation can be allowed to proceed to particular selected points and then interrupted so that tracing can be used.

This section describes these options in detail. We start with the tracing options inherited from the Penman system and then list the process result options specific to KPML. Both kinds of options are reached by the command <Generation Modes>. This command brings up a further menu to the user whereby various levels of information detail can be set during generation.

The command <Reset Generation Modes> then resets the options displayed in the <Generation Modes> menu so that no tracing or display during generation is activated.

An example of the menu in the state following initial loading of the KPML system is shown in Figure 7.3; the initial value of flags is `nil`, meaning 'off'.[5] As indicated above, most of these modes are for tracing and are directly inherited from the Penman system. The more focused debugging and development options that KPML provides are placed under the heading 'Result focusing'. An additional group of commands (some new, some old) includes those options that effect the generation process itself in some way rather than the information that is to be presented. Each group is described in detail in the following subsections.

Note that it may be necessary to scroll this menu to find all the options presented.

7.5.2 Generation tracing modes

In this section we describe those tracing modes that allow information to be given concerning various aspects of the generation process *during* generation. Some of these might particularly be useful for the novice gaining familiarity with the workings of the system during generation, or for very difficult to diagnose problems that arise with user-defined resources.

[5] The internal symbol names for these flags are listed in Appendix A below; this enables them to be used to control the amount of information that is given during generation when the window interface is not being used.

`Automatically create new examples ···`	`T`	**NIL**
RESULT FOCUSING OPTIONS...		
	none	
`Cumulate System and Inquiry Activity ···`	`traced`	
	`all`	
`Update environment record ···`	`T`	**NIL**
GENERATION TRACING OPTIONS...		
`Realize Selectively ···`	`T`	**NIL**
`Show Constituent Starts ···`	`T`	**NIL**
`Realize until constituent number ···`	**NIL**	
`-----`		
`Show System and Inquiry Activity ···`	`T`	**NIL**
`-----`		
`Show Why System is Firing ···`	`T`	**NIL**
`Show Disabled Candidate Systems ···`	`T`	**NIL**
`Show System Entry Dependencies ···`	`T`	**NIL**
`Show Preselections ···`	`T`	**NIL**
`-----`		
`Show Immediate Realizations ···`	`T`	**NIL**
`Show Lexical Features ···`	`T`	**NIL**
`Show Lexical Selection ···`	`T`	**NIL**
`Show Ordering Constraints ···`	`T`	**NIL**
`Show Ordering Events ···`	`T`	**NIL**
`Show Ordering Results ···`	`T`	**NIL**
`-----`		
`Show Associations ···`	`T`	**NIL**
`Show Inquiry Answer Source ···`	`T`	**NIL**
`Show Entailed Inquiry Response ···`	`T`	**NIL**
`-----`		
`Single Step ···`	`T`	**NIL**
`Enter Debugger On Warnings ···`	`T`	**NIL**

Figure 7.3: Generation tracing and result focusing modes

7.5.2.1 Show Constituent Starts

Each pass through the grammar is responsible for the realization of some particular part of the structure that is being built. Parts of structure are defined in terms of bundles of grammatical functions that previous cycles through the grammar have composed and assigned preselections to. When realizing such a cycle therefore, it can be relevant to know both the members of the function bundle that that cycle is to be concerned with and the preselections that have been established for it.

When this flag is set, at the beginning of each pass through the grammar the message:

```
Realizing bundle: ((FUNCTION1 FUNCTION2 ... FUNCTIONn)
                   ((Preselect FUNCTIONi FEATUREi)
                    (Preselect FUNCTIONj ...     )
                    ...
                  ))
```

appears. The first sub-list contains the functions that collectively form the function bundle that the pass will be concerned with. These will be functions that the pass through the grammar responsible for their higher level of structure had conflated. The second sub-list contains all the preselection realization statements that that higher level pass performed with respect to any of the members of the bundle. The pass about to be begun will therefore be committed to selecting all of the grammatical features mentioned in the preselection list and all the features that these entail.

The special case of the first pass through the grammar produces a similar message, citing the pseudo function bundle *TOP*, any preselections that may have been set manually, and the knowledge representation hub that is to have a linguistic result generated for it.

7.5.2.2 Show System And Inquiry Activity

Traces the system that is entered, the chooser entered, the inquiry questions asked, their results and the choice that the system makes, in a convenient form. This is all that is necessary to see the traversal of the grammar network in progress and the chooser decisions that were made to control that traversal. Under the new-style interface a *Generation History* window is brought up that shows the successive decisions made during generation; under the old-style interface, several panes are already present for showing this information.

The successive tracing of the generation process is useful for getting to know a set of resources and how it functions in generation. Even for simple sentences, however, this option presents a great deal of information. For serious resource development the more focused tracing aids provided by KPML should be used.

An example of the *generation history* window is given in Figure 7.4. It consists of two main panes, set across the middle of the window: the *system history* and the *inquiry history*. The system history pane shows each system and the feature selected in that system; thus the first line of this pane informs us that in the system CAUSE-ADJUNCT, the feature 'noncause' was selected. The inquiry history pane shows the asked inquiry and the response that it received; the first complete entry here informs us that the inquiry `certainty-q` was asked and received the response `notcertain`. Above these are displayed on the left the current system that has

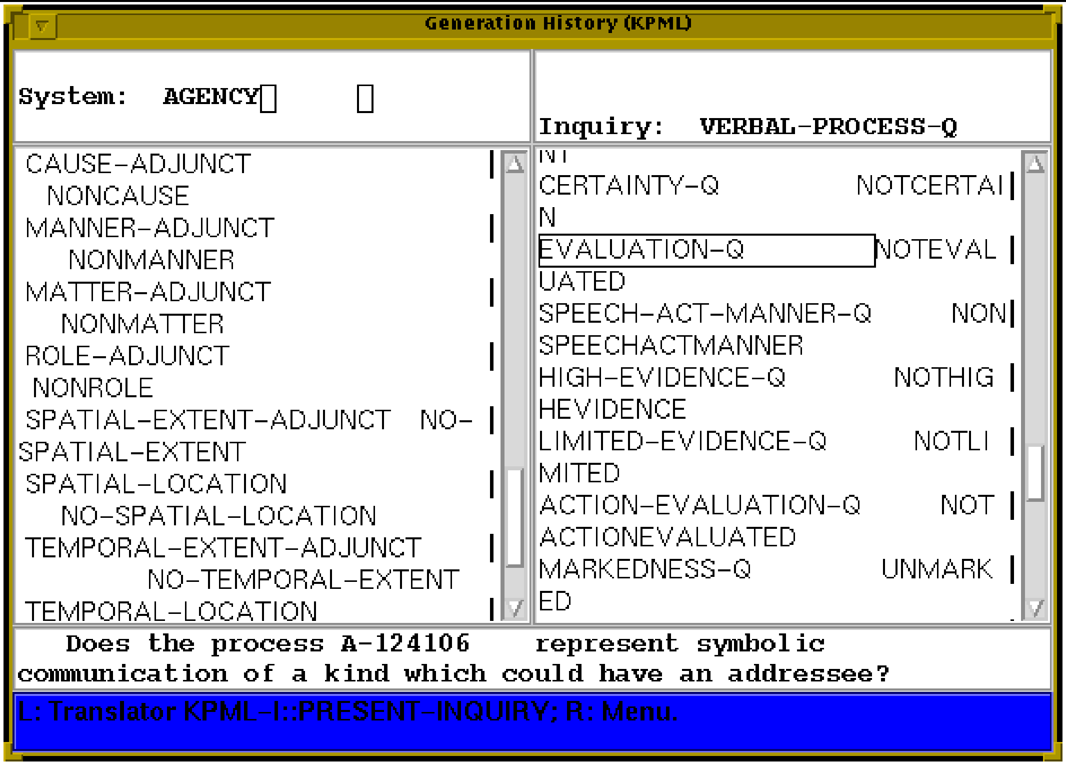


Figure 7.4: Generation History Window

been reached in network traversal (here: AGENCY), and on the right the current inquiry that is being asked in chooser traversal (here: `verbal-process-q`). Below the central panes is a single long pane displaying the natural language gloss of the current inquiry. All of the linguistic objects mentioned are mouse-sensitive.

Since there are very many inquiries and systems that are entered for each grammatical unit generated, it is usual that this option is combined with the *Realize selectively* option.

7.5.2.3 Show Why System Is Firing

Prints the *last* feature that caused the entry condition of the system being entered to become satisfied.

7.5.2.4 Show Disabled Candidate Systems

Systems may be *disabled*. A disabled system can be entered but it has no effect. If this flag is on a message will be printed whenever a disabled system is entered. The user may disable and enable systems at will by using the appropriate commands that are obtained by right-clicking on any mouse-sensitive system name that appears in the window interface (cf. Section 6.5.2). The commands INSPECTOR:<:Disable system *system-name*> and INSPECTOR:<:Enable system *system-name*> can also be given. Disabling a system is sometimes convenient while debugging linguistic resources. You may load several different versions of a system (as long as they have distinct names!). By disabling all but one of the versions you may check out the functionality of the enabled system. Each different version of a system in turn can be checked out this way. This is easier than reloading a new grammar each time you want to check out the effect of changing only one system. See also Section 7.8.1.

7.5.2.5 Show System Entry Dependencies

Shows what systems are ready to be entered, and the system which is actually selected from that list to be entered.[6]

7.5.2.6 Show Preselections

If this flag is set, at the beginning of each cycle or pass through the grammar, the preselections that will be enforced during that cycle are shown. This information is given in two forms: first, as a list of the grammatical features that appeared in the preselection realization statements that called for the pass, and second as the *path augmented* list of preselections derived from the first list. This latter contains all the features that would need to be selected in order to ensure that those of the first set were also; i.e. for each feature preselected, all those features on the path that runs from the starting system (that of *rank*) to the feature required, need to be selected also - this is done automatically and is called *path augmentation*.[7]

7.5.2.7 Show Immediate Realizations

The grammatical features that form the output of grammar systems may have *realizations* associated with them. As described above, these realisations are applications of the functional operators by which the structural output of the grammar is built up: i.e. during the generation process the operators that are executed impose constraints upon the structural output. With the *Show Immediate Realizations* flag on, a message containing the system name and the realization that is being performed is printed as soon as its associated feature is selected.

7.5.2.8 Show Lexical Selection

Prints information about the choice of lexical items from the lexicon; i.e. whenever a *lexify* realization statement is performed, a message of the form *The chosen word is: X* is produced. In addition, when certain lexical items are decided on purely grammatical grounds (for example, verbal auxiliaries), an account of their determination is produced. This account is in

[6]From this one can determine the effect on system entry that the system dependencies, defined in the global variable `system-dependencies`, have. These system dependencies are responsible for helping to decide which of several apparently equally eligible systems should be entered. Relying on particular orders is therefore possible, although not recommended. The forms for defining such dependencies are described in Section 12.2.11.

[7]Note that since KPML does not attempt to provide a semantically complete internal representation of the subsumption lattice entailed by the systemic network (this is still beyond the practical capabilities of available feature logic implementations: cf. (Henschel 1995)), it approximates full paths by tracing backwards (i.e., leftwards in the systemic network) until a feature participates in a disjunctive entry condition. Guidance is then given for preselections through such entry condition: see Section 12.2.7.

terms of the lexical *feature list* of the constituent, the *classify list*, the *outclassify list*, the list of lexical terms conforming to the classify list, and the revised list after filtering with respect to the outclassify list.

7.5.2.9 Show Lexical Features

Yet more lexical selection tracing.

7.5.2.10 Show Ordering Constraints

Prints information showing the effect that the *default order* of constituents has on the resulting ordering. Default orders are specified in a file named `ordering-constraints.gram` (see Section 12.2.12).

The information that is produced concerns four phases of ordering:

- Potential Default Orderings: shows just those default orders defined by the grammar that might be relevant to the presently generated structure;
- Added Default Orderings: removes any of the possibly relevant default orders that in fact violate or are inconsistent with the positive statements of ordering made by realization statements during the traversal of the grammar;
- Compiled Precedence Constraints: ordering is defined in terms of two kinds of information - precedence and adjacency. The partition realization operator provides precedence information only; the order realization operator provides both precedence and adjacency information. The information given here is a list of pairs showing the combined precedence information taking into account the filtered default orderings shown previously;
- Compiled Adjacency Constraints: the information here is a list of pairs showing the adjacency information taking into account the filtered default orderings shown previously.

Throughout these displays *conflation aliases* are used for the function bundles that are ordered rather than the literal function names that may have been used in actual realization statements. A Conflation alias is the first function in a function bundle.

7.5.2.11 Show Ordering Events

When this flag is set, realizations having to do with ordering, i.e. Order, OrderAtFront, OrderAtEnd, Partition, are printed as they occur, along with the system responsible for their being performed.

7.5.2.12 Show Ordering Results

Setting this flag produces information concerning the order that the grammar has decided upon for each level of structure. At the end of each cycle, i.e. when each level of structure has been completed, *Orderings* information for the cycle is shown in two forms:

1. Function structure: this is an ordered list of the function bundles (i.e. the constituents that have been constructed by function conflation and insertion for the cycle) showing those bundles' member functions.

2. Realization: this is the result string that can be produced on the basis of the information accumulated so far; i.e. lexicalisations will appear as actual words but constituents that still need to be generated by further passes through the grammar are shown in terms of their function bundles e.g. (FUNCTION1+FUNCTION2+ "is" FUNCTION6+FUNCTION7)

7.5.2.13 Show Associations

The association between grammatical functions and entities in the knowledge base that permits inquiries to interrogate the appropriate parts of the knowledge during generation are maintained in the *function association table* (Section 12.2.7). The level of indirection that this introduces permits inquiries always to be expressed in terms of the grammatical functions defined in the grammar; these functions' case-by-case reference to appropriate knowledge representation entities is thus ensured by the function association table.

When this flag is set information concerning the establishment of these grammatical function and knowledge base hub associations is given during generation. In particular, whenever a hub association is created by *copying* one functions association on to another, the following kind of message appears:

```
In system SYSTEM-NAME, copyhub FUNCTION1 --> FUNCTION2.
```

In addition, at the end of each pass through the grammar—or, if the 'Show System and Inquiry Activity' flag is also set (Section 7.5.2.2), as they occur in a separate window—the function association table entries are shown. Each of these entries consists of five fields of information,

- Function: the grammatical function participating in the association;
- Concept: the knowledge base hub participating in the association;
- Presentation-spec: a specification of the particular details of the hub that are to be expressed in this mention of it;
- Term set: the set of possible lexical items that the grammar has selected as being potentially appropriate for the expression of the concept in the current mention of it;
- Term: the actual lexical item that came to be used for the concept.

7.5.2.14 Show Inquiry Answer Source

When this flag is set and the resources are being used in 'implemented' mode (the default case: see Section 7.4.7), the source of each response that an inquiry receives is displayed. The possible sources are:

- operator code: the implemented form of the inquiry ran and returned a result,
- SPL keyword: the response was directly specified in the input SPL,
- default: neither of the previous two options applied and a default response was used.

7.5.2.15 Show entailed inquiry response

Prints a message whenever a response to an inquiry operator entailed by some preselection is used.

7.5.3 Generation process control options

7.5.3.1 Realize Selectively

As sketched in Section 2.3.1, generation proceeds in cycles through the grammar network. This flag permits the user, at the beginning of each pass through the grammar, to decide whether to perform that cycle or not - i.e. whether to realize the constituent that cycle is responsible for generating or to skip realization of that constituent. This is convenient for debugging when you are trying to examine what happens in certain constituents but do not care about others. It also provides convenient pre-given points for pausing so that tracing modes can be altered.

The form of the option that is presented to the user is:

```
Do you want to realize
   ((FUNCTION1 FUNCTION2 ... FUNCTIONm)
    ((Preselection-type-realization FUNCTIONi FEATURE)
     ...
    ))
```

This presents the members of the function bundle defining the constituent that is about to be realized and the preselections, classifies, lexifies, and inflectifies defined for that constituent via constraints imposed on the member functions of the bundle during the just completed traversal of the grammar network. An actual example would be:

```
Realizing Bundle:
                ((VOICE FINITE TEMPOO)
                   ((INFLECTIFY FINITE PAST-FORM)
                    (INFLECTIFY FINITE SINGULAR-FORM)
                    (INFLECTIFY FINITE THIRD-PERSON)
                    (CLASSIFY FINITE  OUTCLASSIFY-NEGATIVE)
```

```
(CLASSIFY FINITE OUTCLASSIFY-REDUCED)
(CLASSIFY VOICE OUTCLASSIFY-NEGATIVE)
(CLASSIFY VOICE BEAUX)))
```

This display also shows which grammatical function contributes which constraint in the constraint set as a whole. That is, the constituent shown here is constrained to possess the lexical features [Past-form], [Singular], [Third-person], and [BeAux] and not to have the lexical features [Negative] and [Reduced], and, in addition, we know that it is the component grammatical function Finite which brought the constraints concerning past form, singular form, third person, and reduced, while it was the function Voice that brought the constraint 'BeAux'. Both functions were also constrained not to be negative.

7.5.3.2 Realize until constituent number

This flag, another exception to those that take `t` or `nil` as value, expects an integer identifying a particular constituent in the structure of a generated sentence to be given. These numbers can be directly read off the print form of the grammatical functions as they appear, for example, in their graphed or textual form (see, e.g., Figure 7.2). They represent the network *traversal cycle* that introduced the constituent into the structure. When set to an integer, generation will proceed with whatever other flag values have been set until the identified constituent is reached. Then generation will pause and the *Generation Modes* menu will be presented with the option 'realize selectively' already activated. This can be used for quickly locating and going to a problematic constituent during debugging of the grammatical resources. It is not necessary, as was the case with Penman, to step through the previously generated constituents by hand.[8]

Note: this option is now incorporated implicitly in the command options for generated strings (Section 10.3), which provides a more convenient and quicker way of achieving the same effect without the user needing to identify the constituent number.

7.5.3.3 Single Step

Causes the generation process to pause at the end of each inquiry. At this point, various data structures can be examined. A menu prompts for continued execution. This can best be combined with the 'Show System and Inquiry Activity' flag to step through the generation process inquiry by inquiry: in fact, since single stepping without at least this information is probably not useful, **unless the 'System and Inquiry Activity' flag at least is also set, single stepping will *not* occur.**

The user dialog box that is brought up on single stepping also allows several other operations in addition to continuing. In particular, generation may be aborted or the generation modes (cf. the command DEVELOPMENT: <Generation Modes>) altered.

[8] It appears currently not possible to give a `nil` setting once a number has been given; as a workaround, the number zero can be given. This has the same effect as `nil` since traversal cycle counting starts from 1 and so a cycle number zero is never found.

Note that in all cases either 'yes' or 'no' must be selected finally in order to exit from the dialog box.

7.5.3.4 Enter Debugger on Warnings

Whenever this flag is set, all warning conditions that are reported to the user are followed by an immediate entry to the Lisp debugger. This is clearly not intended for the normal kind of resource debugging that users will carry out, but provides one fairly straightforward way of suspending the generation process temporarily. The 'continue' option offered by Lisp will normally continue the generation process.

7.5.4 Generation result focusing modes

In this section we describe the focusing operations that allow information to be picked out of the final and interim results of generation. This includes not only the final strings generated, but all partial results (such as syntactic structures, associations of syntactic and semantic objects, inquiry responses, chooser decisions) that are reached during generation. Unlike the generation tracing modes described below, it is normally the case during result focusing that the user actively specifies particular linguistic events that are to be monitored during generation. This is done by selecting the tracing options offered for objects of particular linguistic types. These options are also described here.

7.5.4.1 Cumulate System and Inquiry Activity

This flag provides the basic option of result focusing. Three levels of cumulation are possible:

- off (nil), where no information is preserved (the default),
- traced, where only information concerning explicitly traced linguistic objects and events is preserved,
- all, where all information concerning traversal of the systemic network during generation is preserved—i.e., systems entered, features selected, choosers used, inquiry responses received.

The latter extends on the information available when the flag <Generation Modes>: 'Show System and Inquiry Activity' is set. Examples of use are given in Section 7.5.5.

7.5.4.2 Update Example Record Fields

Whenever set, this flag causes the generation of examples to update any prestored information maintained in the records of those examples.

This flag must be set if creating a fully recorded set of examples that support the use of example selection by features (see Section 6.2.1), retrieval of selection expressions, etc.

When this flag is not set, then the generation history of an example is *not* recorded: only the generated string and associated rich mouseable structure is transferred to the example record.

The use of examples and example records is described fully in Chapter 10.

7.5.5 Viewing focused results

Whereas generation tracing (Section 7.5.2) will immediately show any information traced (either in the *Development* window itself or in special purpose windows brought up for particular types of information), cumulated information is maintained in the background and is only displayed when requested.

This is done by issuing the command DEVELOPMENT:<Show Cumulative History>. This brings up a *Cumulative Generation History* window that contains the information selected for cumulation: typically system or inquiry activity. An example of the window is shown in the lower half of Figure 7.5; this example is discussed below.

The *inspector* command <:Clear Cumulative History> causes the cumulative history to be cleared; this can be useful before bringing up a history window if it already contains unwanted history.

7.5.5.1 The cumulative history window commands

The cumulative history window has a few specific commands of its own described as follows.

7.5.5.1.1 Redisplay The command CUMULATIVE-HISTORY:<Redisplay> forcibly causes the contents of the window to be redisplayed; this might be useful if generation has been continued and a new state of affairs is to be shown in the history window.

7.5.5.1.2 Clear history The command CUMULATIVE-HISTORY:<Clear history> clears the window.

7.5.5.1.3 Display options The command CUMULATIVE-HISTORY:<Display options> controls what kind of information is given concerning inquiries. Any number of the following modes can be selected:

- ***hubs***: when this mode is set, then the symbol identifying the semantic hubs (typically SPL terms) used in the particular inquiry ask or identify are shown in the display.
- ***id***: when this mode is set, the unique identifier of the semantic hub (or SPL term) is shown in the display.
- ***formal-parameters***: when this mode is set, the formal parameters used in the inquiry call as specified in the particular chooser at issue are shown in the display.

Setting none of these modes gives information equivalent to that shown in the *Generation History* window when the tracing flag 'Show System and Inquiry Activity' is set.

7.5.5.1.4 Quit The command CUMULATIVE-HISTORY:<Quit> exits and removes the history window.

7.5.5.2 Example of use

Figure 7.5 shows an example of the use of the cumulated generation history window. In this example, the chooser for the system NOMINAL-LIKE-GROUP-CLASS (graphed in the top window in the figure) has been traced (see Section 7.6) and a sentence has been generated. Issuing DEVELOPMENT:<Show Cumulative History> then brings up the window shown in the bottom of the figure. Since the formal parameters for the inquiries are shown in the chooser graph, the display options have been set to `:hubs` only. The cumulative history window shows for each time the traced chooser was used, the system with which it is associated, the feature selected at that time, the inquiries asked, and their parameters and response. It is therefore straightforward to recover which path was taken through the chooser during each of its instantiations during generation and why that path was selected.

The first box in the *Cumulative Generation History* window, for example, gives here the following information. In the system NOMINAL-LIKE-GROUP-CLASS the feature 'nominal-group' was selected. This was because the inquiries `property-q` and `quantity-q` were both asked of the semantic term `HEAD` and received the responses `notproperty` and `notquantity` respectively. Comparing this with the chooser shown in the upper part of the figure, we can see that the first inquiry to be asked, `property-q` has two possible responses (`property` and `notproperty`) and is asked of the grammatical function 'Onus'. The function association for 'Onus' must therefore have been the semantic term `HEAD`. Following the obtained `notproperty` path in the chooser leads on to the second inquiry posed. The response here, `notquantity` then results in 'nominal-group' being selected as seen.

The 7 instances of NOMINAL-LIKE-GROUP-CLASS shown in the *Cumulative Generation History* include examples of the three possible paths through the associated chooser.

All of the semantic terms shown in the window are mouse sensitive supporting further information inspection.

7.6 Activating result focusing and tracing

7.6.1 Activation of tracing

Additional linguistic object type-specific commands are provided for activating selective tracing and information cumulation during generation. These are generally available by direct typing of the command name in the interaction window, or by right clicking on an appropriately object.

When selected, the use of a linguistic object (system, chooser, or inquiry) can either be reported during generation *tracing* or by showing the cumulative generation history following generation. If the 'Cumulate System and Inquiry Activity' flag is not set, then the information will be produced in tracing mode in a *Generation History* window (cf. Figure 7.4). If the cumulation flag is set, then no tracing information is produced until the user explicitly calls

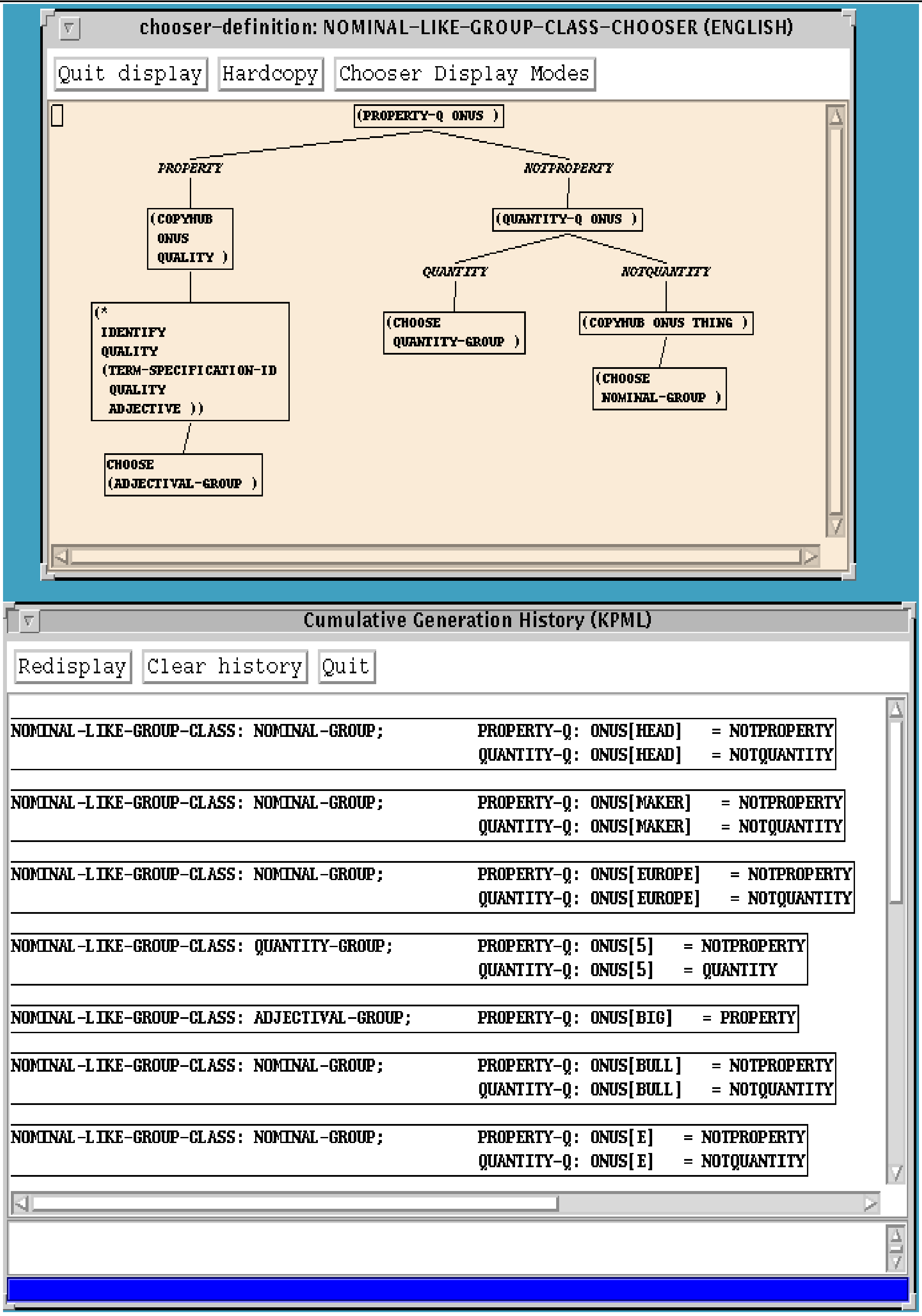


Figure 7.5: Example of using the cumulative generation history

for it with DEVELOPMENT:<Show Cumulative History>.

The relevant commands are as follows.

7.6.1.1 Individual system tracing

The right-click menu command <trace system> causes the clicked upon system to be added to the list of currently traced systems. When any system on this list is entered, the following information is produced in the *Development* result pane:

- the preceding system and feature whose selection was responsible for the entry conditions of the traced system being fully met,
- the feature selected in the traced system.

In addition, if the 'Cumulate System and Inquiry' flag is set, the information that the traced system has been entered and which feature was selected is added to the cumulative history.

Individual systems can be removed from the tracing list by selecting the matching <untrace system> command.

7.6.1.2 Individual chooser tracing

The only meaningful way of tracing choosers is to trace the full set of inquiries that any chooser uses. The right-click menu command <Trace inquiries of chooser> therefore causes all the inquiries used by the clicked upon chooser to be traced. The operation of tracing inquiries is described in the following section. All inquiries for a chosen chooser can also be removed from tracing by issuing the corresponding <Untrace inquiries of chooser> command.

An additional option that individual chooser tracing supports when the 'Cumulate System and Inquiry Activity' flag is set is to take the cumulated inquiry information for a chooser (that can be shown by the DEVELOPMENT:<Show Cumulative History> command), and to fold this into the graphical chooser display available under INSPECTOR:<Print Chooser> (Section 6.3.2.2) or any of its equivalents. Setting the 'Cumulate System and Inquiry Activity' to `:all` naturally provides this option for *all* choosers.

For this option to be activated, the flag CHOOSER-GRAPH:<Chooser Display Modes> 'Generation Paths Shown' has to be selected (which it is by default in a newly loaded KPML system).

An example of the default behavior when a chooser has been traced and graphed is shown in Figure 7.6. This shows one traversal through the English PRIMARY-TENSE chooser. The definition of this chooser (i.e., without traversal paths) was shown in full in Figure 6.10. The present figure shows just the middle portion of the chooser that was actually effected during the traversal at hand.

The inquiry query and response shown in a traced and graphed chooser can be varied by altering the setting under the CUMULATIVE-HISTORY:<Display Options> command (Section 7.5.5). In the present example, the options of *hubs* and *formal parameters* have been selected. This shows for each inquiry, all of its formal parameters and their semantic associations. The semantic results of identifying inquiries are also shown. The central portion of the chooser shown in Figures 6.10 and 7.6 can therefore be compared node for node. The path taken through the chooser is also highlighted (by being in color on color screens, and by a shade of grey on monochrome screens). We can directly see that, in this case, the traversal path follows the arcs 'noncounterfactual', 'extensional', 'notlogicotemporal', and 'precedes'; this results in the choice of the grammatical feature 'past'. Chooser nodes that do not lie on the traversal path (for example, that below the final 'notprecedes' arc) are shown as they are in the straightforward chooser definition graph.[9]

Note that this combined graphical and traversal option does not make the use of the cumulative history window redundant. The cumulative history window (as illustrated in Figure 7.5) gives an overview of several instantiations of any given chooser—as many instantiations as were invoked during generation since the last clear of the cumulated history. Asking for a graphically displayed chooser for which several instantiations are on record brings up a set of windows, one for each instantiation. Each window shows one traversal through the selected chooser. For choosers that are used frequently, this may become less simple to interpret than the simple overview given in the history window.

7.6.1.3 Individual inquiry tracing

The right-click menu command <trace inquiry> causes the clicked upon inquiry to be added to the list of currently traced inquiries.

When the 'Cumulate System and Inquiry Activity' flag is unset, using any inquiry on this list results in the normal inquiry related information being displayed directly in the *Generation History* window as illustrated in Figure 7.4.

When the 'Cumulate System and Inquiry Activity' flag is set, full information concerning the system, inquiry formal and actual parameters, the inquiry response, and the feature selected in the system are cumulated for display in the cumulative generation history if required. In this case, no information is produced in a *Generation History* window.[10]

Individual inquiries can be removed from the tracing list by selecting the matching <untrace inquiry> command.

7.6.2 Clearing tracing selections

The command DEVELOPMENT:<Clear Tracing Option> brings up a menu from which particular classes of tracing selections, including those of the previous subsection, can be selected for clearing. The options in full are:

- Clear all tracing

[9] Note that this particular chooser definition contains an oddity: the 'notprecede' option for `precede-q` that does not lead anywhere; this can be ignored here.

[10] Note that displaying a chooser graphically when only some of its inquiries have been traced and the flag 'show generation paths' is set can lead to an incorrect graph. This can be avoided by tracing the chooser rather than inquiries.

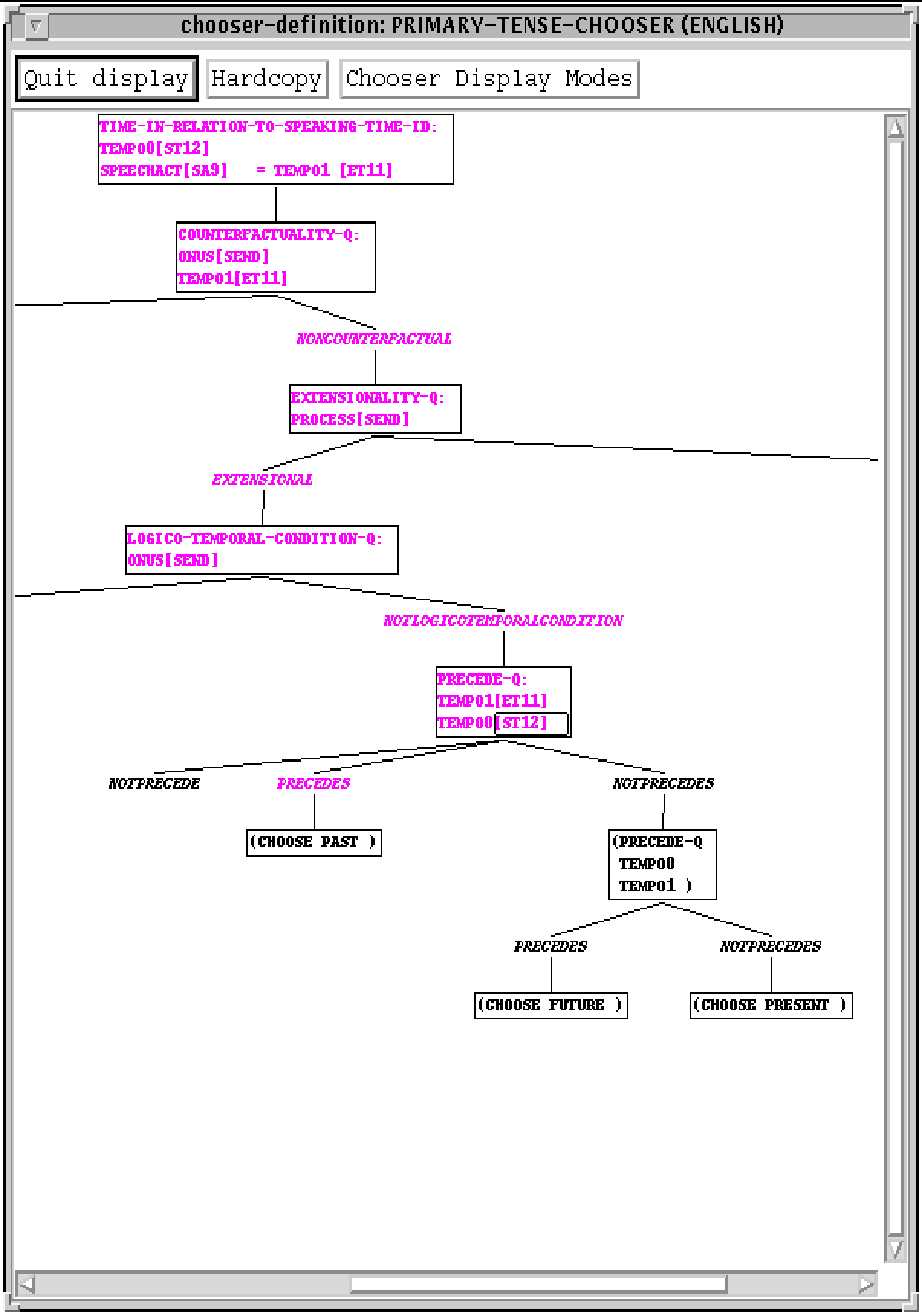


Figure 7.6: Example of graphed chooser showing generation path

- Clear traced systems
- Clear traced choosers
- Clear traced inquiries
- Clear paused inquiries (cf. Section 7.8.2)
- Clear collected features (cf. Section 6.2.3.4)
- Clear resource graph stop points (cf. Section 6.2.3.5)

7.7 Graphical representation of systemic network traversal

In addition to the particular tracing of generation paths through individually selected choosers, KPML also maintains information about the traversal path through the systemic network as a whole.

7.7.1 Traversal and resource graphs

When the grapher display mode RESOURCE-GRAPH:<Display Modes> 'Show previous generation path' is set, then any systemic networks that are graphed (e.g., with DEVELOPMENT: <Graph Grammar>) also show highlighted for each system the last feature that was selected in that system during some traversal. This display mode can also be set from the commands DEVELOPMENT:<Grapher Display Modes> and CHOOSER-GRAPH:<Chooser Display Modes>. The default on new loading of the KPML system is that it is activated.

Note that it is the last feature selected in a system on *any previous* traversal of the systemic network that is highlighted. This may be confusing if one is only interested in seeing the most recent traversal path through the network. This is because features will be highlighted in systems which were not even used in the most recent traversal.

It is also possible to focus on just the last traversal so that only those systems that were actually used in the last traversal have their feature selections highlighted. Whenever the DEVELOPMENT:<Generation Modes> 'realize selectively' flag is set (Section 7.5.2), then only those features selected during the last traversal are highlighted. The realize selectively flag forces the generation process to pause after each grammatical unit generated—i.e., after each traversal cycle through the systemic network; graphing the systemic network at this point will highlight the features from the selection expression of that grammatical unit only.

An example of using the 'show previous generation path' mode for both the systemic network and choosers is given in Figure 7.7.

Here we see a trace of a traversal through the TENSE region of the Nigel grammar of English. The larger graph in the lower portion of the figure shows an extract of the grammar systemic network beginning with the system SECONDARY-TENSE.[11] We can see that on the last traversal through the grammar, the feature 'secondary-tense' was selected, leading on to two systems:

[11] The linguistic details and motivations for this treatment of tense are based on (Halliday 1976) and are given in Matthiessen (1984).

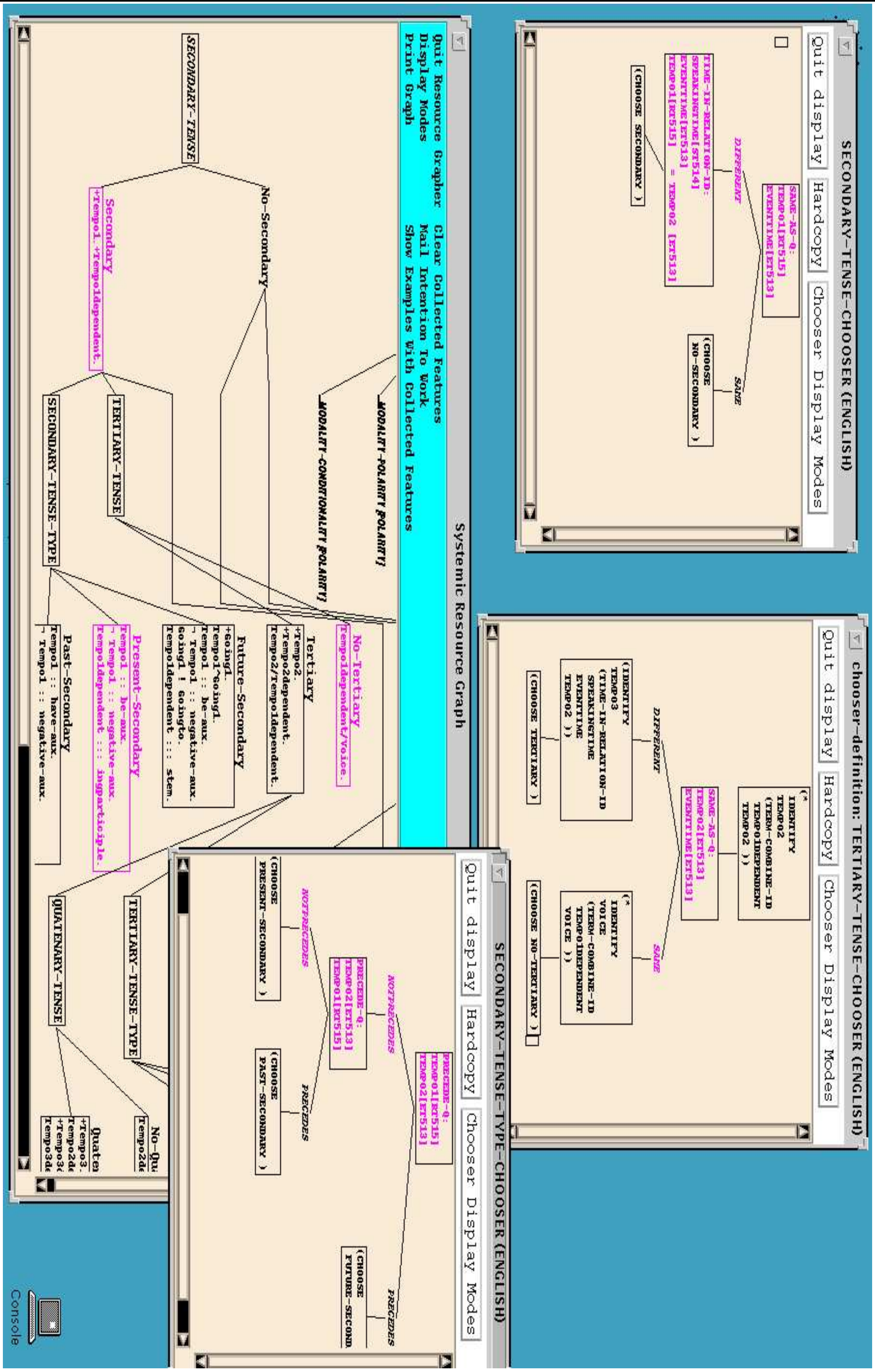


Figure 7.7: Example of generation path tracing

SECONDARY-TENSE-TYPE and TERTIARY-TENSE. The highlighting of the selection expression in the graph tells us that the features 'present-secondary' and 'no-tertiary' respectively were chosen in these two systems. The tense selected was therefore some primary tense followed by a present secondary tense; in other words, one of:

am/is/are running
was/were running
will be running

Examples showing these realizations of the selected tense could also have been displayed by collecting the critical tense features and then invoking, for example, RESOURCE-GRAPH:<Show Examples with Collected Features> (cf. Section 6.2.3.4).

Now, if the user wishes to find out the temporal semantic conditions to which such a tense selection corresponds, then left mouse-clicking on the features 'secondary', 'no-tertiary', and 'present-secondary' brings up the three choosers responsible, shown in the Figure from left to right. Since the previous generation path mode is activated, these chooser graphs also have folded into their display the inquiry questions and associated semantic specifications that held for the traversal in question. The semantic conditions can then be immediately collected; i.e., with the time intervals `TEMPO`n denoting a sequence of reference times that relate the speaking time to the `EVENTTIME`:

`TEMPO1 = RT515` (a time interval)
`EVENTTIME = ET513` (a time interval)
`TEMPO1` $\neq$ `EVENTTIME`
`TEMPO2 = ET513`
`TEMPO2 = EVENTTIME`
`TEMPO1 not-precede TEMPO2`
`TEMPO2 not-precede TEMPO1`

That is, the reference time and the event time are overlapping but not equal, and there are no further reference times intervening before the event time.

Any portion of a specified systemic network can have its semantic commitments displayed in this way, thereby providing relatively quick access to the semantic motivations for particular grammatical choices or forms according to the linguistic resource used.

7.7.2 Dynamic traversal tracing

It is also possible to inspect the paths taken through the systemic network dynamically during generation. The command INSPECTOR: <:Traversal Graph> brings up a window in which features are added dynamically as they are selected on traversal. The dependency relations between features selected are also shown, producing an extracted graph from the systemic network as a whole. The nodes of the traversal graph consist of the feature selected and the system to which that feature belongs. Both system and feature are mouse-sensitive in the normal ways (cf. Section 6.5.2 and 6.5.3 respectively). In addition, the options for pruning a systemic network graph described in Section 6.2.3.5 also hold for the dynamic traversal graphs.

A sequence of successive views of a traversal graph taken at the outset of generation for an example is shown in Figure 7.8. This contains the first 7 steps in generation. The latest growth is in each case shown in a different colour or shade of grey. Note that this growth would normally be shown in a *single* window: the cumulative view shown here is for illustrative purposes only. The graphs show the progressive refinement in linguistic specification that occurs when the systemic network is traversed. The first decision indicates that clauses should be generated, this is then refined to the subtype of clause 'clause', which is in turn refined to the subtype of clause 'full', etc. The graph gives more information than the straightforward list of a selection expression since a systemic network includes conjunction in its connectivity definition: thus, the penultimate graph here shows that the unit being generated is both 'nonconjuncted' *and* 'mood-unit', and these subtypes both further specify the type 'clause-simplex'. Finally, in the last snapshot taken we can see that the type 'independent-clause-simplex' is dependent on both 'independent-clause' *and* 'mood-unit'; note also that here two new nodes appear ('independent-clause' and 'independent-clause-simplex') with respect to the situation shown in the preceding graph. This is because the first of these is a gate (i.e., a system of one choice). The traversal path is therefore free to go directly to this gate's successors whenever they may be selected.

Traversal graphs can be useful for exploring how particular sets of features are related to one another. The information given is equivalent to the selection expressions obtained from graphed structure nodes (Sections 7.9.3.1 and 10.2.5) or from constituents of a generated string (Section 10.2.5.1). The selection expressions shown by these other methods are displayed as simple lists of features however. This means that the dependencies between features will not be clear unless one is reasonably familiar with the resources being used.

More selective areas of traversal can be selected by combining traversal graphs with *collected features* (Section 6.2.1.3). When features have been collected, a started traversal graph will *only consider features dependent on those collected features.* If no features are selected that are dependent on the collected features, then the traversal graph will show no growth. An example of dynamic traversal with three collected features ('temporal-location', 'temporal', and 'declarative') is shown in Figure 7.9; this example also shows the contribution of graph pruning: some of the descendents of the feature 'declarative' have been removed from the graph.

Dynamic traversal will keep on adding to the displayed graph as long as that graph is the most recently started and as long as generation continues. It is usually desirable to have a trace of the features selected during a single traversal: therefore use of traversal graphs is normally to be combined with the 'Realize selectively' generation mode (cf. Section 7.5.3.1). A given traversal graph will in any case only show features from a single rank (i.e., it will not mix features selected from, e.g., clauses and nominal groups), since each node can only show at most *one* selected feature.

Note that dynamic traversal is only activated when the result focusing flag 'Cumulate System and Inquiry Activity' (cf. Section 7.6) is set to `all`. In addition, the traversal graph command should be given *prior* to starting generation—otherwise it may become confused about the state of generation.

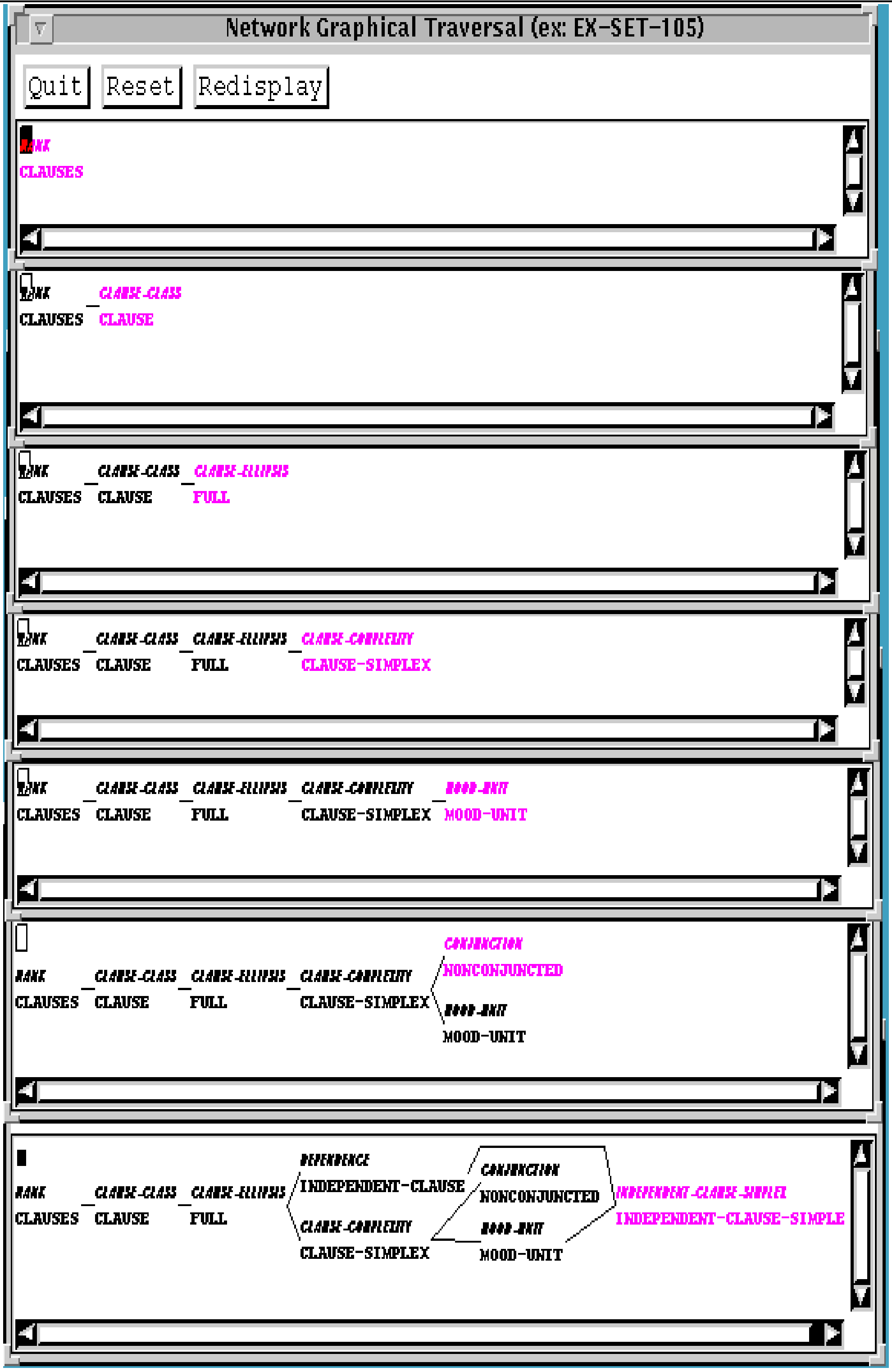


Figure 7.8: Successive views of the features selected during network traversal

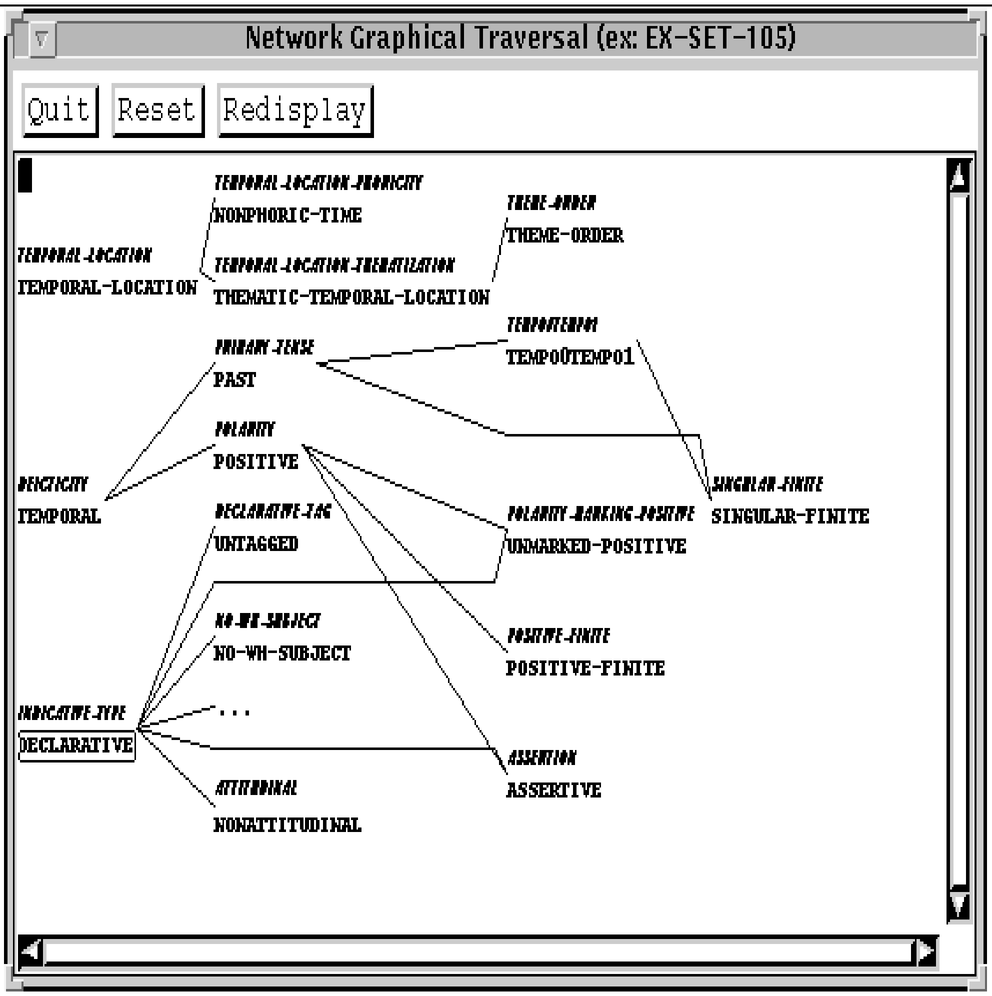


Figure 7.9: Example of selective traversal tracing by collecting features

7.8 Additional generation process control options

7.8.1 Disabling and enabling systems

Any system may be right-clicked upon to produce a further menu of operations. Two commands here, <Disable system> and <Enable system>, have consequences for the generation process. When a system is disabled, it is temporarily removed from those systems that are considered during generation. That is, such systems will not be entered during generation and no feature from such systems will be selected. A disabled system may subsequently be re-instated by a corresponding <Enable system> command.

7.8.2 Pausing on inquiries

The two commands DEVELOPMENT:<:Pause on inquiry> and DEVELOPMENT:<:Stop pausing on inquiry> provide a means of generating until a particular inquiry, or member of a set of inquiries, is reached. Generation then pauses (entering the Lisp debugger).

7.8.3 Pausing and restarting generation

At any time during generation, the generation process may be paused by issuing the command DEVELOPMENT:<Pause> and restarted with the command DEVELOPMENT:<Resume>.

Generation can be abandoned at any time with the command DEVELOPMENT:<Abort>.

7.9 Inspecting the results of generation: Graph Structure

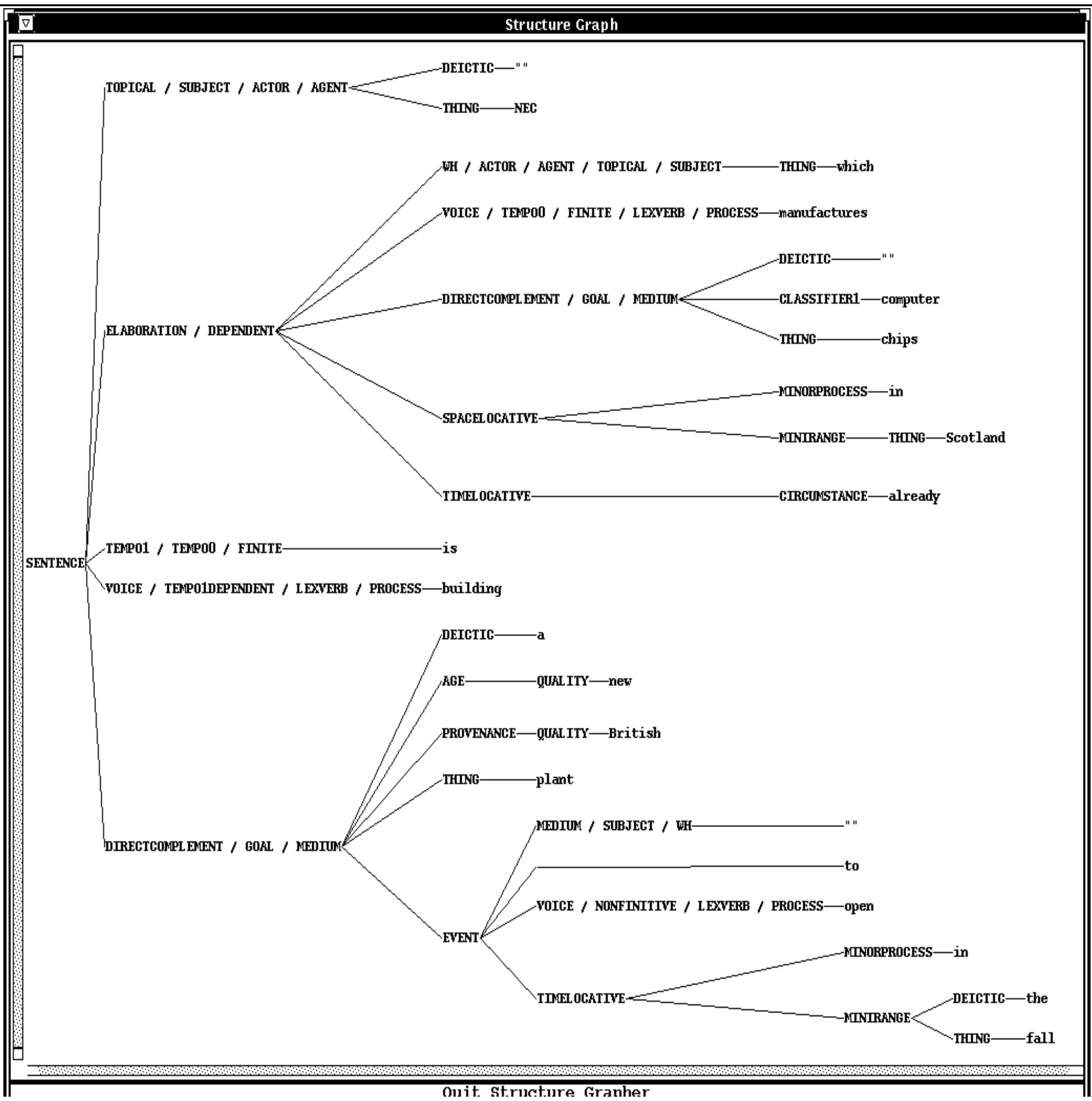
7.9.1 Introduction to structure graphs

The most direct way of inspecting the results of generation, including the decisions that were made to get to that result, is by graphing the sentence structure. The command DEVELOPMENT:<Graph Structure> brings up a graph of the last structure that was generated (or part thereof, if generation is incomplete but sufficient information for a graph representation was obtained). This can be used as the starting point for inspecting all aspects of the generated result.

An example of such a structure is given in Figure 7.10. The grammatical structure is shown 'sideways' as a graph with the largest constituent placed on the left and successively decomposed into its constituent parts moving to the right.[12] Each constituent is shown in terms of the functions that go to make up its function bundle. The structure shown is the Nigel example sentence REUTERS11 from the ISI 1993 Penman release.[13]

[12] This orientation can be changed, see the options below; it is, however, probably the most suitable for systemic functional structures due to the long functional labels that constituents receive.

[13] It is also possible to graph individual constituents from a generated structure. This is managed however via the example record and the facilities offered there for structure graphing: cf. Sections 10.2.5 and 10.3.



ISI Penman example: Reuters11:

> "NEC, which manufactures computer chips in Scotland already, is building a new British plant to open in the fall."

Figure 7.10: Example of structure graphing

Each of the non-terminal nodes of the structure graph are mouse-sensitive. Clicking on them gives a further menu that allows diverse information concerning the generation process responsible for the structure to be shown. One very commonly used option is that which shows the *selection expression*, i.e., the list of grammatical features that were selected for its generation, of that node. Other possible operations allow inspection of various constraints that were used to construct the structure. They are described in full in Section 7.9.3.

In all cases it is important to note that it is the *full internal data structure used during generation* that is inspected. The options here provide, therefore, the most detail that can be obtained concerning the generated structure. This differs from the superficially very similar looking graphed structures that may be produced from stored examples or example records. These latter, as described in detail in Chapter 10, contain only a subset of the full information, usually leaving out generation-process internal constraints that could be reconstructed from the definition of the grammar. As an added reminder of the difference, when available, the result of generation structure graph is printed blue and the example record structure is printed black. Example records are already very large: at present the space-cost seems to outweigh the information loss. The missing information can always be reconstructed be re-generating the particular example.

When the strings selected for the realization of constituents are shown in a structure graph, it should be noted that these depict the selected lexemes *before* some of the final string operations summarized in Section D are performed.

A postscript file of the graphed structure suitable for printing or including as figures can be created in the normal way under the GRAPH:<Print Graph> command (cf. Section 6.2.2).

The structure graph command <Quit> exits from the structure graph and then removes the window.

7.9.2 Structure Grapher Options

Issuing the command STRUCTURE-GRAPH:<Options> brings up a menu analogous to the corresponding menu for systemic networks discussed in Section 6.2.1.5. There is only one content-oriented option for structure graphs, however:

- ***Highlight currently/last generated node***: when this flag is set (the default), the grammatical constituent that was most recently generated (or, if graphing is invoked prior to completing its generation, the unit still being generated) is highlighted in the structure graph. If generation is complete, then no node is highlighted. This option is particularly useful if an unexpected problem that suspends generation occurs and no tracing was being produced to indicate where in the generation process one was.

Figure 7.11, for example, shows four successive views of a structure during generation: each view was produced immediately after completing a single grammatical unit and prior to commencing the next (by means of the realize selectively flag under DEVELOPMENT:<Generation Modes>: Section 7.5.2). The node highlighted in each case, therefore, is the larger grammatical unit *immediately containing the unit that is about to be realized.*

In the first snapshot, the generation process has just produced the structure for the 'Sentence' and is about to commence generating this unit's substructure. The next unit to be realized can generally be recognized as the topmost unfilled child of the highlighted unit: i.e., in the first snapshot, the grammatical constituent labelled 'Topical/Medium/Subject'. In the second snapshot, the 'Topical/Medium/Subject' constituent has been generated and generation is about to commence on its (only) child, 'Thing'. Similarly in the third snapshot, which has moved on to the (only) child of the 'Thing' grammatical unit, the 'Stem'. This latter grammatical unit is immediately realized (probably morphologically), and does not need another traversal of the systemic network. In the fourth and final snapshot, therefore, the child of 'Stem' ('Head') has already been filled in and the 'Topical/Medium/Subject' constituent of the sentence as a whole is complete. The containing grammatical unit then reverts to the 'Sentence'. The first and second subconstituents of the 'Sentence' have now both been filled, and so the next unit to be realized is that labelled as 'Spacelocative'.

The remaining structure grapher options concern layout and production of hardcopy versions of the structure graph:

- ***Send created postscript files to printer***: if this flag is set, any postscript file produced by invoking the <Printgraph> command is sent directly to the default printer rather than simply being left in the hardcopy directory. The printer command used is `lpr`.
- ***Structure graph orientation***: this flag controls the orientation of graphs; the possibilities are `:horizontal` (the default) and `:vertical`.
- ***Vertical scaling***: the distance between elements vertically.
- ***Hardcopy vertical scaling***: the distance between elements that will be used in postscript files for hardcopying.
- ***Hardcopy directory***: the directory where postscript files for hardcopying will be stored (when the *printgraph* menu option is used).
- ***Hardcopy with header***: this flag determines whether header information (containing the current language, and, if hardcopy, the date of production of the graph) is shown in the graph or not.
- ***Suitable for eps***: when set, this flag causes hardcopy versions of graphs to be produced in 'single page' mode. Postscript files for inclusion in text documents should normally be produced with this flag set, otherwise extended postscript will not produce the right results.

7.9.3 Operations available on structure constituents

The full list of available inspection options for particular constituents in the graphed grammatical is as follows. The options are reached by left-clicking on the desired grammatical constituent. With the exception of selection expressions, the information presented always appears in the *Inspector* window: this is because this information is typically used as the starting point for further information searches of the kind described under information chains

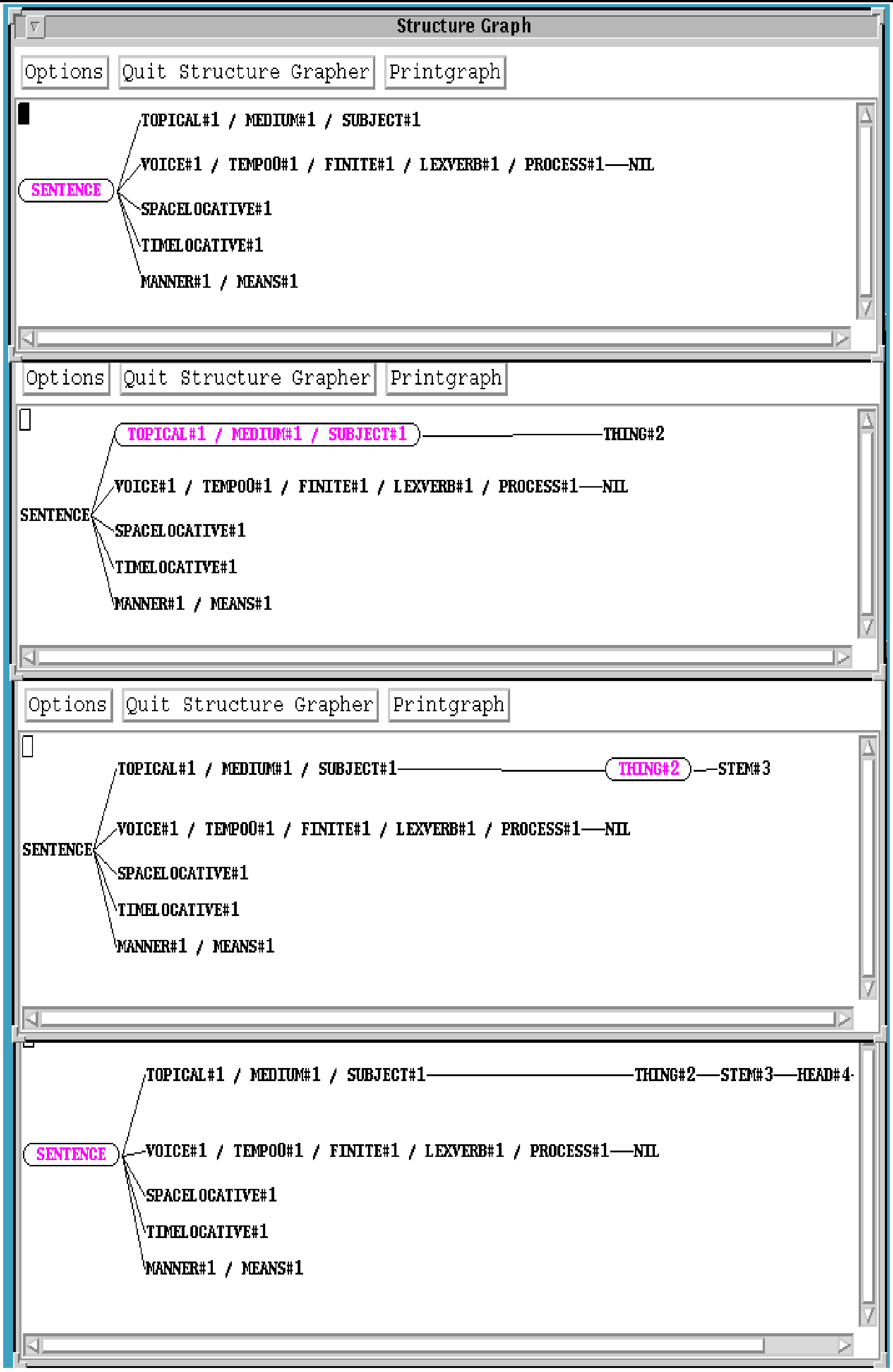


Figure 7.11: Successive structural snapshots during generation indicating 'last' generated node

in Section 6.5. Further details of the realization constraints referred to here are given in Section 12.2.5.

7.9.3.1 Selection expression

Produces a list of the features selected during the traversal of the systemic network that was responsible for the generation of the clicked upon node. The list either appears in the INSPECTOR's *interaction result* pane or in a pop-up window of its own as toggled by the appropriate switch under the ROOT:<Flags> command. Each of the feature names shown is mouse-sensitive and can be clicked upon for further graphing of resources or for listing the definitions of the systems involved, etc.

7.9.3.2 Preselections

Prints in the interaction results window a list of the preselection realization constraints that were imposed upon the clicked upon node by its parent grammatical unit. The features specified as preselections are mouse-sensitive.

7.9.3.3 Orderings

Prints in the interaction results window a list of the ordering realization constraints that were imposed upon the the clicked upon node by its parent grammatical unit.

7.9.3.4 Lexical constraints

Prints in the interaction results window a list of the lexical realization constraints (i.e., classify, inflectify and lexify realization statements) that were imposed upon the the clicked upon node by its parent grammatical unit.

7.9.3.5 Associations

Prints in the interaction results window a list of the function associations made during the generation of the clicked upon node. These associations are displayed as a sequence of lists where the first element in each identifies the grammatical function, the second the semantic unit associated with the grammatical function, the third the set (if any) of lexemes selected for the constituent, and the fourth the single lexeme selected for the constituent's realization. All are mouse sensitive as appropriate.

7.9.3.6 All structural constraints

This option is only intended for difficult to find problems that may have been caused by non-standard implementations of inquiries, or new choosers and inquiries. It pushes onto the global variable `*nodes*` in the Kpml package a pair consisting of:

- a list of the component grammatical functions making up the clicked upon node;
- the complete internal data structure constructed and accessible from that node.

As long as the internal data structure is that created by Penman-inherited code (i.e., for the grammatical structures in KPML), its contents are extremely verbose. All information is present there, albeit in a very unwieldy form. All the standard information can be reached more appropriately and conveniently from the other graph node options described here—this option is therefore intended to be used when non-standard additions to the standard capabilities are being experimented with, and where access to the internal data structures themselves is required.

Note: applications should not build code that depends on the internal form of these data structures. There is no guarantee that it will be preserved across subsequent KPML releases. Interaction with KPML-internal details of generation should only be defined in terms of recognized interface structures, such as those produced as a possible result of the say function, for example (Section 14.1).

7.10 Inspecting the results of generation: Operations on the produced strings or textual structure displays

It should be noted here that a further possibility for inspecting the partial results of generation is offered by direct mouse-clicks on both the generated string and, if it is displayed, the textual version of the final structure produced in the *Development* results pane. This can be a quicker way of finding information than going via the structure graph. It is important to understand, however, that the mouse operations here are operating *not* on the internal data structures used during generation, but on the *example record* that was cumulated during generation. The options are, therefore, slightly different and not limited to the last sentence that was generated. The full possibilities here are given in Section 10.3. A detailed introduction to KPML example records is given in Chapter 10.

When 'generated strings' are on the activated pop-up windows given under the ROOT: <Flags> command, then strings generated are brought up in their own display window. The results shown in this window respect the same display flags as results shown in the *Interaction results* pane on the DEVELOPMENT window. The results are exhibit the same mouse sensitivity as strings shown in the *Development* window with all the normal options for generated strings (cf. Section 10.2.5.1).

The pop-up window provides a convenient way of maintaining several generated results on screen at the same time, as well as supporting diverging fonts (cf. Section 12.2.2.3). When generating in contrastive mode, individual windows are popped up for each language.

7.11 Switching Languages

The command DEVELOPMENT:<Set Language> allows the user to set the 'current language'. This is the language that KPML assumes for all generation and displays of information. The default KPML behaviour is that language switching is an entirely KPML-internal affair. That is, language switching involves no additional reading or re-reading of external linguistic resource definitions and is achieved solely on the basis of the KPML-internal multilingual data structures.

Alternative means of generating in different languages are provided by the multilingual modes as described in the following subsection. Moreover, if a particular example is defined only for a single language, then that it is the language that will be used during generation unless this is explicitly overriden.

Note that any necessary changes that go beyond the systemically expressed resources (e.g., language specific changes to the upper model, etc.) are beyond that supported automatically under this command. Such changes are in any case *to be avoided*: the natural language conditionalization mechanisms provided by KPML should be used instead. See also the comments on this topic in Section 12.1.

Although largely a relic of less multilingually consistent sets of resource definitions, it is possible to cause language switching to have a variety of side-effects. When the internal flag `*patch-loading-on-language-switching*` is set, then switching languages causes language specific files to be loaded for the language being switched into. These files are any inquiry implementations, orderings, punctuation and KPML code patches that are found in the concerned language variety directory (see Section 12.1 for the file names and directory structure). Loading the file `inquiry-implementations.lisp` overwrites all existing inquiry implementations! If present, this file should clearly contain definitions of *all* inquiry implementations needed by the language in question, since existing definitions are either flushed (in overwriting mode) or simply overwritten (in merging mode). Language specific implementations that are to be used *in addition* to the standard implementations rather than instead can be placed in a file `inquiry-increment.lisp`.

For optimal switching between languages it is best if a language variety directory contains *no* language specific code patches or non-uniquely named inquiry implementations. Only then is language switching a completely KPML-internal affair, requiring no loading of files for further information. When resource sets are not mutually compatible and changes to the system by means of file-loading is required, the frequent changing of current language is made unattractive. In such cases, detailed contrastive work is probably better performed by making a number of instantiations of the system, one for each language required.

7.12 Summary of generation process information chains

The tracing and inspection facilities described in this chapter provide a further set of possible information chain transitions over and above those for the linguistic resources summarized in Section 6.6. These provide chains of information involving the *actualization* of the potential represented by the linguistic resources: i.e., the results and decisions made during the generation process itself.

These are summarized in Figure 7.12. The arrows pointing to unboxed linguistic objects printed in italics (inquiries, systems, etc.) mark points of entry to the linguistic potential information chains shown in Figure 6.12. The boxes with a black circle in their top right

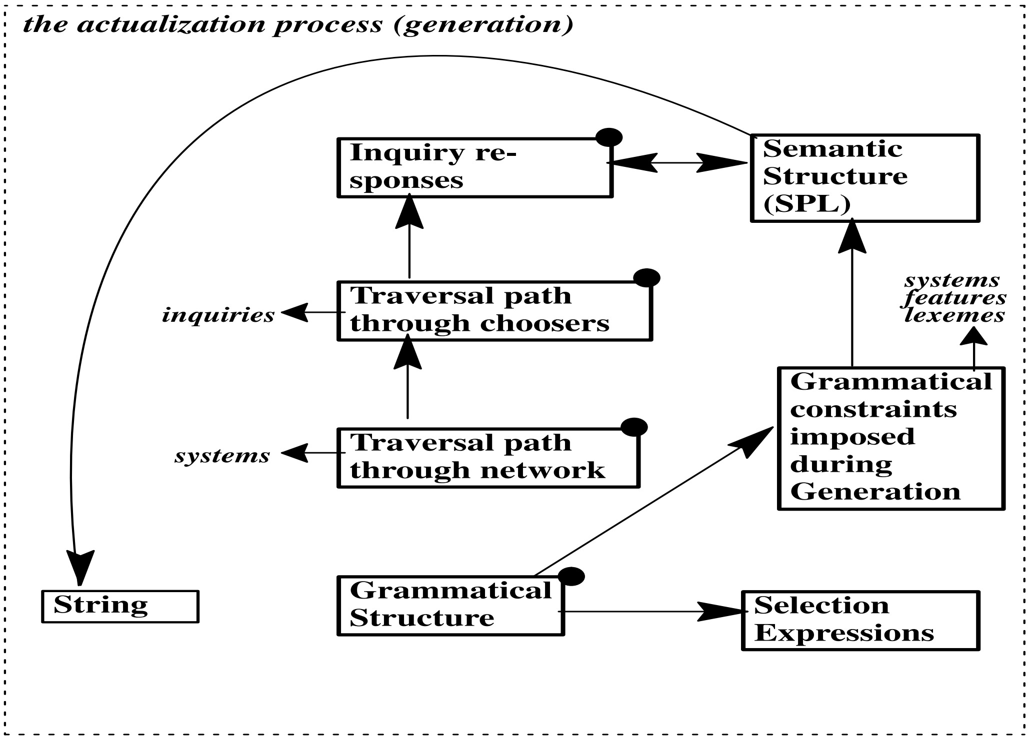


Figure 7.12: Summary of actualization process information chains

hand corners have points of entry provided by explicit KPML commands in addition to possible activation by mouse-clicks.

7.13 How to debug resources: a sketch of a method

The following is one straightforward way of debugging resouces when they do not perform as expected or required. There are other ways, and individuals will probably develop their own preferred styles of working. Situations may also arise which are too complex for the simple strategy outlined here—but as a starting point it would still serve well.

When a sentence or other linguistic unit has been generated and is known not to produce the correct result, the following steps can be performed without any additional tracing activated:

1. Generate (or attempt to generate) the unit desired, starting from the assumed semantics, with either DEVELOPMENT:<Generate Sentence> or from Lisp: `(say '<SPL-spec>)`.

2. When something has been generated (or generation has broken), examine the structure produced with DEVELOPMENT:<Graph Structure>. (For large structures, more selective structure graphing can be used as set out in Section 10.3.)

3. Find an example in the structure of a constituent that did not generate as expected.

4. Click the parent node (if any) and examine the constraints set on the problematic constituent (particularly the preselections and lexical constraints: cf. Section 7.9.3): are these correct and sufficient for the desired behaviour?

5. If not, it is the parent node that is at fault: go back to step (4) this time considering the parent node.

6. If correct, bring up the selection expression of the problematic node (cf. Section 7.9.3).

7. Find the first feature on the selection expression list that deviates from that necessary for desired behaviour. (When more familiar with the resources being used, this can be very quickly established since one knows what the features are for. With more unfamiliar resources, some of the inspection tools (Chapter 6) and examples (Chapter 10) may be usefully applied.)

8. Examine the system where the wrong feature was selected in order to find out why:

   - if there is a chooser, then this can be traced (cf. Section 7.6.1.2), generation can be redone, and the chooser examined in order to find which inquiries produced inappropriate responses—debugging can here move to the interpretation of the semantic input, checking that the given inquiry implementations find the necessary semantic distinctions;
   - if there is no chooser, then the correct feature must be selected by preselection: this may indicate that insufficient constraints were brought to bear from the parent node.

Chapter 8

The 'old-style' KPML interface

The old-style KPML interface provides an interface very similar to that available with the Penman system, with slightly extended graphical and mouse-oriented facilities.[1] This interface consists of a single top level interaction window which combines panes for presenting the most useful information for debugging linguistic resources of the multilingual systemic-functional type. It also includes a main root menu of available operations; many of the main commands for the distinct new-style interface command menus are reachable here via submenus.

The old-style window interface is available for both Allegro and Lucid Common Lisp, CLIM-1 and CLIM-2. It is the only option available if Allegro Common Lisp is not being used, or if CLIM-2 is not present. The old-style window interface is not being actively developed at this time.

The top level interaction windows and their contents are as follows:

<table>
<tr><td>Current System Name</td><td rowspan="2">Operation Menu</td></tr>
<tr><td>System and Feature History</td></tr>
<tr><td>Current Inquiry Name
Current Inquiry Definition</td><td rowspan="4">Interaction Results</td></tr>
<tr><td>Inquiry Response History</td></tr>
<tr><td>Target Sentence</td></tr>
<tr><td>Command Interaction Window</td></tr>
</table>

The *Operation Menu* includes the most common actions that a resource developer will require, but does not exhaust the commands available. Further commands can be typed directly at the *Command Interaction* window and by special keystrokes. Details of all these commands are given in the sections below, organized by desired functionality.

Results of operations performed are displayed in the *Interaction Results* window.

The *Current System Name*, *System and Feature History*, *Current Inquiry Name*, and *Inquiry Response History* windows are present for the display of particular kinds of information con-

[1] Slightly more than was available in KPML 0.8.

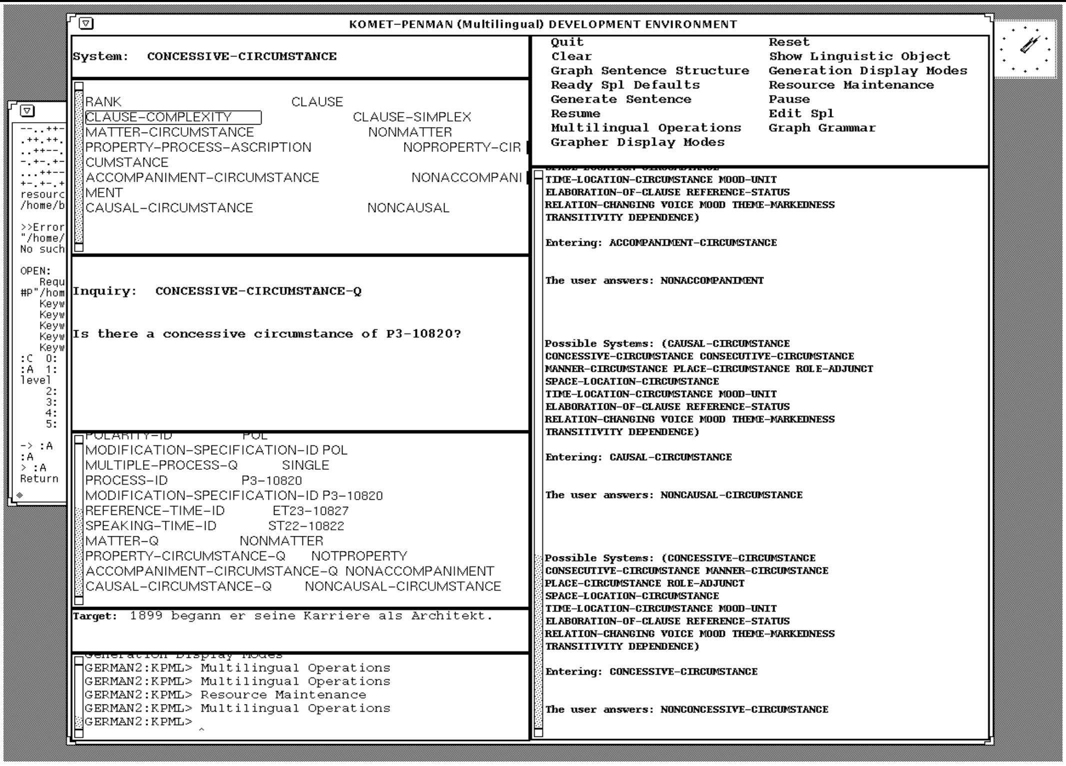


Figure 8.1: Old-style top level interface window

cerning the text generation process and the traversal of the grammar; some of them are 'live', i.e. mouse-sensitive for the ready display of useful information concerning their contents. Moving the mouse around a window will quickly reveal the mouse-sensitive portions.

The *Target Sentence* window shows a target form for a sentence that is being generated: this form is associated with example input structures by the user as a reminder of what the input structures are intended to generate.

Finally, as is generally the case with KPML, it is recommended that the user sets up the screen so that the calling Lisp listener can also be seen in the background while working with KPML (as can be seen on the left of the screendump shown in Figure 8.1).

8.1 Description of the interface 'sub-windows'

An example of the top level interface in the middle of an interaction session is shown in Figure 8.1.

Running from top to bottom on the left-hand side of the screen, the windows are:

- System Name: displays the grammar system that is currently being traversed through. Left-clicking on the system name in this window will display the definition of that system in the Interaction Results window on the right-hand side of the screen. Right-clicking will present a menu of options, one of which is to enter the grammar system network browser that displays the connectivity of the linguistic resource network in graphic form. This possibility is more fully described under the Graph Grammar operation (Section 6.2).

- System Feature History: maintains a scrollable list of the systems that have been traversed through and the choice of feature that was made in each of those systems; clicking on any entry allows the system concerned to be inspected in a similar way to that available in the System window.

- Current Inquiry Name: shows the formal name of the current inquiry that is being put to the environment; the english gloss of this inquiry is shown in the Inquiry question window. Clicking on the inquiry's name in this window causes the definition of the inquiry to be displayed.

- Current Inquiry Definition: as long as the appropriate flag from the Generation Display Modes menu (Section 7.5.2) is set the informal natural language form of the current inquiry appears here.

- Inquiry Response History: is a scrollable window showing the names of all the inquiries that have been put to the environment and the responses that were received. Clicking on any of the entries in this window causes the definition of the selected inquiry to be displayed.

- Target text or Input text: shows the text that the grammar is trying to generate. This is set in the case of examples from a field in the example record data structure. It serves as a reminder to the user of the form that the example will generate. This is the `targetform` field of an example record; the user can set this as a suitable reminder of what the linguistic resources are being used for in order to generate.

- Command Interaction Window: all commands (including those selected by clicking on the operation menu) that the user gives are entered here. Also, any responses that the user needs to supply which are not handled via separate menus are entered here. This is also, therefore, where the prompts for such information appear.

On the right-hand side of the screen there are two windows:

- Operation Menu: the most commonly required operations available to the user. These menu options provide for starting, pausing, and ending the generation process, setting the quantity of information that is given during generation, examining particular aspects of the generation process as it occurs, examining the linguistic resources (paradigmatic specifications) and the results of using those resources (syntagmatic structure), and loading and saving linguistic resources.

  This main menu appears as follows.[2]

[2]Minor differences in the positioning and ordering of the options can occur as the menu is dynamically constructed.

Quit	Reset
Flags	Clear
Show Linguistic Object	Graph Sentence Structure
Generation Display Modes	Ready Spl Defaults
Resource Maintenance	Generate Again
Pause	Resume
Multilingual Operations	Reset Generation Modes
Graph Grammar	Grapher Display Modes

Most of these commands can also be typed in directly at the Interaction window; those with submenus typically allow the submenu commands to be typed at the interaction window also.

The functions and uses of those menu options particularly concerned with controlling the window interface are given in detail below in Section 8.2.

- Interaction Response: this is the window that holds all the general information that may be given during generation because of the flags that are set from the Generation Display Mode menu option and the specific information that the user can ask to be displayed at any time, such as, for example, the displaying of systems, choosers, or inquiries by clicking on the mouse-sensitive window areas described above.

8.2 Basic Old-Style Interface Operations

The interface oriented basic operations offered by the *operation menu* are as follows.

8.2.1 Clear

Immediately (or as soon as the process gets a chance...) clears all windows, including the scrolling windows history.

8.2.2 Flags

Brings up a menu containing a host of flags that control the finer running of the KPML system. This can typically ignored until a more precise idea of the possibilities that KPML offers has been gained. These options are as described in Section 5.4.2.

8.2.3 Pause

Temporarily stops the generation process. While the process is stopped one may use any of the display functions to look at the current state of the generation process. Note that pause only works for generation started in the window interface with commands such as <Resource Maintenance: Operations on Examples: Generate Sentence>, <Generate Again>, etc., and not for generation started elsewhere (for example directly in the Lisp listener via the say function as described in Section 14.1).

8.2.4 Quit

This command exits from the interface window and then destroys that window.

8.2.5 Resume

Continues the generation after a <Pause>.

8.2.6 Reset

Immediately (or as soon as the process gets a chance...) clears all windows, including the scrolling windows history, and forces any existing generation process that has been started to exit.

8.2.7 Show Linguistic Object

Leads to the inspection options—mostly similar to those described for the *Inspector* window in Chapter 6.

8.2.8 Generation Display Modes

Sets the generation modes as described in Section 7.5.1; these modes can then be reset with the <Reset Generation Modes> command.

8.2.9 Resource Maintenance

Leads on to options similar to those available under the *Development* window of the new-style interface (Chapter 7).

8.2.10 Multilingual Operations

Leads on to options similar to those available under the *Root* window of the new-style interface (Chapter 5).

8.2.11 Graph Grammar

Provides similar functionality to the INSPECTOR:<Graph Grammar> command (Section 6.2). The modes for graphical display can be set by the command <Grapher Display Modes>.

8.2.12 Graph Sentence Structure

Displays the grammatical constituency of the last generated sentence or linguistic unit (cf. Section 7.9).

8.2.13 Ready SPL Defaults

Older versions of the system required that default values for inquiries be explicitly set if required (cf. Section 7.4.4). This command activates defaults on demand.

8.2.14 Generate Again

Generates the last example again, as with DEVELOPMENT:<Generate Again>.

8.3 Further type-in commands

The following commands can be typed directly at the interaction window and are not available from the menu. Command completion is provided.

8.3.1 Abort

Commands being typed in at the *interactor window* can be aborted at any time by typing a control-Z.

8.3.2 Environment Directories

Brings up a menu in which the environmental file directories that the KPML system uses for various kinds of information access and display. The directories currently maintained here are:

- Root of resources: the directory where all linguistic resources hang.
- Hardcopy directory: the directory where postscript versions of graphed information are sent.
- Merging results directory: the directory that records the actions taken when resources are being merged during loading rather than overwritten when the most verbose tracing flags are set (see Section 5.7.2.2).
- Example runner results directory: the directory where the results of attempting to generate all loaded examples (see Section 9) are recorded.

8.3.3 Show Path To

This shows the path through the loaded systemic network that is necessary to reach the specified linguistic feature. This may be incomplete if complex entry conditions (i.e., disjunctions) are found on the backward chaining path. See also Section 6.5.3.4.

8.3.4 Evaluate Lisp Expression

This command is given by typing a comma "," as the command name in the Interaction window. The user is then expected to type in a Lisp expression. This is evaluated and the results are shown in the General Information window.

8.4 Various mouse-click triggered commands

In general, a left mouse click over a mouse-sensitive object in the Interaction results window will print a definition or description of the object selected. In addition, however, a right mouse click will offer a menu of further commands which vary depending on the type of object selected. For the display type options, see the descriptions given in Section 6.3; for the options that effect generation (tracing, enabling, etc.), see Chapters 9 and 10.

The options for a grammatical system are to:

- print the description of the system,
- disable the system from use in generation,
- enable the system for use in generation,
- trace the system when it is used in generation,
- stop tracing the system,
- graph the network beginning at the system selected.

The options for a systemic feature (i.e., a term in a systemic network system) are to:

- print the description of the feature (showing the systems which have the feature as an input condition and the system where the feature is defined),
- only show systems having the feature as input,
- only show systems having the feature as output,
- print the path through the systemic network leading to the systemic feature,
- show the list of loaded examples that use the feature.

  Note: this option will only show examples where the selection expressions have already been provided by generation (Chapter 10). In order to save

space, many examples do not include this information. It can be added, of course, by generating the example with the *Update Example Record Fields* **option set (Section 7.5.2).**

The options for an inquiry are to:

- print the inquiry definition,
- print the definition of the inquiry implementation,
- show who can ask the inquiry,
- show who can identify the inquiry,
- pause when the inquiry is used in generation,
- stop pausing when the inquiry is used in generation.

Any options marked as `translator` should probably be avoided under Lucid Common Lisp's CLIM.

Chapter 9

Static Integrity Checks: Resource maintenance

9.1 Background concepts

The resource debugging support tools offered by KPML can be divided into three broad classes:

- static tests during resource loading,
- static tests on whole resource set,
- generation tests.

Static tests examine the resources as defined and attempt to determine inconsistencies, possible mistakes, etc. These tests are described more fully below. Generation tests involve using the linguistic resources for generation: this will typically bring out far more detailed inconsistences or errors than the static tests can.

The warnings issued by the static tests carried out during loading should always be carefully attended to. Inherently inappropriate or incorrect resource definitions can lead to resource sets that are difficult to debug using the generation tests!

There are in addition two classes of messages that the system will give while running static tests or during generation: warnings and cautions. Warnings are issued when an error is known to have occurred in the system. Cautions are issued when a potential error or suspect condition is recognized.

9.1.1 Static tests during resource loading

The most common warning given here is that a grammatical feature is defined for some grammatical system and that feature was already known as being defined for some other grammatical system. This is an untenable situation since a grammatical feature can only be used uniquely. When this occurs, therefore, a warning is given and the previously existing grammatical system is *disabled*. Disabled grammatical systems play no further role for integrity checking or for generation. They are, however, still present in the system and can be reactivated (assuming that the originating error condition no longer applies) with the <:Enable system> command (cf. Section 7.8.1).

The uniqueness condition for grammatical features holds only *within a single language* however. It is, of course, acceptable to have distinct languages which assign a grammatical feature of the same name to distinct grammatical systems. Disablement of a system is therefore always relative to particular languages. It is possible for a system to be disabled for one language but enabled for another.

9.1.2 Static tests on whole resource set

The INSPECTOR:<:Grammar Consistency Tests> command carries out a range of general consistency checks on the resources as loaded.[1]

In addition, prior to generation with any resource set, KPML runs through a standard set of network connectivity checks which can produce various warnings. Note that this is a required step before using the resources for generation and is thus triggered automatically if generation is requested before connectivity has been established. During this phase, a number of start-up tests are run. Problems here are given as warnings as follows:

- Output features of systems which are not reachable from entry features - ideally there should not be any of these as this type of warning indicates a mismatch between some of the systems of the grammar.

- Input features not recognized as the output of any system. There should always be just one of these - the feature *Start*, which is the input for the system RANK. This is the topmost level of the grammar and therefore is not the output of a system. Any other features in this list indicate some error in the grammar.

- Lexical features not called for by any Classify or OutClassify - Note that lexicons may also be intended for use by other systems and so may contain features which are not used by the currently loaded resources; such features are listed in the start up warnings.

- Features demanded by Classify or OutClassify but not present in the lexicon. This warning indicates that there is a mismatch between the lexical features that the grammar expects to be able to call on, and those that actually exist. If such a warning occurs, then either the grammar or the lexicon should be adjusted to eliminate them.

- Words demanded by Lexify but not present in the lexicon. Anything appearing under this warning should be added to the lexicon.

- Features demanded by preselect but not chosen in the grammar. This means that there are systems which are trying to preselect for features that don't exist in the grammar. This should not occur.

[1] With the present release, this test is probably best run only when single language varieties are loaded.

- Chooser for system choosing differently to the system. If a chooser for some system contains a possible choice of a feature that is not one of the output features of its associated system, then this is an error; the chooser or grammatical system probably needs to be fixed.

It is possible to call for more stringent start-up tests by selecting the *Show Cautions* flag (see Appendix A). Then cautions such as the following will be given:

- Too many void features on system. This means that the system outputs named in the warning are ones which neither serve as input for other systems nor have any terminal realization. When resources are being developed, there tend to be quite a few of these, usually marking the starting points where further development is intended.

- Chooser for system choosing differently to the system. If the chooser has been designed to choose only some of the available options in the system to which it is attached, then it will be reported here. Usually this is because the other options in the system lead to undeveloped areas of the grammar or they are always handled by preselection.

Chapter 10

Resource Verification: Example Sets and Test Suites

One of the main tasks that an environment such as KPML has to support is the ongoing verification that the resources defined do what they are supposed to do. That is, in this case, that a correctly formed semantic specification will lead to an appropriate linguistic realization of that specification in the desired languages. The principle means adopted to achieve this is by supporting extensive test suites, or *example sets*, for any resource set released. A test suite is constructed by generating from a wide range of semantic specifications, attempting to cover as many components of the grammar as is possible. Developing such test suites relies heavily on the generation functionality of KPML and the extensive resource debugging aids provided.

10.1 Example sets and test suites

While debugging or maintaining a set of resources it is usual that a standard set of examples be maintained. This is a list of either semantic specifications or records of responses to inquiries for a complete sentence. The form of an example record is described in Section 12.2.9. Example sets are a crucial way of verifying that a resource set is consistent and adequate. Since it is not feasible to check all combinations of defined features when resources become realistic in size, the consequences of changes and extensions can be monitored by verifying that the generation of the example set has not been compromised. When adding new features to the resources, corresponding examples should be added to the example sets. The ideal is to achieve for each language variety an *exercise set* that includes sufficient examples to 'exercise' every feature of the linguistic resource defining the variety.

Example records are created by storing particular kinds of information concerning the sentences that are generated by a linguistic resource. Each time the generator runs, a particular example record is either selected—explicitly by the user from menus of prestored examples—or created—if semantic input is specified directly (by providing an SPL specification for example). This selected/created example record is updated according to the details of the generation process for the linguistic units generated. The example record therefore provides an abbreviated record of the *results* of the generation process.

Some information is always stored to the currently active example record. This minimal information is that used for presenting the final generated string to the user (the string that is printed in the *Development* window or in its own pop-up window following generation); this is the 'mouseable structure' described in detail in Section 14.5. The string display of this mouseable structure can be used for recovering information about the generated linguistic unit without updating any of the stored example records. As long as such a string is shown in the *Development* window, the minimal associated information remains inspectable. This information is sufficient for supporting the commands on generated strings for graphing structure and showing associated semantic specifications, but not for the commands for showing selection expressions. If this further information is sought for example records that are not sufficiently complete, the message:

```
No information maintained for this node.
```

or something similar will be displayed.

Even the minimal 'mouseable structure' information that is always produced is *not* automatically transferred to the information associated with the named example with which generation was started. Simple generation does not, therefore, alter information that has been loaded from example sets. In order that any information be stored to the maintained example records and maintained, the flag DEVELOPMENT:<Generation Modes> 'Update Example Record Fields' (Section 7.5.4.2) must be set.

With this flag set, the basic information plus additional information is added to the current example record *and*, following generation, used to update the maintained example record associated with the selected example name. The full information collected is then:

- the semantic entity that is the principal 'hub' for any traversal of the grammar (i.e., the head semantic term),
- the set of features selected (the selection expression) during traversal of the systemic network for each such hub,
- the complete set of inquiries posed, their actual parameters and their responses,
- the grammatical structure generated as represented in the 'rich mouseable structure' form (Section 14.5).

This provides the necessary information for additional operations such as showing all examples that use some set of systemic features and presenting the selection expressions and semantic information associated with each grammatical constituent generated. It also supports useful on-line example-based documentation and debugging capabilities.

Most of the standardly released linguistic resources include at least one prestored set of full example records. These files are generally quite large, and so are available separately from the resource definitions. Loading these files enables examples to be found for the defined systemic network features, as well as all the grammatical structures to be inspected, *without* having to generate the example first.

Prestored full example sets are created simply by invoking the example runner over the chosen set of examples with (i) the updating flag set, and (ii) the degree of example runner detail

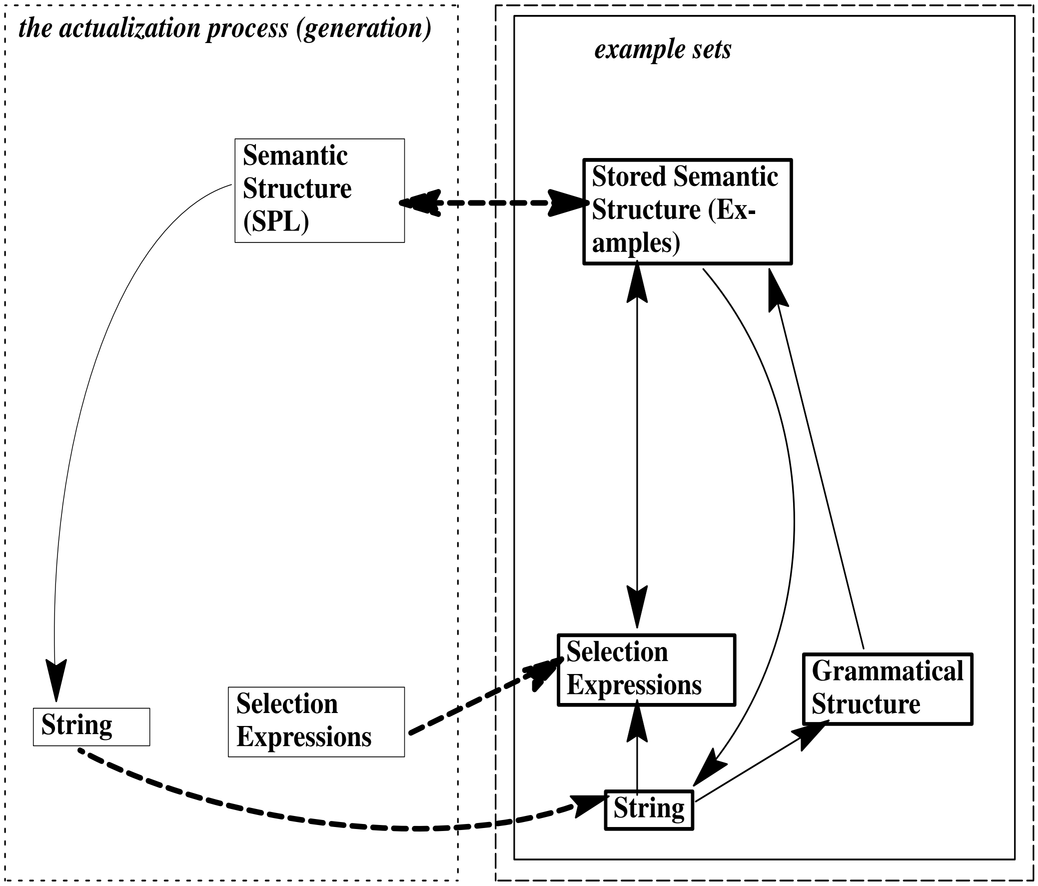


Figure 10.1: The relation of the generation process to example records

(described in Section 10.2.9) set to `:complete`. The user can, therefore, extend these full example sets, or make new such sets, freely at any time.

The operation of the 'Update Example Record Fields' flag is shown graphically in Figure 10.1. This extends the information chain diagram for the generation process of 'actualizing' linguistic resources given in Figure 7.12. The dashed arrows mark the additional information flow that occurs whenever the updating flag is set.

Having a set of full example records loaded also extends the possibilities for following information chains described in Chapter 6 considerably. The complete set of information chain transitions is summarized in Section 10.4.

Note that the example record only records the last generated version of an example in the last language for which generation proceeded. Other information is accessible from the interface as long as the generated strings for any example are being displayed (cf. Section 10.3). In order to save the information for multiple languages, individual save examples should be carried out following generation in the desired languages. The information saved in an example is extensive and it is probably desirable to break this down into as small a packets as possible. Hence single example records do not accumulate any more than the basic results of generation in multiple languages.

10.2 The example operations

This section describes all of the operations that may be performed on examples. This includes both the commands available under the DEVELOPMENT:<Example Operations> command and those commands that are reached by clicking on mouse-sensitive example names in any of the KPML windows.

10.2.1 Load Examples

Whereas the standard behaviour for loading described in Section 5.7 loads by default *all* example set definitions for a language variety, it is also possible to be more selective about which sets of examples are loaded into the KPML environment.

The command <Example Operations: Load Examples> brings up a menu of the example sets available for a selected language variety. Selecting from this menu loads the selected set only. This permits particular example sets to be worked with. The example sets offered in the *Load Examples* menu consist of those files with extension `.spl` found in the appropriate language directory as set out in Section 12.1.

10.2.2 Write Examples

The command <Example Operations: Write Examples> writes out the currently loaded examples to the appropriate directory of the selected language variety. The directory structure of the loaded examples is preserved.

The amount of information in the examples written is limited to:

- the name of the example,
- the target form of the example,
- the logical form of the example.

This enables basic sets of example to be created. If more information is required—for example, full example records—this can be achieved using the `:complete` detail option under the example runner (cf. Section 10.2.9.2),

10.2.3 Clear Examples

The command <Example Operations: Clear Examples> clears *all* loaded examples—i.e., not just the examples for the current language variety.

10.2.4 Generate from example SPL

The command <Example Operations: Generate from example SPL> brings up a menu of examples, selection of one of which initiates generation by appeal to the semantic specification stored in that example (rather than by user interaction or by rote from an example file). Generation is nevertheless normally undertaken in 'implemented' mode (Section 7.4.7), which means that inquiry implementations, where they exist, are used to interrogate the environment (knowledge base, upper model, etc.) rather than having a user intervene in the generation process or having inquiries take their responses directly from the example record. The menu showing available examples can be configured in various ways as described in Section 7.4.2.

As described in Section 7.4.2, generation of examples can also be started by the command DEVELOPMENT:<Generate Sentence>.

In addition, clicking left on any mouseable example name, or selecting the first option in the right-click menu from any mouseable example name, also invokes generation for the clicked upon example.

10.2.5 Graph example structure

The command <Example Operations: Graph example structure> brings up a menu of examples as for <Generate from example SPL> and DEVELOPMENT:<Generate Sentence> (Sections 10.2.4 and 7.4.2 respectively). Selecting an example from this menu brings up a graph representing the generated structure associated with the selected example. The structure graph for an example only exists, if

- the example has already been generated within the current session with KPML, or
- the example has been loaded from a suitably complete example set (cf. Section 10.2.9).

Note that although the graphed structure usually looks very similar to that graphed following generation by the DEVELOPMENT:<Graph Structure> command (Section 7.9), the example structure is, of course, based on the *example record* and not on the internal data structures manipulated during generation. This has two consequences:

1. the inspection possibilities are limited to the information preserved in the example record,
2. the graph reflects the mouseable sentence structure rather than the true grammatical structure—while these are by default in KPML equivalent, they need not be as Section 14.5 describes. Figure 10.2 illustrates this by showing several graphs of the same grammatical structure but with decreasing discrimination of constituents. The top-right graph is the default, with all constituents and terminals mouse sensitive. The bottom-right graph has, in contrast, no terminals mouse sensitive and only those constituents that are either nominal groups or prepositional phrases. Thus only these constituents (nominal groups: the Subject "the news", the Addressee "him", and the Minirange "noon"; prepositional phrases: the Spacelocative "at noon") have their functional labels shown in the graph. The top-left graph distinguishes only the nominal groups.

Finally, the bottom right has no mouseable grammatical units and reflects simply the sequence of strings (including punctuation) representing the generated result.

The inspection possibilities for graphed example structures are reached by clicking left on any constituent shown in the graph. The options are:

- ***Selection Expression***
- ***Semantic Expression***

Of note here is that the identification of selection expressions proceeds on the basis of the *head semantic term* associated with a constituent—in most released resources this is the semantic entity associated with the pseudo-grammatical function `Onus` during each systemic network traversal. Selection expressions will be shown for all traversals of the systemic network that are concerned with the same semantic head. This means that selecting the selection expression for a given constituent can result in several selection expressions being shown.

Example graph structures are displayed in black (rather than, when KPML is running in colour, the blue of the generated structures graphs). Another difference, briefly noted above, is that since the example graphs reflect more the structure of the final *string* rather than the actual grammatical structure, these graphs include any punctuation that the string has been allocated.

10.2.5.1 Display generated string

The command <Display generated string> is to be found on the right-click menu associated with any mouseable example name. Invoking this command brings up in the *Development* window a printed representation of the generated string associated with the clicked-upon example. This representation obeys the general layout flags for generated strings present under the ROOT:<Flags> command. The mouseable constituency of the string is always determined when the example is generated: this cannot subsequently be changed without re-generating.

An important additional functionality of this command is to make the displayed string sensitive to the current set of *collected features* (cf. Section 6.2.3.4). If there are some collected features, then any constituents in the displayed generated string that contain these features in their selection expressions will be highlighted.

A relatively complex example combining this functionality with several of the features offered by KPML is shown in Figure 10.3. Here we see on the top left portion of the figure two overlapping systemic resource graphs (cf. Section 6.2) leading from the RANK system (not shown in partly covered graph) to the MINOR-PROCESS-TYPE system for the language variety Dutch. Here we have focused in on just one feature, 'portion process', removing all others from the graph with the graph pruning facility (cf. Section 6.2.3.5).

Assume that we are interested in seeing how this grammatical feature is in fact realized in sentences—what role does it play?

To begin to get a sense of its use, we collect the feature by right-clicking and selecting the collect feature option (cf. Section 6.2.3.4). We can then ask, by means of the command

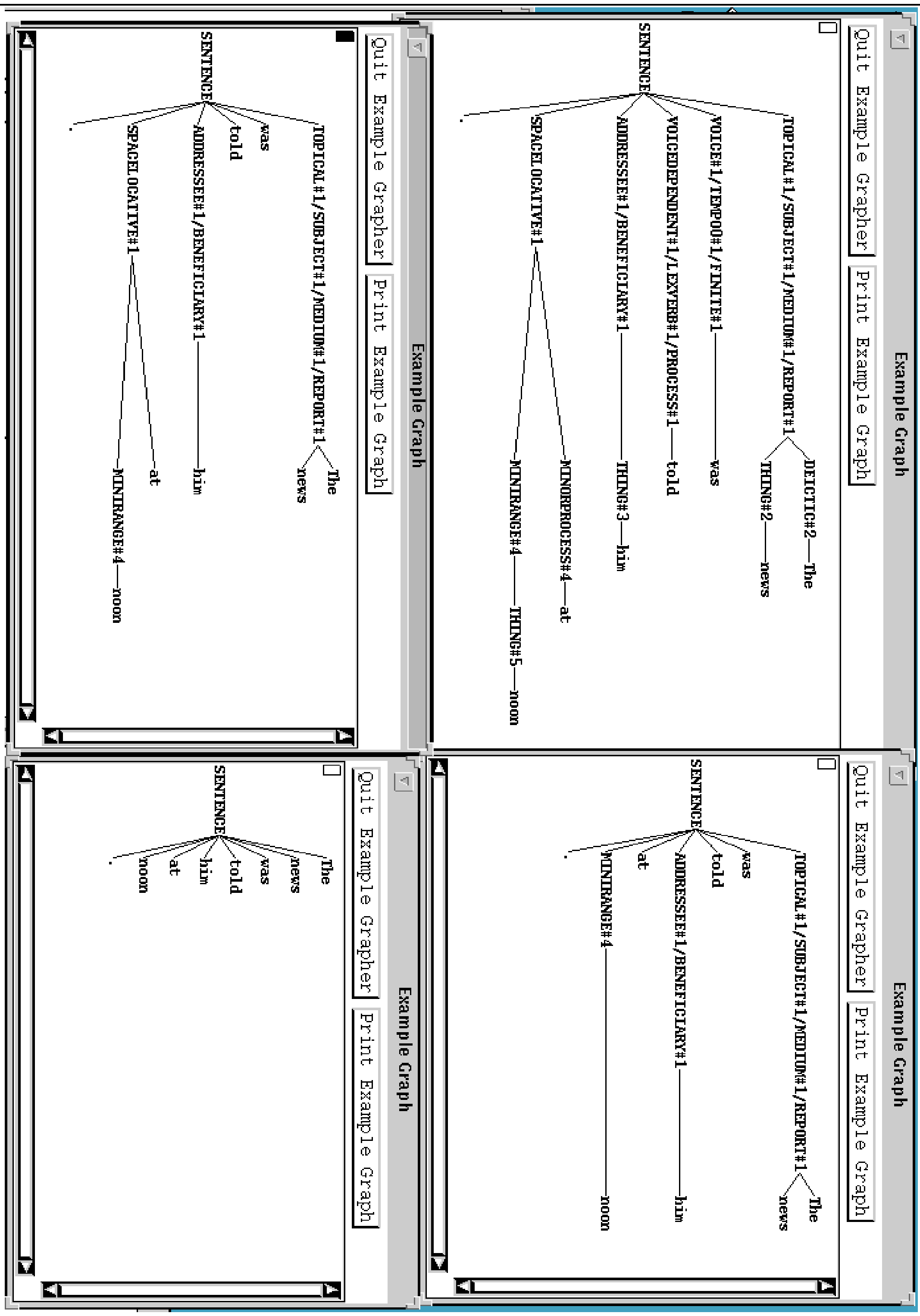


Figure 10.2: Reducing constituent discrimination in example structure graphs

<Show examples with collected features> in the grapher menu, which of the loaded examples use this feature. The result of this operation is shown in the *Inspector* pane bottom left. Right-clicking on any of the example names shown there and selecting the 'display generated string' option produces the corresponding string in the *Development* window on the right of the figure. A selection of the examples have been printed in this way. The constituents of the example sentences using the collected feature are highlighted (in colour on color screens; on monochrome screens they show up as a shade of grey). Thus we can immediately see that the feature 'portion process' occurs in phrases such as the following: "van een eigen huis", "van Mannesmann AG", "van de eeuw", etc.—probably already giving a general impression of the role of this grammatical feature.

10.2.6 Show examples with features

The command DEVELOPMENT:<Example Operations: Show examples with features> is equivalent to the commands available directly from the resource grapher window GRAPH:<Show example with collected features> (Section 6.2.1.3) and from the *Inspector* window INSPECTOR: <Examples using features>.

Any invocation of the command produces in the *Inspector* window a list of example names where the features currently collected (see Section 6.2.3.4) occur in some network traversals responsible for generating the examples is produced. The list is mouse sensitive thus allowing the further mouse-click commands for examples:

- 'Say example' for generating the example (Section 10.2.4),
- 'Rename example' for copying the contents of the example to a new example record with a different name (Section 10.2.7),
- 'Graph structure' for graphing the associated structure (Section 10.2.5), and
- 'Display string' for displaying the associated generated string (Section 10.2.5.1).

10.2.7 Copy examples with new names

The command DEVELOPMENT:<Example Operations: Copy examples with new names> makes a copy of a set of specified examples and prompts for new names for these copies. Subsequently the new names will also appear on the list of examples offered to the user for selection for generation or inspection.

This command may be used for saving a working version of an example, and then changing either the example or the resources in order to be able to compare the effects of the change side-by-side with the situation before the change (since if the original example contains generation process information such as the selection expressions, this information will naturally have been preserved).

Since this feature can be very useful in checking successive alterations to a set of resources, the flag DEVELOPMENT:<Generation Modes> 'Automatically create new examples' provides this as the standard behaviour *whenever* an example is generated. Thus setting this mode and issuing DEVELOPMENT:<Generate Sentence> for the example `Behrens4`, for example,

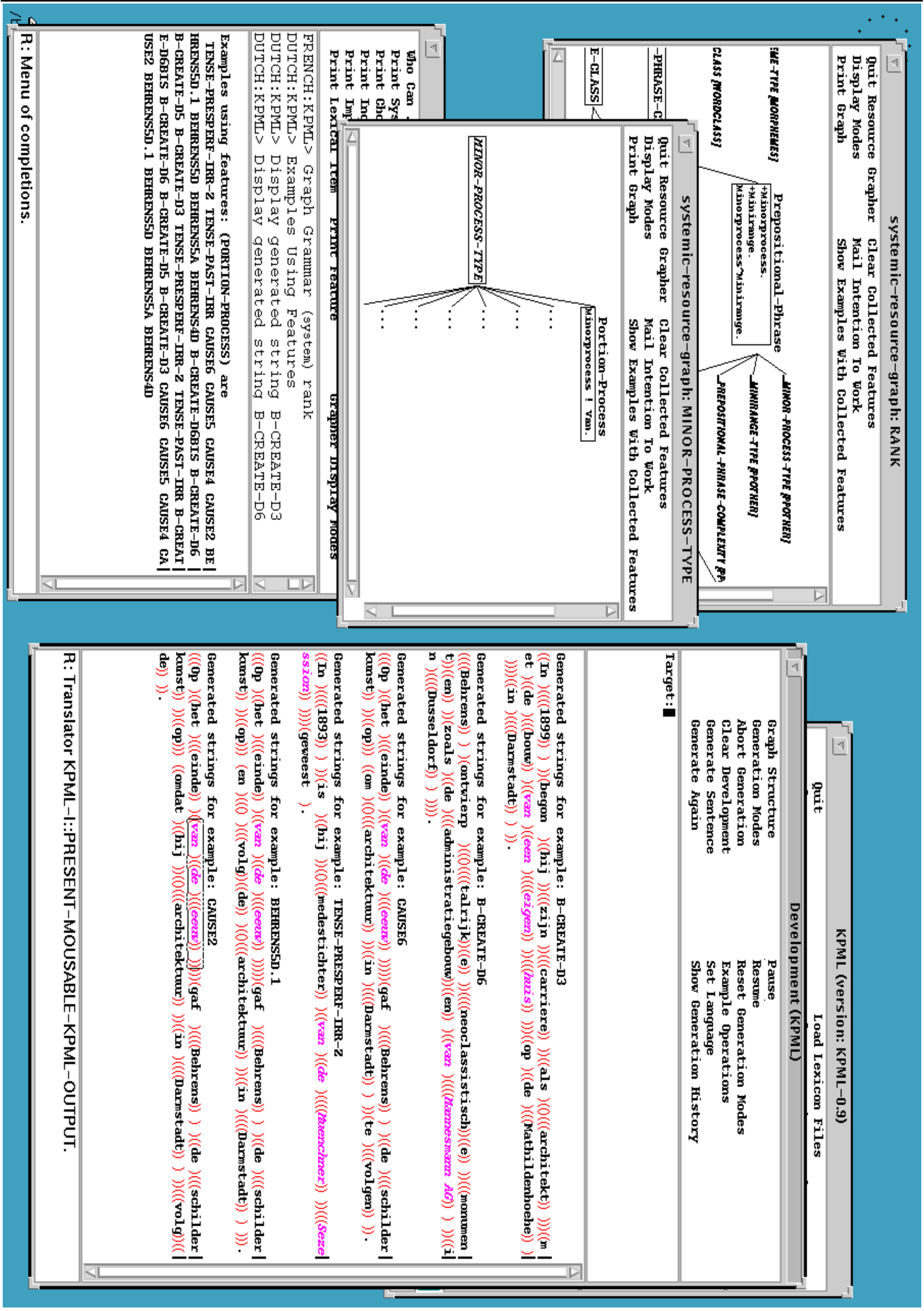


Figure 10.3: Using collected features and example string displays

first causes the example record labelled `Behrens4` to be copied to a new example (named: `Behrens4[hh-mm-ss]`, where the extension denotes the time of creation of the new example), and then initiates generation *on the new example* not on the old. This means that the original example record is preserved untouched and can be inspected and compared with the new.

Issuing a DEVELOPMENT:<Generate Again> command in this mode will have precisely the same effect: i.e., the previous example generated is first copied, and the new example record is then used for generation. Invoking generation by mouse-clicking appropriately on an example name is also effected in the same way.

With this mode in force, no example record is ever changed: all invocations of generation always produce a new example record and work with this. Different versions of examples are therefore maintained simultaneously. Subsequently, versions that are to be kept can be renamed and unwanted versions can be deleted.

10.2.8 Delete some examples

The command <Example Operations: Delete some examples> brings up a menu of examples as for <Generate from example SPL> and DEVELOPMENT:<Generate Sentence> (Sections 10.2.4 and 7.4.2 respectively). Any number of examples may be selected from this menu. The selected examples are then deleted—i.e., removed from the example list. They are then no longer accessible in any way.

10.2.9 Example runner

10.2.9.1 Starting the example runner

An exercise set can be run in its entirety to test the loaded and active linguistic resources. This may be initiated by selecting the command DEVELOPMENT:<Example Operations: Example runner>. Example running is a batch operation: no interaction is expected with the user when generation is proceeding in this mode. Progress during example running is reported in the originating *Lisp listener* from which KPML was started—*not* in one of the KPML windows. Any errors that arise that would necessitate user interaction (such as answering an inquiry, deciding on a feature selection, etc.) are trapped and result in the generation of the effected examples 'failing'.[1] Following example running, any examples that failed are listed.

The results of an example run are typically written to a file. The name of the file created consists of `eg-runner-` and the date and time. The directory of this file can be changed by using the the ROOT: <Environment Directories> command (Section 5.4.1); initially the default directory is `/tmp`.

The example runner can run both over semantic specifications in the form of SPL examples and over records of the inquiry responses obtained. If the linguistic resources loaded are adequate for the examples, this operation should run all the way through without any warnings being issued. Any warnings that do occur appear in the example running file as comments.

[1]This does not, therefore, include Lisp errors. If resources are so misformed that Lisp errors occur, then the example runner enters the Lisp debugger as usual and example running is suspended.

Note: since it is not possible to further interact with KPML during the execution of the example runner, all flags required should be set appropriately beforehand.

10.2.9.2 Levels of detail while example running

Four different levels of detail are provided for the output produced during example running. The level of detail desired may be set by the corresponding flag in the ROOT:<Flags> menu.

The levels differ in the quantity and form of the information written to the example runner file. They can be described briefly thus:

- `:low` – minimal detail (the default): this shows only the example name and the strings generated.
- `:medium` – this shows the logical specifications used and strings generated from these.
- `:high` – this shows all of the information for low and medium detail and the textual structure display of the structure generated. If the 'Update environment record' option is activated (Section 10.1 and Figure 7.3), the selection expressions of the generated examples are also shown.
- `:complete` – this mode causes complete example *definitions* (as described in Section 12.2.9) to be written to the example running file. These definitions contain *all* of the information associated with an example record. This mode can therefore be used to create sets of prestored example sets suitable for supporting on-line documentation and the availability of string, structure, and selection expression information for all examples loaded.

  The `:complete` mode turns off all warnings and forces the values produced by inquiries to be accepted without question. This option should only, therefore, be used when the example records to be written have been debugged sufficiently to serve as a proper example set.

All of the created files can be read as Lisp files for further automatic processing: although only the files under the `:complete` detail mode setting are explicitly intended for this. Indeed, the `:complete` example running files are usually so long that they would normally *only* be used in this way.

Corresponding extracts from an example runner execution for the first three levels of detail are shown in the following subsections. Regular use of the example runner is recommended for ensuring that a set of linguistic resources under development remains consistent as the resources grow.[2]

10.2.9.3 Low detail example running

The following is an extract from the beginning of the example runner file created under the `:low` detail setting. The sentences are for illustration purposes only and do not represent

[2] When more than one generated string is produced for an example, only the first of these appears in the example runner file. This restriction does not apply to the `:complete` detail file.

actual resources.

```
(in-package "KPML")
;;; Summary of a run of the example runner on 20-7-1995 20:44:12
;;;        4 examples selected.
(EX-SET-1
"It  is  raining  cats and dogs. "
)

(EX-SET-11
"Each  system  from  New York  his  and  Smith 's
twentieth-century programmer  created . "
)

(PRIMER-14A
"A  ship  loads  a  truck . "
)

(REUTERS2
"The  difference  has  led to  some  schizophrenic  behaviour . "
)
```

The general form of the low detail output file is therefore:

```
package-info
(example-name1 generated-string1)
(example-name2 generated-string2)
 ...
(example-nameN generated-stringN)
```

10.2.9.4 Medium detail example running

The following is an extract from the beginning of the example runner file created under the `:medium` detail setting. The sentences are as for the `:low` detail example.

```
(in-package "KPML")
;;; Summary of a run of the example runner on 20-7-1995 20:45:41
;;;        4 examples selected.
;;; Generating from logical form / spl:
(A / AMBIENT-PROCESS
     :LEX RAIN
     :TENSE PRESENT-CONTINUOUS
     :ACTEE
     (C / OBJECT
          :LEX CATS-AND-DOGS
          :NUMBER MASS )
     )
```

```
(EX-SET-1
"It  is  raining  cats and dogs. "
)

;;; Generating from logical form / spl:
(M / CREATIVE-MATERIAL-ACTION
     :LEX CREATE
     :TENSE PAST
     :THEME T
     :ACTOR
     (P1 / PERSON
          :LEX PROGRAMMER
          :OWNED-BY
          (
               :AND
               (P2 / MALE
                    :EXPRESS-TYPE NO
                    :IDENTIFIABILITY-Q IDENTIFIABLE
                    :NUMBER SINGULAR )
               (P3 / PERSON
                    :NAME SMITH )
               )

          :IDENTIFIABILITY-Q IDENTIFIABLE
          :TEMPORAL-QUAL-Q NOTEMPORAL
          :TEMPORAL-NONORDERING
          (C / TIME-INTERVAL
               :NAME TWENTIETH-CENTURY )
          )

     :ACTEE
     (T / OBJECT
          :LEX SYSTEM
          :DETERMINER EACH
          :LOCATION-CLASSIFICATION-Q NONLOCATION
          :SOURCE
          (N / SPACE-INTERVAL
               :NAME NEW-YORK )
          )
     )


(EX-SET-11
"Each  system  from  New York  his  and  Smith 's
twentieth-century programmer  created . "
)

;;; Generating from logical form / spl:
(E / LOAD
     :ACTOR (S / SHIP )
     :ACTEE (T / TRUCK ) )


(PRIMER-14A
"A  ship  loads  a  truck . "
)

;;; Generating from logical form / spl:
(LEAD / LEAD-RELATIONAL
```

```
     :LEX LEAD-TO
     :CIRCUMSTANTIAL-ASCRIPTION-Q INCLUDED
     :TENSE PRESENT-PERFECT
     :DOMAIN
     (DIFFER / DIFFER
          :LEX DIFFERENCE
          :DETERMINER THE )

     :RANGE
     (BEHAVE / BEHAVE
          :LEX BEHAVIOUR
          :DETERMINER SOME
          :PROPERTY-ASCRIPTION
          (SCHIZOID / SCHIZOPHRENIC
               :LEX SCHIZOPHRENIC-ADJ )
          )
     )


(REUTERS2
"The  difference  has  led to  some  schizophrenic  behaviour . "
)
```

The general form of the medium detail output file is therefore:

```
package-info
example-logical-form1
  (example-name1 generated-string1)
example-logical-form1
  (example-name2 generated-string2)
      ...
example-logical-formN
  (example-nameN generated-stringN)
```

10.2.9.5 High detail example running

The following is an extract from the beginning of the example runner file created under the `:high` detail setting. Only the first example of those shown for `:low` and `:medium` detail is shown.

```
(in-package "KPML")
;;; Summary of a run of the example runner on 20-7-1995 20:50:54
;;;        4 examples selected.
;;; Generating from logical form / spl:
(A / AMBIENT-PROCESS
     :LEX RAIN
     :TENSE PRESENT-CONTINUOUS
     :ACTEE
     (C / OBJECT
          :LEX CATS-AND-DOGS
          :NUMBER MASS )
```

```
    )


(EX-SET-1
"It  is  raining  cats and dogs. "
(


1: Function structure:
 [SENTENCE]
    [SUBJECT#1]                          = "It "
    [TEMPO1#1/TEMPO0#1/FINITE#1]         = "is "
    [VOICE#1/TEMPO1DEPENDENT#1/LEXVERB#1/PROCESS#1] = "raining "
    [RANGE#1]
        [DEICTIC#2]
        [THING#2]                        = "cats and dogs"

)

((C-15132 NOMINAL-TERM-RESOLUTION OBLIQUE NONSUPERLATIVE NONREPRESENTATION
  NONPARTITIVE NONQUANTIFIED NONORDINATIVE NOMINAL-GROUP-SIMPLEX NONTYPIC
  NO-POST-DEICTIC NOT-PROCESS-QUALIFIED NOT-PORTION-QUALIFIED
  NOT-LOCATION-QUALIFIED NOTBENEFICIARY-QUALIFIED NONATTRIBUEND-QUALIFIED
  NOT-TEMPORAL-QUALIFIED NOT-ACCOMPANIMENT-QUALIFIED NONAGENT-QUALIFIED
  OTHER-NONPERSON NONPERSON NONPROCESSUAL NOT-STATUS-MODIFIED NOT-SIZE-MODIFIED
  NONPROVENANCE-MODIFIED NOT-COLOUR-MODIFIED NOT-AGE-MODIFIED UNCOUNTABLE-NOUN
  LEXICAL-THING PRIMARY-NONCLASSIFICATION NOMINAL NONNUMERIFIED NO-QUANTITY
  NONSELECTIVE-PARTIAL PARTIAL NOMINAL-NONSPECIFIC NONSPECIFIC-INSTANTIATION
  NONSINGULAR NONPLURAL CLASS-NAME NONINTERACTANT
  NONELABORATING-NOMINAL-GROUP-COMPLEX NONEXTENDING-NOMINAL-GROUP-COMPLEX
  NONWH-NOMINAL NOMINAL-GROUP NOMINAL-LIKE-GROUPS GROUPS GROUPS-PHRASES START)
 (A-15115 LEXICAL-VERB-TERM-RESOLUTION DO-NEEDING-VERBS TEMPO0TEMPO1 REAL
  VOICE-LEXVERB RANGED MATERIAL-RANGED ACTIVE-PROCESS UNTAGGED
  MOOD-SUBJECT-EXPANDED SUBJECT-INSERTED IT-SUBJECT METEOROLOGICAL MIDDLE
  MATERIAL NOT-PHASE NO-TEMPORAL-LOCATION NO-TEMPORAL-EXTENT
  NO-SPATIAL-LOCATION NO-SPATIAL-EXTENT NONROLE NONMATTER NONMANNER NONCAUSE
  NONACCOMPANIMENT TRANSITIVITY-UNIT UNMARKED-POSITIVE NO-WH-SUBJECT ASSERTIVE
  NONATTITUDINAL DECLARATIVE NO-TERTIARY PRESENT-SECONDARY SECONDARY PRESENT
  POSITIVE-FINITE POSITIVE TEMPORAL FINITE-INSERTED FINITE-CLAUSE INDICATIVE
  NONINTERNAL-SUBJECT-MATTER INDEPENDENT-CLAUSE-SIMPLEX INDEPENDENT-CLAUSE
  NONCONJUNCTED MOOD-UNIT CLAUSE-SIMPLEX FULL CLAUSE CLAUSES START)
 (C-14972 NOMINAL-TERM-RESOLUTION OBLIQUE NONSUPERLATIVE NONREPRESENTATION
  NONPARTITIVE NONQUANTIFIED NONORDINATIVE NOMINAL-GROUP-SIMPLEX NONTYPIC
  NO-POST-DEICTIC NOT-PROCESS-QUALIFIED NOT-PORTION-QUALIFIED
  NOT-LOCATION-QUALIFIED NOTBENEFICIARY-QUALIFIED NONATTRIBUEND-QUALIFIED
  NOT-TEMPORAL-QUALIFIED NOT-ACCOMPANIMENT-QUALIFIED NONAGENT-QUALIFIED
  OTHER-NONPERSON NONPERSON NONPROCESSUAL NOT-STATUS-MODIFIED NOT-SIZE-MODIFIED
  NONPROVENANCE-MODIFIED NOT-COLOUR-MODIFIED NOT-AGE-MODIFIED UNCOUNTABLE-NOUN
  LEXICAL-THING PRIMARY-NONCLASSIFICATION NOMINAL NONNUMERIFIED NO-QUANTITY
  NONSELECTIVE-PARTIAL PARTIAL NOMINAL-NONSPECIFIC NONSPECIFIC-INSTANTIATION
  NONSINGULAR NONPLURAL CLASS-NAME NONINTERACTANT
  NONELABORATING-NOMINAL-GROUP-COMPLEX NONEXTENDING-NOMINAL-GROUP-COMPLEX
  NONWH-NOMINAL NOMINAL-GROUP NOMINAL-LIKE-GROUPS GROUPS GROUPS-PHRASES START)
 (A-14955 LEXICAL-VERB-TERM-RESOLUTION DO-NEEDING-VERBS TEMPO0TEMPO1 REAL
  VOICE-LEXVERB RANGED MATERIAL-RANGED ACTIVE-PROCESS UNTAGGED
  MOOD-SUBJECT-EXPANDED SUBJECT-INSERTED IT-SUBJECT METEOROLOGICAL MIDDLE
  MATERIAL NOT-PHASE NO-TEMPORAL-LOCATION NO-TEMPORAL-EXTENT
  NO-SPATIAL-LOCATION NO-SPATIAL-EXTENT NONROLE NONMATTER NONMANNER NONCAUSE
  NONACCOMPANIMENT TRANSITIVITY-UNIT UNMARKED-POSITIVE NO-WH-SUBJECT ASSERTIVE
```

```
  NONATTITUDINAL DECLARATIVE NO-TERTIARY PRESENT-SECONDARY SECONDARY PRESENT
  POSITIVE-FINITE POSITIVE TEMPORAL FINITE-INSERTED FINITE-CLAUSE INDICATIVE
  NONINTERNAL-SUBJECT-MATTER INDEPENDENT-CLAUSE-SIMPLEX INDEPENDENT-CLAUSE
  NONCONJUNCTED MOOD-UNIT CLAUSE-SIMPLEX FULL CLAUSE CLAUSES START))
)
```

... etc.

The general form of the high detail output file is therefore:[3]

```
package-info
example-logical-form1
  (example-name1 generated-string1
     ( function-structure-11
       function-structure-12
       ...
       function-structure-1M_1)
     (
       (semantic-head11 selection-expression11)
       (semantic-head12 selection-expression12)
       ...
       (semantic-head1X_1 selection-expression1X_1)))

⋮

example-logical-formN
  (example-nameN generated-stringN
     ( function-structure-N1
       function-structure-N2
       ...
       function-structure-NM_N)
     (
       (semantic-headN1 selection-expressionN1)
       (semantic-headN2 selection-expressionN2)
       ...
       (semantic-headNX_N selection-expressionNX_N)))
```

Note that, since the information here is produced from the associated example records, the amount of detail given for the function structures obeys the specifications for mouseable structures as illustrated in Figure 10.2 and described in Section 14.5.

[3]Note that the function structure information appearing is not ideal if this file is to be read into Lisp for automatic processing since some care is necessary to avoid reader errors.

10.2.10 Features used in examples survey

The command DEVELOPMENT:<Example Operations: Features used in examples survey> displays in the *Development* window a list of all the systemic network features that are selected in the selection expressions to be found in the currently loaded set of example records, and a list of all systemic network features that are *not* selected. This command could be used, for example, to check completeness of an exercise set that is intended to cover all features that a linguistic resource defines.

10.3 Operations on example strings and textual displays

Usually, all of the strings that are generated within the window interface and appear in the *Interaction Results* pane of the *Development* window, as well as their grammatical structure display versions (cf. Section 7.4.2), are mouse sensitive and can be used as the starting points for inspecting various aspects of the generation process.

As noted in Chapter 7 and above, this mouse sensitivity operates *not* via the internal data structures used during generation, but via the stored *example records* that are maintained by the KPML system whenever the flag DEVELOPMENT:<Generation Modes> 'Update environment record fields' is set (Section 10.1). Since not all information is stored, this restricts somewhat the information that can be retrieved (when compared with the options under the DEVELOPMENT:<Graph Structure> (Section 7.9) command for example, where the internal generation data structures are used). It also, however, makes available a more representative selection of possible information, since all loaded examples are always available for inspection and comparison. As illustrated in Section 10.2.5, the 'mouseability'—i.e., which components are mouse sensitive—of the resulting generated strings can be further fine-tuned by the user as set out in Section 14.5.

The following subsections describe the commands that may be invoked directly from the mouse sensitive constituents in a displayed string or in the textually displayed grammatical structure.

10.3.1 Operations on displayed strings

The menu of commands for operations on the strings displayed in the *Development* window are reached by mouse-clicking *right* on a highlighted constituent. Constituency is made more visible by ensuring that the flag ROOT:<Flags> 'Show constituency display in generated strings' is set. Moving the mouse over the string will in any case quickly show the constituents which are mouse-sensitive. The default KPML behaviour when newly installed is that all constituents and terminals are mouse sensitive.

The string-mousing commands are as follows.

10.3.1.1 Show corresponding fundle

This command displays in the *Development* window the full functional label (the 'function bundle') (e.g., `TOPICAL#10/MEDIUM#10/SUBJECT#10`) of the clicked upon constituent. The number following each functional description is the 'traversal cycle number' that is also shown in structure graphs.

10.3.1.2 Graph corresponding constituent and below

This command brings up a example structure graph as described in Section 10.2.5 but *only for the substructure of the clicked upon constituent.* This is, of course, particularly useful for focusing in during debugging or maintenance on some part of a complex sentence.

10.3.1.3 Inspect selection expression

This command shows the selection expressions for all grammatical units sharing the same head semantic term (or 'onus') as the clicked upon constituent. The selection expressions are shown according to the mode set in the ROOT:<Flags> menu.

10.3.1.4 Inspect corresponding semantic term

This command shows in the *Inspector* window the semantic term (typically an SPL expression) that provides the semantics for the clicked upon constituent.

10.3.1.5 Partial re-generation

The right-click menu command <...: Generate up to but not including a constituent with this number> invokes generation for the selected example and suspends the generation process when a traversal cycle is about to be started with a traversal cycle number equal to that of the clicked upon constituent. This provides a speedy way of skipping over generation until a problematic or interesting constituent is reached. When generation is suspended, the DEVELOPMENT:<Generation Modes> menu (Section 7.5.1) is brought up with the flag 'Realize Selectively' automatically set. Any additional flags required can be set at this point. Then, on exiting the generation modes menu and as long as the realize selectively option was not deactivated, the user is asked whether the paused upon constituent is to be realized or not. At this point, further information can be obtained from the structure graph or the inspection options.

If the string clicked upon represents the last example generated, then this command is equivalent to requesting that generation be restarted but should stop just prior to generation of the clicked upon constituent.

10.3.2 Operations on displayed structures

The menu of commands for operations on textually displayed grammatical structures displayed in the *Development* window are reached by mouse-clicking *left* on a highlighted constituent. Grammatical structures are shown following generation when the appropriate flag in the ROOT:<Flags> menu is set. Moving the mouse over the structure will in any case quickly show the constituents which are mouse-sensitive. The default KPML behaviour when newly installed is that all constituents and terminals are mouse sensitive.

These commands form a subset of those for string-mousing. Since the structure display already shows the functional label of a constituent, this option is not present. The structure-mousing commands are therefore as follows.

10.3.2.1 Graph this constituent and below

This command brings up a example structure graph as described in Section 10.2.5 but *only for the substructure of the clicked upon constituent.* This is, of course, particularly useful for focusing in during debugging or maintenance on some part of a complex sentence.

10.3.2.2 Show selection expression

This command shows the selection expressions for all grammatical units sharing the same head semantic term (or 'onus') as the clicked upon constituent. The selection expressions are shown according to the mode set in the ROOT:<Flags> menu.

10.3.2.3 Show corresponding semantic term

This command shows in the *Inspector* window the semantic term (typically an SPL expression) that provides the semantics for the clicked upon constituent.

10.3.2.4 Generate again up to but not including this constituent

This command performs the same operation as the equivalent command for string-mousing (Section 10.3.1.5): that is, generation is restarted and is suspended when a constituent with a traversal cycle number equal to the clicked upon constituent is reached.

10.4 Full summary of linguistic resource information chains

We can now extend the view of information chains given in Chapter 6 to include the possibilities offered by examples sets discussed in this chapter. Figure 10.4 extends the diagram of Figure 6.12 accordingly.

As Figure 10.4 shows, there are two distinct kinds of linguistic object which are maintained by KPML: objects that represent the linguistic *potential*—i.e., the linguistic resource definitions

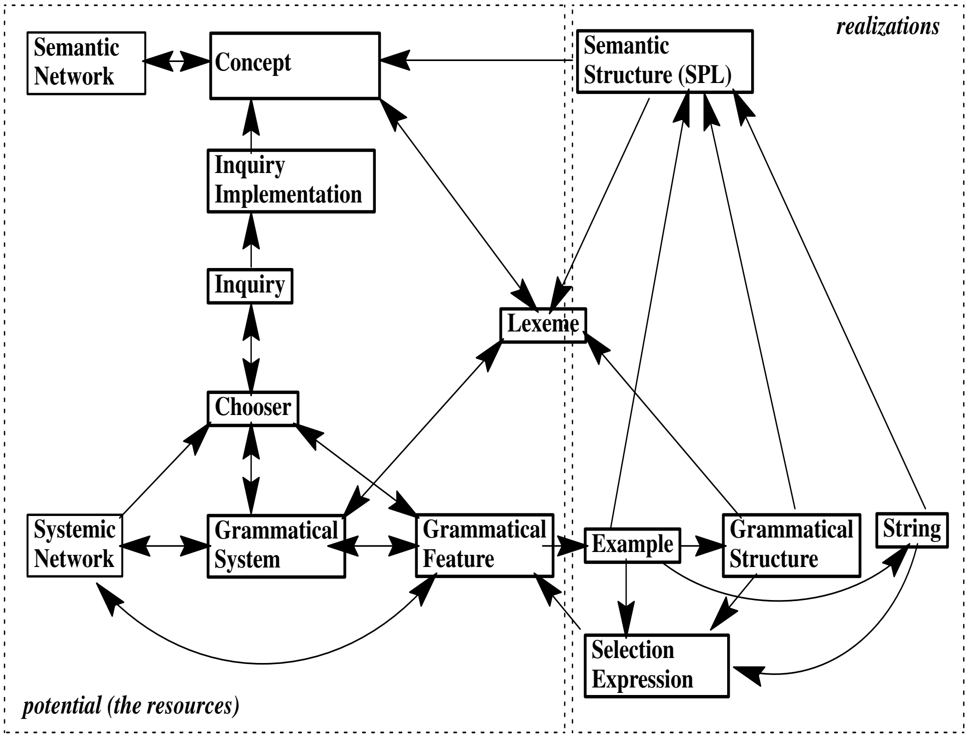


Figure 10.4: Information chain possibilities: potential and realizations

themselves, and objects that represent the result of using that potential—i.e., the *realizations*, or linguistic structures, that are produced (grammatical structures) or consumed (semantic structures) during generation. The possibilities for inspecting resources were described in Chapter 6; the information concerning realizations extends these possibilities considerably. Most information concerning realizations is stored as part of example sets: as emphasized above, information here is only available if example sets have been created during the generation or if pre-loaded (cf. Section 10.1).

Once stored or loaded, it is possible from any example to retrieve its associated grammatical structures, selection expressions (i.e., traversals through the systemic network), generated strings, and original semantic specification. These then can form the starting points for further resource exploration as Figure 10.4 indicates.

Chapter 11

Maintenance: Resource Patching

11.1 Introduction

When working with KPML for the construction and development of linguistic resources for generation, it is usually the case that sets of resources will be successively modified and tested. To support this process, KPML provides for linguistic resource *patches*. This facility allows changes to be isolated from a stable background set of resource definitions. Once the changes have been sufficiently tested, it is then possible to incorporate them in the main body of definitions.

Since the use of the patching facility has several repercussions for the behaviour of the system, the default situation is that patch usage is *not* activated. The use of patching and these repercussions is described in the following subsections: first the consequences for loading linguistic resources are described, then the consequences for saving linguistic resources, and finally some general consequences of working with the patching facility are listed. When the patching facility is not activated, loading and saving behavior is as defined in the sections above and any patches specified in the linguistic resources are *not* loaded. This is the default system behavior.

In order to activate the patching facility, one simply needs to add the pseudo linguistic 'object' `resource-patches` to the focused linguistic object list. This is done with the normal KPML command <Focusing Operations> in the root window (Section 5.6).

For the present, the selective patching facility is limited to definitions of systems, choosers, and inquiries since these are the objects that primarily define linguistic resources. Versioning of examples is, however, also provided.

11.2 Patching and loading linguistic resources

11.2.1 Loading patching behavior

When `:resource-patches` is on the list of focused linguistic objects, loading linguistic resources with the command <Load linguistic resources> will *in addition* to the behaviour described for that command in Section 5.7 also load any patches defined for the linguistic resource being loaded. Such patches must be placed in subdirectories of the main directory for the language variety being patched and have names ending with the string `Patches`. The internal structure of these patch subdirectories is an exact mirror of the resource directory itself.

If there are several Patch directories available, the system will ask the user which is to be loaded. If there is only one, this will be loaded without user intervention.

For example, if a set of resources named '`french`' is to be patched with respect to systems and choosers of the region MOOD, then the directory structure should be as follows. (See Chapter 12 for the general directory structure.)

```
...Root/FRENCH/Grammar/...
               Lexicons/...
               Examples/...
               ...
               Patches/Grammar/MOOD.systems
                               MOOD.choosers
```

All components of the main directory (i.e., the directory `FRENCH` in the current example) may be patched in this way. The patches should have the same multilinguality properties—either monolingual or multilingual—as those of the definitions being patched.

N.B. all resource patch directory *must have* a subdirectory `Grammar` in order to be recognized appropriately as conforming to the expected KPML resource directory structure. (This is important if, for example, one is for some reason using a patch directory for some other purpose, such as alternative lexicon items or examples.)

As is usually the case, the set of object types to be loaded is defined by the list of objects on the focused linguistic object list (see Section 5.6). Thus, if systems only are focused and patching is activated, issuing a <Load linguistic resources> command will *only* load systems: first the main definitions and then any patches concerned with systems.

If no linguistic objects are specified as focused apart from `resource-patches`, then *all* objects will be loaded in the normal fashion, followed by patches.

Region focusing (Section 5.6.3) may be used to further restrict patch loading if required.

11.2.2 Achieving 'patch' status

Once a set of linguistic resources has been loaded then, as long as resource patching is activated, all system, chooser and inquiry definitions that are loaded—*regardless of whether via an explicit load instruction from Lisp, an evaluation in an Emacs-buffer, etc.*—are marked as *patches* with respect to the original language definitions for their corresponding language varieties. This means that it is possible to make arbitrary changes to a set of resources, and then (see next section) to save these changes without affecting the original language definitions. The only definitions that are immune to this are main linguistic resource definitions (i.e., those not in Patch-directories) that are loaded with the KPML <Load linguistic resources> command. Any patches loaded in this way remain, of course, marked as patches.

One way of creating the patches for French MOOD referred to above is then as follows:

1. load the original French resources (which would not yet have had any patches defined),
2. ensure that the patching facility is activated,
3. edit the required definitions of the MOOD systems and choosers,
4. evaluate/load the changed definitions,
5. save the French resources.

This would create the two files that appear in the Patches directory above and the necessary additional directory structure without changing any of the original definitions.

Care should be exercised when loading/evaluating definitions in order that the desired loading behavior is enforced. For example, unless merging is activated (Section 5.7.2.2) any definition loaded will replace *all* definitions of the same named object for other languages. If this is not required, then merging mode must be explicitly selected *and* the language restriction for the object to be loaded must be set as appropriate. For example, if resources for English, German and French are loaded and it is only required to patch the definition of the chooser ADVERBIAL-TYPE-CHOOSER for German, leaving definitions (if they exist) for English and French untouched, then a chooser definition beginning:

```
(chooser :name (:german ADVERBIAL-TYPE-CHOOSER)
         :definition ((... ))
         )
```

should be evaluated with merging (i.e., not overwriting) mode set. If the `:german` restriction is not present, then the definition will be taken as holding for all known languages; if merging mode is not present, then overwriting mode will force all other objects of the same name to be deleted when this one is loaded.

If the user knows that patches are going to be made for a single language, then it is possible to set up a context in which all definitions will automaticallly be restricted to a given language without needing to explicitly add a language restriction. This is enforced by the command ROOT:<Set Default Language>.[1] It is then possible to evaluate definitions without the explicit language restriction and still to obtain the behavior described above where only the definitions of a single language are patched.

This is clearly more convenient if, for example, a resource definition file has been loaded into an Emacs buffer, and a definition has been edited and then evaluated. Typically definitions in a file do not have individual language conditionalizations, and would therefore, without the <Set Default Language> command, be interpreted incorrectly when evaluated.

[1]This is equivalent to issuing an explicit `in-language` command at the Lisp listener (cf. Section 12.2.1). The effects of the `in-language` command can be overridden by a subsequent `in-language` or by calling the function `(clear-region-and-language-defaults)`.

11.2.3 Automatic loading of selected patches

It is possible to use the KPML Lisp functions for loading linguistic resources in order to load certain prespecified sets of patches. This is intended particularly for ease of setting up preloaded resource sets and combinations, for example as might be specified in `site-specifics.lisp` or `kpml-user-specifics.lisp` files for some installation (see 3.5).

This is done by calling the Lisp function `load-linguistic-resources` (Section 14.4) with the additioanl keyword `:patches`. The value of the keyword should be a list of strings giving the names of the patches to be loaded. The patches will be loaded in the order given as they are found while navigating the resource directory. No error is indicated if a named patch is not found. If more than one set of patches is to be loaded at a given level of versioning, then a list of names replaces the single name as element.

As an example, the following function call:

```
(load-linguistic-resources :german
  :patches '("GERMAN-P1" ("GERMAN-P1-Pa"  "GERMAN-P1-Pb")))
```

loads the linguistic resources for language variety German, which is expected to include at least a patch called `GERMAN-P1` at top-level and two parallel patches (`GERMAN-P1-Pa` and `GERMAN-P1-Pb`) within this. These patches are then also loaded and any other possibilities present are ignored.

Note that the search for applicable patches proceeds breadth first and so the embedding of the `:patches` list is not equivalent to the directory structure. If a third element were added to the above `:patches` list, for example, this would be a candidate patch for the second-level patch `GERMAN-P1-Pa`. The significant order is the order in which patches are sought, not where they are embedded.

11.3 Patching and saving linguistic resources

When `:resource-patches` is present on the list of focused linguistic objects, saving behavior initiated by the command ROOT:<Store linguistic resources> is affected as follows. First it is assumed that the user is working using patches rather than with any direct alterations to source linguistic resource definition files, and so only linguistic objects marked as belonging to *patches* are to be saved. This will then be noted explicitly in the save dialog box that is brought up. If this is not intended, then it is possible at this point to override this. As with loading, the types of linguistic object saved can be further restricted by setting the list of focused objects; if only `resource-patches` is set, however, then all patched systems, choosers, and inquiries are saved in an appropriate patch subdirectory. Whenever `resource-patches` is set, *no* changes are ever made to any non-patch linguistic resource definitions.

Whenever patches are saved, new versions of the default ordering and punctuation files are also written out within the patch directory (unless, of course, there is an object focusing restriction excluding them).

A save linguistic resources command that involves writing patches will create a patch directory of the form `yyddmm-hhmmss-Patches` indicating the time of creation.

If a save linguistic resources is used to create a new language variety, then this new resource will be created with any patches present *already folded into the main definitions*. If it is required to create a new set of resources for a language inheriting both the definitions *and* the patch structure from some other language, then the patches need to be saved explicitly.

If a save linguistic resources is used when region focusing is present, then only those regions focused will be saved as patches.

Note that the patch saving facility is generous in the directory structures that it creates. The patches subdirectory will be a full mirror of the originating resource directory even if there are no patches present at that time to fill it. That is, even if there are no files present containing patch-specific lexicon entries, there will still be a `Lexicon` subdirectory created automatically within the patches directory. The user can delete these if required; they are not crucial to the operation of the patching facility.

11.4 Some further consequences of using the patching facility

When the patching facility has been activated, systems, choosers and inquiries may have one of two statuses: either patched or non-patched. It may sometimes be useful to know whether a linguistic object that is being examined during resource development and testing belongs to the original resource definitions or to a patch. To aid this, resources that are defined in patches are displayed in *italics* when inspected in the KPML inspection window. This allows them to be readily identified as patches. Note that once an object has been patched, it is *not* possible to examine the pre-patch version.

It is, of course, possible that a linguistic object of a given name is only patched for some subset of the languages for which it is defined. For example, the screendump of Figure 11.1 shows various views on the system (which were produced by printing the system named COMPARATIVE-PROCESS-TYPE; these views were produced by giving the command in the *interactor* pane in *contrastive* display mode: cf. Section 6.3.3.2. The resulting display shows that this system has only been patched in its German version—the other variants are as given in the main source definitions and so are not marked as patched.

11.5 Modifying linguistic resources

KPML provides direct interaction with GNU Emacs (or GNU Mule: see Section 12.2.2.3) to support the editing or modification of loaded linguistic resources. This is *only* supported when KPML is started as a subprocess of Emacs/Mule.[2] All the basic linguistic objects (systems, choosers, inquiries, lexical items, and examples) presented in a KPML window have an option <Edit...> in their right-click mouse menus. Selecting this option brings up an editor buffer in the originating Emacs/Mule containing just the definition of the clicked upon linguistic object. The presentation form of the linguistic object as it appears in the editor buffer is controlled by the multilingual flags as described and illustrated in Section 6.3.3. The object brought up in the editor buffer may then be freely edited.

[2]Note: the mode of interaction provided in the Penman interface whereby SPL specifications could be edited from a stand-alone Penman process by starting a new editor-process for each edit is *not* supported.

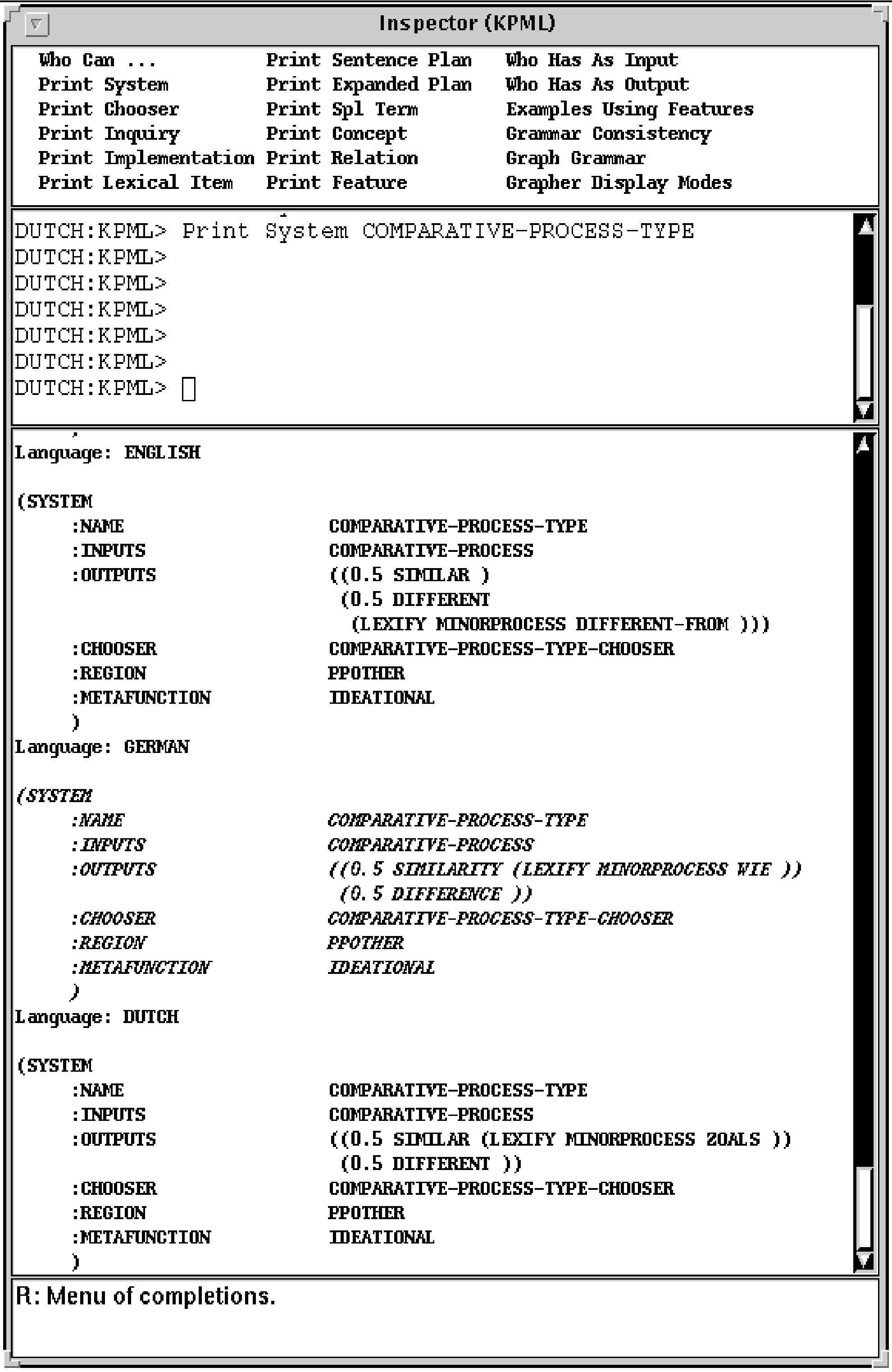


Figure 11.1: Selective patching according to language

The originating KPML window waits for control to return from Emacs/Mule. Return may be made in two ways:

- the changes made in the editor buffer may be *accepted* by giving an Emacs command: cntrl-C cntrl-C. The modified definition is then made part of the currently loaded resource definitions. If the patch mode is activated (which it probably should be when editing resources in this fashion), then the modified linguistic object is appropriately marked as a patch.

- the changes (if any) made in the editor buffer may be *discarded* by giving an Emacs command: cntrl-C cntrl-Z. Control is returned to KPML but there is no effect on loaded resources.

Note that the usual considerations with evaluating linguistic resource definitions apply: if these definitions do not themselves explicitly contain appropriate language conditionalizations, then such conditionalization should be indicated with the ROOT: <Set Default Language> command (cf. Section 11.2).

There are two exceptions to the general editing behaviour described here.

The first exception is for inquiry implementations: it is also possible to issue <Edit Inquiry Implementation> commands. Such a command loads the appropriate Lisp file containing the definition of the clicked upon inquiry implementation (typically a file `inquiry-implementations.lisp` or `inquiry-increment.lisp`: cf. Section 12.1) and positions the editor cursor at the required Lisp definition. If no such inquiry implementation is known to the Lisp process, then a new editor buffer is started with a skeleton definition of an appropriate inquiry implementation in place for editing. The user should write this definition to a file (the Emacs buffer proposes a default consisting of the date of creation and the inquiry name) and then evaluate as normal (EMACS: <cntrl-C cntrl-S> under the Allegro Emacs protocol).

The second exception is for *SPL terms*: that is for elements of SPL expressions. Each full SPL term—for example, that might be produced when the semantics associated with some grammatical constituent is inspected or when an example is inspected—may also be individually editing. The actions on completion are:

- EMACS: <cntrl-C cntrl-Z> discard edits (as above),

- EMACS: <cntrl-C cntrl-C> exit buffer and *begin generating the semantic fragment that has been edited.* It is, of course, very easy to create non-sensical or incomplete SPL expressions in this way that will run into difficulties when generating. It is also possible, however, to quickly create specific test cases for debugging purposes and to focus in on problematic semantic areas. Note that selecting any SPL fragment and regenerating with the same set of preselections holding that were selected when the fragment was generated in the context of a larger SPL should, given the same grammar, etc., always produce the same result for the constituent concerned.[3]

[3]Preselections can be set externally to the grammar for use in debugging by means of the supplement preselections flag (cf. Appendix A.1).

Editing commands can also be given directly from the *Inspector* window.

If KPML is started from Emacs, additional resource definitions can also be straightforwardly evaluated in any other Emacs buffer, but it then remains the task of the user to find the appropriate files for editing.

There is never any automatic updating of the originating resource files—this remains the responsibility of the user. If the patching facility is activated, then it is possible, as described above, to write out just those changes that have been made during a session to a patch directory. If patching has not been activated, then writing out resources following a session where modifications have taken place will create a new resource set incorporating the changes made. **Note: care must be taken that this does not prematurely destroy the existing resources!**

The additional steps necessary for installing the Emacs/Mule interface are described in Section 3.2.

11.6 Example record versioning

As described in Section 10.2.7, the flag DEVELOPMENT:<Generation Modes> 'Automatically create new examples' causes each new generation request to create a new version of the specified example record. These distinct versions can either be deleted when no longer required, or saved in the normal way. This facility therefore provides a basic versioning capability for example sets.

11.7 Acquiring lexical items

The normal KPML generation behaviour when an unknown lexical item is requested follows that of the Penman system. That is, a temporary lexical item whose spelling is the upper case variant of the associated concept name is inserted.

Thus, with the following SPL input,

```
(s / nondirected-action :lex skip
   :actor (p / person :name John))
```

assuming that the concepts `nondirected-action` and `person` are defined (which they are in the standardly released upper model), and that the proper name `John` is also defined (which it is not usually), but the lexical item `skip` is *not* defined, then the following strings would be generated for English: either

"John SKIP"

without morphology and

"John SKIPs"

with morphology. The capitalization is the indication that a required lexical item has not been found.

With KPML it is possible to activate an automatic lexical acquisition mode in which all required lexical items that are not defined are created on the fly with a default set of lexical and morphological features appropriate for the grammatical context in which they appear in their sentences of use. This mode is activated by the flag ROOT:<Flags> 'Acquire Lexical Items'. When set, the above SPL input would not only produce "John skips" but also leave a new lexeme defined for English called `skip` (i.e., the form given in the SPL specification).

This mode is most useful when a set of examples containing unknown lexical items is run through (by using the example runner, for example). The lexical items newly acquired can then be written to a file of lexeme definitions by means of the function `make-new-lexical-items-file`. This allows new definitions to be straightforwardly added to the linguistic resources for the concerned language variety; naturally it might then be necessary to provide idiosyncratic or non-default morphological information for these new lexemes.

This function is used from a Lisp listener and has the details:

`make-new-lexical-items-file` ***pathname*** [*function*]

Chapter 12

Resource Organization and Definition Formats

The KPML system assumes (and creates when the resource manipulation operations are used) a particular organization of linguistic resources. Those resources are in turn represented in an extended form of that defined by the Penman system. In general, KPML can interpret Penman-style resources, although the reverse does not hold. This section describes in detail the KPML resource organization and definition format.

12.1 Directory structure and contents

KPML maintains a global variable (`*root-of-resources*` in the `user` and `kpml` packages) which defines one directory to be the *root of linguistic resources*. This variable is normally set up during system configuration but can also be set from the window interface by using the <Environment Directories> command (see Section 5.4.1). Each language variety or multilingual resource set for which separate resource definitions are required then occupies a subdirectory to this root directory. A typical initial form of the directory on initialization of the system would be:

```
                     |-- GENERAL
                     |
                     |-- ENGLISH
                     |
*root-of-resources*----- GERMAN
                     |
                     |-- DUTCH
                     |
                     |-- ML-BASELINE
```

The general file organization for linguistic resources maintained by KPML within any language variety directory is illustrated in the following maximal example. The directory structure for multilingual resources (i.e., a directory containing the combined definitions of several languages, such as `ml-baseline`) is identical to this. Not all of the files need to be present in any given language variety definition.

```
<language-variety>/properties.lisp
                   language-range.lisp
                   Grammar/<regions>.systems
                           <regions>.choosers
                           <regions>.inquiries
                           punctuation.gram
                           ordering-constraints.gram

                   Lexicons/<lexicon-names>.lexicon
                   Examples/<example-set-names>.spl
                   xxx-Patches/...

                   inquiry-implementations.lisp
                   inquiry-increment.lisp

                   basic-spl-macros.lisp
                   basic-spl-defaults.lisp

                   code-patches.lisp
```

For directories containing definitions of several language varieties there may be several sets of inquiry implementations. For such cases it is also possible to use a subdirectory `Inquiry-implementations` parallel to `Grammar`, etc. where the inquiry implementation files can be kept. All files with extension `.lisp` found in the inquiry implementations subdirectory will be loaded.

Each such directory contains either:

- the complete monolingual definition of the grammar and semantics for the language indicated by its name, or
- the complete multilingual definition of a set of languages where the name is a label for the resource set.

Such definitions consist of several distinct kinds of information. The linguistic resources proper are held in the subdirectory `Grammar` in files with extensions `.systems`, `.choosers`, and `.inquiries`. One distinction between systemic-functional resources as they are generally maintained and supported in KPML and earlier versions of, for example, Penman, is that the resources are divided into *functional regions* (see Section 2). Although always present in the Nigel grammar of English, this information was not previously used for maintenance and modification. Now, all of the multilingual development support tools and the graphical displays operate in terms of regions. Thus, each linguistic resource file normally corresponds to the resources of a particular 'functional region'. This is not enforced in any way, but files created automatically by the Save Linguistic Resources command (Section 5.9.1) will follow this principle.

Note that any standard KPML resource definitions released were in fact created in precisely this way. The <Store Linguistic Resources> command was given successively for each of the languages available, the individual monolingual definitions often being drawn from a pooled multilingual resource.

The form of entries in each type of the linguistic resource files is given in the next subsection.

Note: in the case that resource files are not found, check that the path definition given in this file is correct for the current directory configuration that is being used.

In addition to the linguistic resources proper, loading linguistic resources from one of the above directories with the ROOT:<Load Linguistic Resources> command (Section 5.7) will also load the following files or file types:

- a file `properties.lisp`: if such a file exists, it is assumed to hold general declarations applying to the language variety or varieties as a whole (see Section 12.2.2).
- a file `language-range.lisp`: if such a file exists, it is assumed to hold a declaration of the range of language varieties dealt with by its containing resource directory (see Section 12.2.3).
- a file `inquiry-implementations.lisp`: if such a file exists, it is assumed to hold the Lisp code that implements the inquiries defined by the linguistic resources (in the files with extension `.inquiries`). **Note: unless the *merging* mode is in force (Section 5.7.2.2), any implementations currently loaded will be lost or replaced during this operation!** If the resources use standard inquiry implementations, then no such file should appear in the language-specific directory.
- a file `inquiry-increment.lisp` that may contain additional inquiry implementations over and above the standard ones. Placing inquiries here avoids the default removal of existing inquiry implementations once a file `inquiry-implementations.lisp` has been found. Any inquiry implementations placed in the inquiry implement should, however, by compatible with other inquiry implementations—this should not be used as a way of patching existing inquiries either since this may not be picked up when switching back to use other language resources.
- all files in a subdirectory `Lexicons` with extensions `.lexicon`: used for adding language specific lexical items. These files can also be loaded as a group separately from other resource components by the appropriate linguistic object focusing (Section 5.6.1) or by the command ROOT:<:Load lexicon> (see Section 12.2.8).
- in a subdirectory `Domains`, all files with extensions `.loom`: used for adding domain concepts for particular examples.
- if it exists, all files in a directory `Examples` that have the extension `.ex` and `.spl`: used for storing test suites for linguistic resources. These files can also be loaded separately by means of the command <load-examples> (see Section 10.2.1).

The following files provide further language specific conditions or changes to the KPML system and are not required (or recommended!) for general use.

- a file `code-patches.lisp`: used for defining additions that go beyond the current language conditionalizations that KPML offers for its resources. This file is loaded when the resources as a whole are loaded *only when the relevant 'linguistic object' focusing is activated.* (Section 5.6.1). The default action is that such files are *not* loaded.

Note: the injudicious use of any `code-patches` files in a set of resources makes *that entire set* subject to their requirements. That is, if language variety X uses a `code-patch`, then all other varieties should then be in a position either to work with the changes introduced or to *undo* the effects of that `code-patches` (for example, via their own `code-patches`!). Using `code-patches` thus potentially compromises the integrity of all resources defined. Changes that apply to all language varieties are properly positioned as KPML (possibly user-specific) patches (see Section 3.4) and not subordinate to particular language variety directories.

Any other files in the directory will be ignored (unless, of course, `code-patches` explicitly uses them).

The result of performing the operation ROOT:<Load Linguistic Resources> is that a complete resource set (monolingual or multilingual) is loaded as the current set of active linguistic resources. Generation can then proceed for the language(s) defined by those resources.

12.2 Resource definition formats

12.2.1 Resource definition files

Each linguistic resource file should be in the Lisp package `kpml`. They all begin, therefore, with the Lisp declaration `(in-package "KPML")`. Also, in addition to this, a linguistic resource file may include as initial in-line commands:

- `(in-region :name Region)`: which defines the resources following the command to belong to the functional region named.
- `(in-language :languages L)`: which defines the resources following to belong to the language specified (`L` may also be a list of languages).

These commands may be combined as follows:

```
(in-region :name Region :languages L)
```

Resource files created by ROOT:<Store linguistic resources> will have appropriate `in-region` and `in-language` commands inserted automatically.

The scope of an `in-region` command in ended either by the end of file or by a matching:

```
(end-region)
```

This is also inserted automatically in files created by <Store linguistic resources>.

The individual types of objects in the linguistic resources supported by the development environment and their definition forms are discussed in Sections 12.2.4–12.2.13.

12.2.2 General language property declarations

It is possible to define general properties that the language variety or varieties maintained in a directory should have as a whole: these are placed in the file `properties.lisp`. The kinds of information typically maintained in this file include:

- language morphology style declarations,
- standard inquiry default environment sequences to be used on starting generation with the resource set,
- associations between particular languages and fonts,
- systems to be standardly disabled,
- general string clean-up operations for the resource set.

These are used as follows.

12.2.2.1 Morphology style declarations

Various options are available for handling morphology. The most common ones are:

- Systemic morphology is adopted: that is, the resource definitions include systemic networks that describe that various morphological patterns of a language variety and their realizations.
- Resource-external morphology is adopted: that is, the resource definitions assume that the morphological features that they use will be interpreted by some non-systemic component of KPML. One example of such a resource definition is the Nigel grammar of English, for which the Penman system provided hardcoded English morphology. This hardcoded morphology is inherited by KPML and so can be used if required.[1]
- KPML-external morphology is adopted: that is, the the resource definitions assume that the morphological features that they use will be interpreted by some component that is entirely external to KPML. The German grammar variant used in the TechDoc project (Rösner & Stede 1994), for example, uses the MORPHIX component for German morphology (Finkler & Neumann 1988) rather than a KPML component. Such interfacing is straightforward, but requires redefinitions of two internal KPML-functions.

The first two options are supported by the following declaration:

```
(define-language-morphology-requirements
  :language LANGUAGE
  :systemicized TF
  :generator-function FN)
```

[1]The version of Nigel released as a KPML-resource set does, however, include systemic resources for morphology. This provides a more flexible and transparent representation of the linguistic resources at word and morpheme rank, but increases the generation time a little since further cycles through the grammar are required.

This defines the language variety `LANGUAGE` to either assume systemicized morphology (when `TF` is true) or not (when `TF` is nil), and to use the function `FN` as the mapping from features used in the systemic linguistic resources (i.e., in classify and inflectify realization statements: see Section 12.2.5) to features that are found in the lexicon for the language (see Section 12.2.8). The latter is optional and if not provided the resources are assumed to use the same features in lexicon and systemic networks.[2] If resources are created by inheritance, they also inherit the morphology requirements, including the generator function.

For use of the latter option, interested users should contact the author.

12.2.2.2 Standard default environments

As described in Section 7.4.4, it is possible to define sets of inquiry defaults that may be activated and deactivated at will during generation. It is also common that a given language variety defines a standard set of environments that simplify the semantic specifications that need to be given for that variety (see also Section 12.2.14). For example, both the English and Dutch resources assume that the following default environments hold.

- `present-tense`
- `basic-assertion`

These mean that any semantic specification that does not specify otherwise will receive inquiry responses that are appropriate for constraining a simple positive assertion in present tense to be generated. Importantly, if a semantic specification is given that lacks the necessary information, and no defaults are present, generation will be suspended and the user or calling process will be asked to provide this information.

Since the use of such defaults is commonplace for Penman-style linguistic resources, the following form is provided for declaring a standard set of environments that will be activated whenever generation is attempted with the language variety concerned. This ensures that switching into a language variety does not lose the minimal sets of defaults necessary for simple generation.

```
(define-language-standard-defaults
  :language LANGUAGE
  :defaults LIST-OF-DEFAULTS)
```

The `LIST-OF-DEFAULTS` should be a list of defined default environment names. The definition for English is, for example:

```
(define-language-standard-defaults
  :language ENGLISH
  :defaults (basic-assertion present-tense))
```

The definition forms for these default environments are described below in Section 12.2.14.

[2]In older resources—for example, the Nigel grammar and resources created from this resource—the lexical features and the grammatical features belong to disjoint symbol spaces and so require a mapping from one to the other. This is being gradually changed as time permits (see the linguistic resource descriptions accompanying those resources).

12.2.2.3 Language-font associations

It is possible to define particular associations between fonts and languages. This can be used to alter the appearance of generated texts in various KPML windows. More significantly, it permits the use of languages with other writing systems than the Latin alphabet. The mechanism described here provides support for single-byte font encodings; the selected fonts must have been installed for the X-server being used in the normal way (see the system administrator if necessary).

Font selections normally only have an effect for generated results pop-up windows (Section 7.10) and generated structure graphs (Section 7.9 and 10.2.5).[3] Examples of these usages are shown in Figures 12.1 and 12.2. In Figure 12.1 contrastive generation has produced pop-up generation windows for English and Greek; only the window for Greek is affected by the font change.

Language-font associations can be defined most simply with a declaration of the form:

```
(define-language-font-requirements
  :language :english
  :font
  "-b&h-lucidabright-demibold-r-normal--20-140-100-100-p-118-iso8859-1")
```

This means that, whenever English generated results are produced, they will be shown using the X-font with the name identified under the `:font` parameter. The font identifier is that used for the X-release font aliases.

The following slight variation on this allows differing selections of fonts for the *inspector* window and the generated string pop-ups.

```
(define-language-font-requirements
  :language :english
  :inspector
  "b&h-lucidabright-demibold-r-normal--20-140-100-100-p-118-iso8859-1"
  :popup
  "-adobe-courier-bold-i-normal--0-0-0-0-m-0-iso8859-1")
```

Note: it is in all cases the responsibility of the user to ensure that the requested fonts exist and are accessible to the KPML process! Setting a font requirement without access causes a string of error messages concerning the unlocatable or unknown font. It is possible to globally disable the font selection facility by making the appropriate selection under the ROOT:<Options> command.

If particular resources require non-standard fonts, this will be clearly documented in the individual resource descriptions. Information about where to obtain the necessary fonts should also be given there.

[3] Setting the global flag `*global-font-switching*` to true will cause all information displayed in the *inspector* and *development* windows to be effected however. Such global font changes take effect when an appropriate <Set Language> is issued.

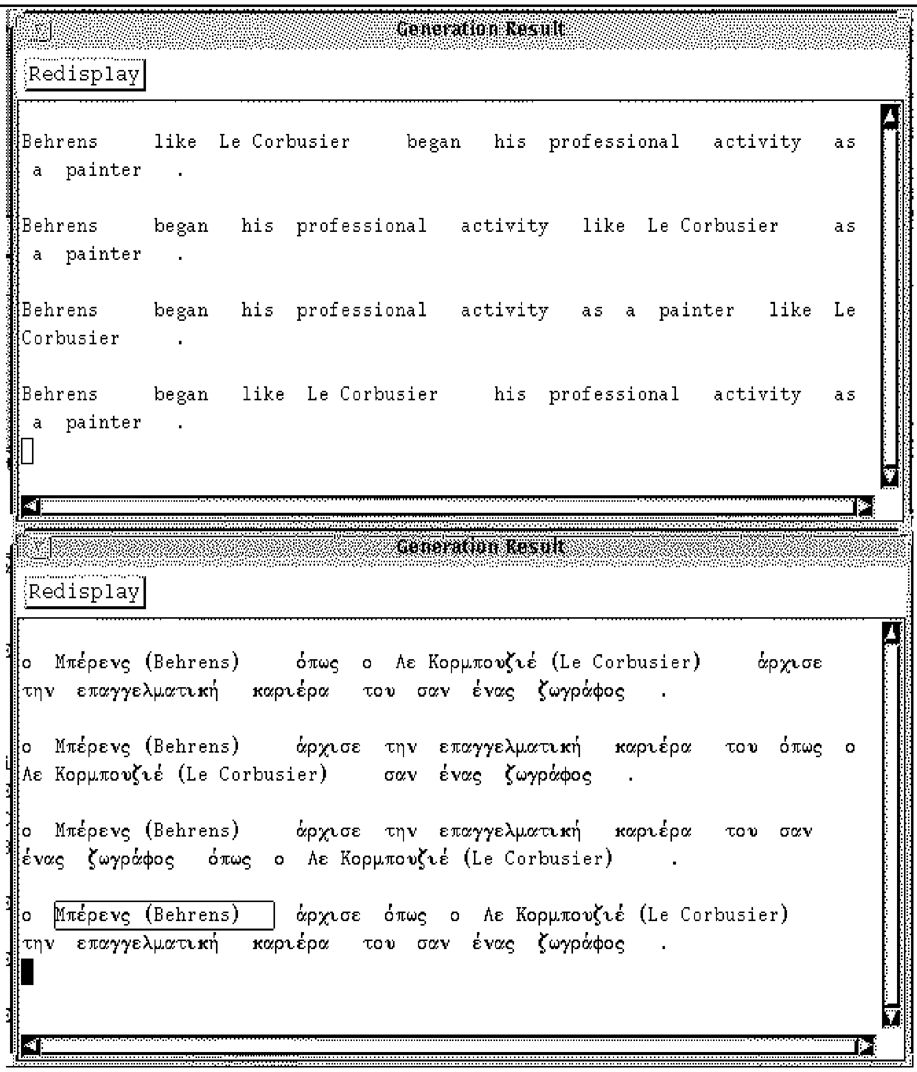


Figure 12.1: Contrastive generation in English and Greek using font associations for Greek pop-up *generated result* windows

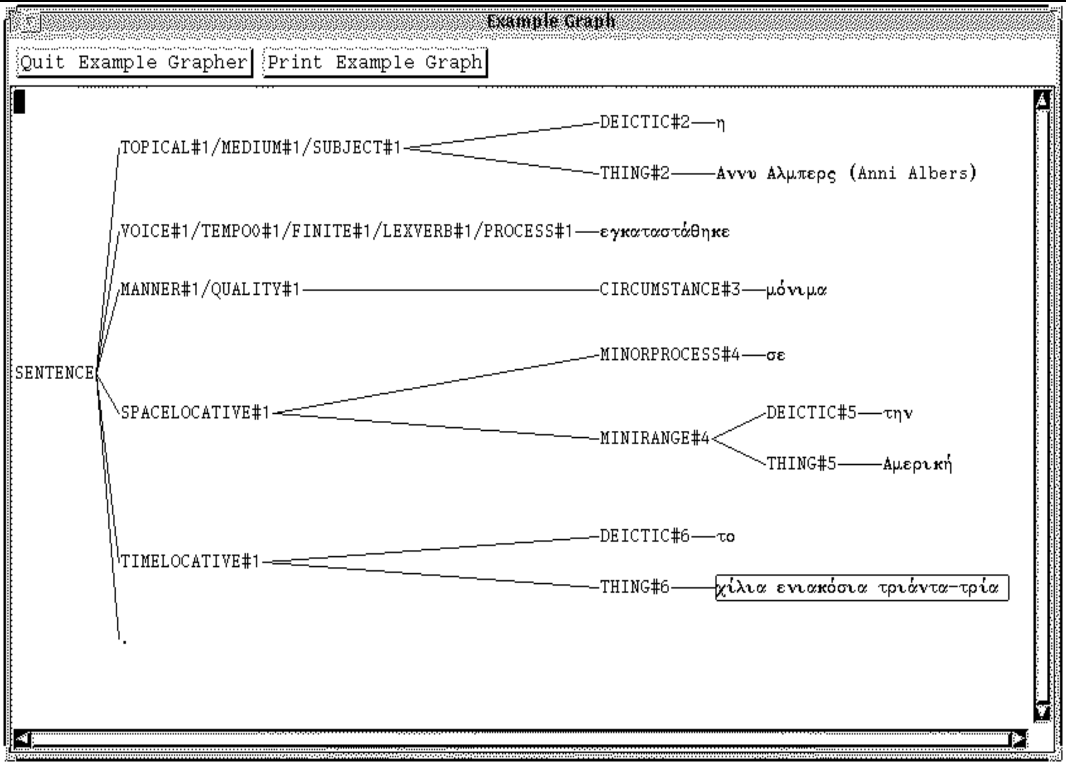


Figure 12.2: Generated structure graph using font associations for Greek

One further possibility for displaying generated strings with different writing systems is to pass the results of generation back to GNU Mule.[4] This can be triggered automatically by using the special font name `:mule` in a language font requirement declaration. GNU Mule must have been installed previously and KPML started within a Lisp buffer within Mule as usually done within Emacs; as always, the user is responsible for ensuring the appropriate software has been installed.

When Mule is specified for the generated string pop-up window font of a language, strings generated in that language will appear in a newly created Mule editor buffer instead of in a *Generated Result* pop-up window from KPML. As an example of use, the following declaration defines the language variety `:Japanese` to use Mule as its output medium. Note that it makes no sense (and is ignored) to specify `:mule` as the output font for the inspector.

```
(define-language-font-requirements
  :language :japanese :font :mule)
```

Subsequently, generating examples with the pop-up generated string flag set causes the strings generated to appear in Mule editor buffers. This is illustrated in Figure 12.3 where two generations of a single example are shown—one using lexical items defined using the Roman alphabet and one using lexical items defined using Mule character codes for Japanese. Such results might be obtained by generating in contrastive generation mode, for example.

[4]GNU Mule is the Emacs-extension permitting editing with many different character fonts, including Japanese, Chinese, Vietnamese, Thai, Arabic, Russian, etc.

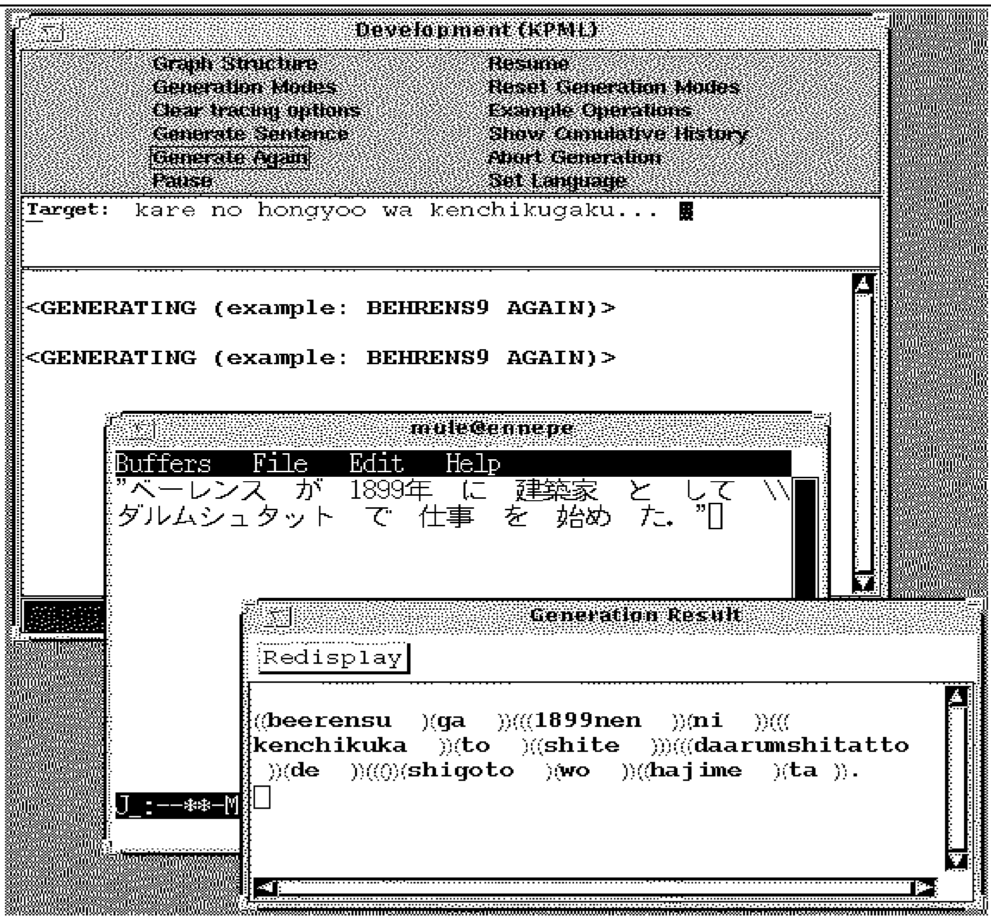


Figure 12.3: Use of Mule for extended character displays

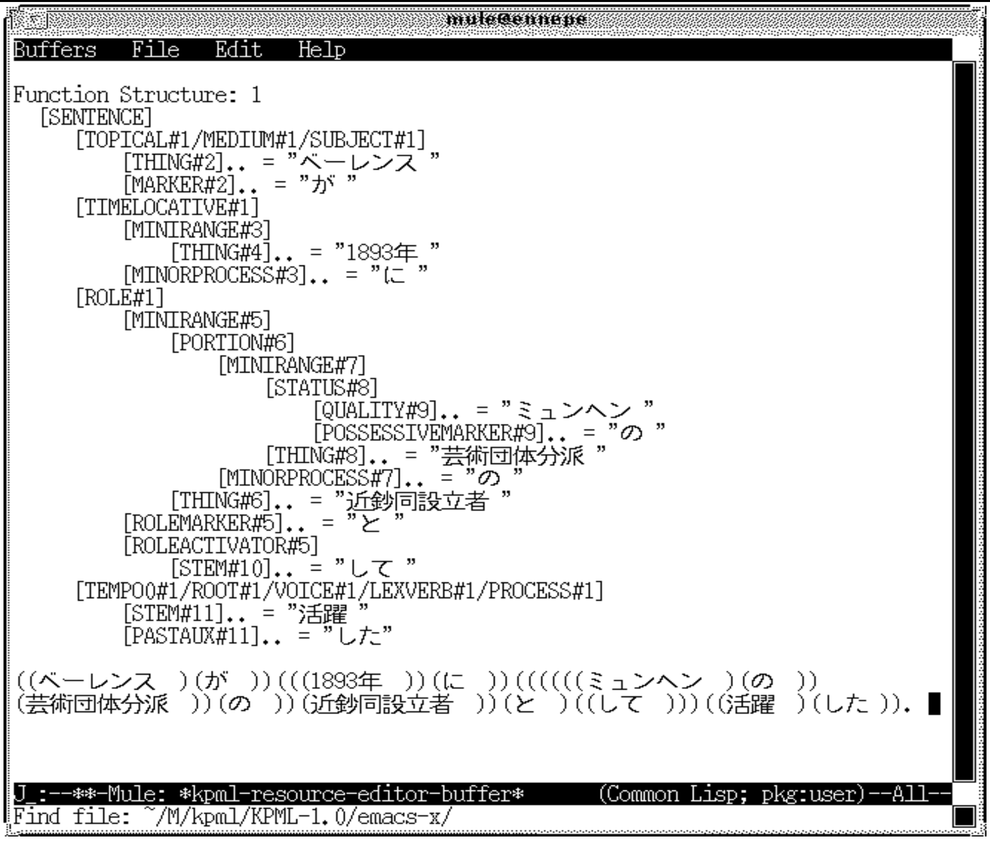


Figure 12.4: Use of Mule for showing grammatical structures filled by Mule-compatible lexeme definitions

Since the character codes for Mule are largely incompatible for those used within Common Lisp, it will *not* make sense to display generated strings or structures using such lexical items within KPML-maintained windows, such as the *Inspector* or the structure graphers. However, since the information displayed in the Mule editor buffer is also sensitive to the *Flag* options for displaying constituent structure (Section 5.4.2), it is possible to obtain a view of the grammatical structure of such strings. An example showing the grammatical structure displayed in a Mule buffer is shown in Figure 12.4.

12.2.2.4 Disabling systems

The only language specific customizations foreseen at present are differing 'disable system' declarations (cf. Section 7.5.2.4). These declarations have the following form:

```
(disable-system 'System-name :Language)
```

12.2.2.5 Final string cleaning-up operations

A number of simple string operations are usually performed on the final string generated before it is presented back to the user or calling process. These include the correction of certain punctuation idiosyncracies (such as the fact, for example, that English does not like combinations such as "—," whereas German does), etc. This simplifies the punctuation rules considerably since these can remain purely functionally and structurally defined, leaving the cleaning up of anomalies (such as perhaps ",?" for a relative clause that appears finally in a question, etc.) to the final string clean-up rules.

String clean-ups for a language variety or resource set are defined with the following declaration.

```
(define-string-cleanups
 :language <LG> :cleanups <CLEANUP-OPS>)
```

The form of `CLEANUP-OPS` is a list of string replacements. The replaces are carried out in the order given. The format of each string replacent is:

```
(NEW OLD)
```

where `NEW` and `OLD` are either strings that will be used in a replacement operation or themselves lists of the form `(type id)`. When `type` is `:lexeme`, then the string used for replacement is the *spelling* slot of the identified lexeme in the currently active language. When `type` is `:string`, then the replacement component `(:string x)` is equivalent to the simple component `x`. `x` must in this case be a string. Thus, an replacement operation

```
("?" ",?")
```

will ensure that whenever ",?" might have been generated (e.g., by the punctuation rules) it will be replaced by "?" in the final string returned as the result of generation.

Note that any string replacements can be carried out at this stage of processing; this may or may not be linguistically defensible. In any case, the mouse sensitivity and annotation of elements in the generated result is preserved across replacements. Appendix D gives an overview of places where the user or external component can cause changes to occur in the string generated.

These conversions also provide for a simple mechanism by which structural embedding, such as

[A and [B and [C and [D ... and Z] ...]]]

can be readily converted to a flat appearing string "A, B, C, D, ..., and Z". The mechanism works in conjunction with the punctuation rules: these impose structural markers (typically `*[*` and `*]*`) which are then processed in the string clean-up phase. The operation sequence:

```
(","                        " & *[*")
(" "                        "*[* ")
(""                         "*[*")
((:lexeme and-conjunction) "&")
```

is sufficient for effecting the change when the corresponding grammar inserts `&` as the structural marker for conjunction and a language variety defines a lexeme called `and-conjunction` for how the final explicit conjunction is to appear in surface strings.

12.2.3 Language variety range declarations

Particularly for multilingual resource sets, it is important the KPML system knows which language conditionalizations it must expect in the resource definitions. For this reason, a file `language-range.lisp` will typically define the varieties dealt with by a given language directory.

The contents of the language range file is typically of the form:

```
(ensure-language-range () :english :german :french)
```

This ensures that KPML can interpret language conditionalizations involving the specified languages.

12.2.4 Systems

A typical system is shown below; this is the system called APPARENT-REALITY which has two features that may be selected - [real] and [apparent]. The first of these has no realization statements associated with it but the second does; realization statements are described in Section 12.2.5 below. The entry conditions for the system are rather complex; they are given as the logical formula under the `:inputs` slot. Only when this condition is true is the choice represented by the system available to be made.

```
(system
   :name APPARENT-REALITY
   :inputs
         (AND
          (OR MATERIAL MENTAL VERBAL IDENTIFYING EXISTENTIAL CIRCUMSTANTIAL
             POSSESSIVE (AND EQUATIVE INTENSIVE) INCHOATIVE-ASCRIPTION
             REAL-ASCRIPTION)
          (OR FINITE-CLAUSE FINITIVE))
   :outputs
          ((0.5 REAL)
           (0.5 APPARENT
                (INSERT REALITY)
                (CLASSIFY REALITY APPEARANCE)
                (INSERT REALITYDEPENDENT)
                (INFLECTIFY REALITYDEPENDENT STEM)
                (INSERT TOREALITY)
                (LEXIFY TOREALITY TO)
```

```
                (ORDER TOREALITY REALITYDEPENDENT)))
  :chooser REALITY-CHOOSER
  :selector CHOICE-MASTER
  :region NONRELATIONALTRANSITIVITY
  :metafunction EXPERIENTIAL)
```

The meaning of the additional slots is as follows:

- `chooser`: gives the name of the chooser (see below) corresponding to this system,
- `selector`: gives the name of the function that chooses between grammatical features (only one such function is provided by the system: the function `kpml::choice-master`; if the user wanted to provide some other function, however, this is where it could be specified),
- `region`: the functional region to which this system belongs,
- `metafunction`: the metafunction to which the region belongs.

Definitions can be evaluated as ordinary Lisp forms once the development environment is loaded.

Note that if an individual grammatical system is redefined in any language, then it is necessary for the system to reestablish the network connectivity for that language. KPML tries to recognize when this is necessary itself in order to remove this from the actions the user has to perform. The operation is actually performed by invoking the Lisp function `reset-system-network` (Section 14).

12.2.5 Realization Statements

12.2.5.1 Introduction

Grammatical systems are not directly concerned with specifying structural constituency but, rather, with specifying the set of grammatical features that a structural product as a whole will instantiate. Thus, the sets of grammatical features produced by making all the choices available in the grammar respecting the interdependencies defined by the network are related to actual linearized syntactic structures via *realization statements*. The process of making all the available choices that the grammar presents is called *traversing the grammar network*. Sentences are therefore generated by a succession of grammar network traversals, or passes through the grammar, one for each major constituent to be produced (cf. Section 2.3.1).

Each traversal of the grammar produces then a collection of grammatical features and each grammatical feature may have associated with it a set of realization statements. These realization statements successively constrain the structure that the grammar is producing. The example system APPARENT-REALITY above shows a number of realization statements that are performed upon selection of the feature 'apparent'.

Realization statements are defined in terms of *functional operations* upon *grammatical functions*. A grammatical function describes the function which a particular constituent is performing in a pass through the grammar. For instance, at the clause level a particular constituent might be functioning for some language as the subject of the clause, so it will be partly defined in terms of a grammatical function called 'Subject'. Similarly, the realization statements associated with the feature 'apparent' shown above concern the grammatical functions: 'Reality', 'Realitydependent', and 'ToReality'. Each pass through the grammar is committed to the generation of a particular level of structure; in systemic terms these levels of structure, corresponding to the major constituents of the product being generated, are termed *ranks* (cf. Figure 2.2).

Thus, in systemic-functional grammar in general, grammatical structures are interpreted as configurations of grammatical functions. That is, particular choices in the grammar will lead to grammatical functions being present, will constrain them to occur together with certain other functions in particular orders, and will further constrain their linguistic realization as constituents. For example, one grammatical feature might constrain the function 'Process' to be present, while another might constrain the functions 'Actor' and 'Subject' to be 'conflated', i.e., both of these functions will become defining components of a single constituent analogous to unification in a grammar implemented in such terms, while another constrains the 'Subject' to be a Singular nominal group. Traversing the grammar network therefore causes a list of functions to be accumulated, along with information on how they are to be combined and ordered, and constraints on how they are in turn to be realized by subsequent traversals of the grammar. The result is a grammatical unit such as a clause or a phrase, completely specified at that rank, although awaiting subsequent grammar traversals to provide the internal linguistic details of its constituents. Grammar traversals continue until constituents have been constructed at a fine enough scale to be realized as words or morphemes rather than as constituents requiring further grammatical organization. All of the structures shown in Chapters 7 and 10 can be seen to be organized in these terms.

In summary, grammatical constituents are defined in terms of combinations of grammatical functions. Configuration of these functions, or *function bundles*, are built up by the interpretation of the grammar's realization statements. Realization statements are expressed in terms of *realization operators* and are triggered by particular choices in the grammar; each choice of grammatical feature from a system in the grammar network may have some particular set of realization statements associated with it which, upon the selection of that choice, will cause the operations necessary for the distinction that that system's choice represents to be reflected in the structural result. By this means, the choices made during execution of the grammar successively construct the configuration of functions that constitutes the structural grammatical output of the grammar.

12.2.5.2 Basic realization constraints

The realization operators supported in KPML may be grouped into three functional categories:

- Functions defining particular grammatical constituents are created by the *Insertion, Conflation* and *Expansion* of grammatical functions; these operators therefore specify structure.
- These constituents may additionally have linear ordering constraints imposed upon them by means of the *Partition, Order, OrderAtFront* and *OrderAtEnd* realization operators.
- The operators *Preselect, Agreement, Classify, Outclassify, Inflectify* and *Lexify* all associate features with functions; they are realizational operators in that they are con-

cerned with how constituents are to be realized rather than with their specification as constituents at a given level of structure. Preselect provides control between ranks, e.g., it provides one means of ensuring subject-verb agreement: if the number is determined at clause rank then making the appropriate preselection of 'singular' or 'plural' for the Subject at the Nominal Group rank and for the Finite verb at Verb Group rank would have the desired effect. This is how, for example, the Nigel grammar of English specifies such agreement. An alternative is offered by the operator Agreement, which sets up sister dependency relations in the grammatical features selected. Classify, Outclassify, Inflectify and Lexify relate bundles (or individual functions) to the Lexicon, either as a particular lexical class or as a specific word.

These realization operations may be defined in more detail as follows:

- Structure specifying realization operators:

  1. Insert – (`Insert Function`) – states that the grammatical structural result of this pass through the grammar will necessarily contain the grammatical function FUNCTION as a defining component of one of its constituents.
  2. Conflate – (`Conflate Function1 Function2`) – states that the named grammatical functions will both be defining components of the *same* constituent. Alternatively, from the perspective of the constituents being constructed, some single constituent comes to include both the named functions as defining components. Within the systemic-functional view, therefore, syntactic constituency is decomposed according to grammatical function, which is taken as basic for structure. Typically the grammar will follow several independent lines of development in each pass, (corresponding to different kinds of functional reasoning), which are ultimately reconciled within a single structural product by the application of the conflation operator. For example, if an ideationally-based chain of reasoning has established that some entity functions as an Agent in its clause, while 'simultaneously' a topicality-based chain of reasoning has established that that same entity functions as Subject in its clause, then performing the conflation of the functions Agent and Subject effects a combination and reconciliation of these lines of reasoning by stating that the grammatical functions Agent and Subject both co-constrain a single clause constituent; that constituent is then functionally multiply labelled. Conflation shares some similarities and historical roots with the notion of unification employed in Functional Unification Grammar (Kay 1979) and its descendents.
  3. Expand - (`Expand Function1 Function2`) - specifies the second grammatical function as a constituent of the first, but *within the same rank*. For example, (*Expand* Mood Subject) means that the function Subject is necessarily contained as a sub-constituent of the constituent labeled by the function Mood, which is a direct clause-level constituent. In this case, there is also a corresponding (*Expand* Mood Finite) realization statement elsewhere in the grammar. Thus, the complete clause structure is analysed as possessing a single constituent labeled the Mood constituent, which in turn has two sub-constituents, labeled by Subject and Finite. This Mood subconstituent does not constituent a separate rank in the grammar however, which is the normal means by which constituency is constructed. The combination of Subject and Finite functions as a significant unit for the clause but

it does not constitute a *structurally* distinct category as would be required to grant it rank status along with clauses, nominal phrases, prepositional phrases, etc.

- Linear ordering operators:

  1. Partition - (`Partition Function1 Function2`) - orders the first function anywhere to the left of the second. This is the least restrictive of four operators that constrain the relative ordering of the grammatical functions inserted into structure.

  2. Order - (`Order Function1 Function2`) - orders the first function immediately to the left of the second. Here the ordering constraint requires that no other constituent can occur between the functions selected, in contrast to the case with Partition.
  3. OrderAtFront - (`OrderAtFront Function`) - orders the function as the leftmost constituent of the level of structure to which the function most immediately belongs.
  4. OrderAtEnd - (`OrderAtEnd Function`) - orders the function as the rightmost constituent of the level of structure to which the function most immediately belongs.

In addition to these explicit statements of order that are triggered when appropriate grammatical feature selections are made during grammar traversal, there are also a collection of *default ordering constraints* that are appealed to when the explicit ordering information is not sufficient for constraining the order of constituents sufficiently for a structural result to be achieved. These default ordering constraints provide a convenient place to state largely invariant or default orders that occur with high frequency; they do not alter the functionality of the grammar. The definition form for default orderings is given in Section 12.2.12.

- Inter-rank realizational operators:

  1. Preselect - (`Preselect Function Grammatical-Feature`) - associates the grammatical feature with the function. This calls for the constituent that the named function labels to be realised by an additional traversal of the grammar which must at least include the selection of the grammatical feature specified. Preselection only operates between ranks. e.g at clause rank a preselection can be made for the group rank, but not for some other element at clause rank or below group rank. The Preselect operator is what triggers recursion in the grammar. When a particular feature is preselected it causes the grammar to be reentered at the Rank system once the current pass through the grammar is completed.

     Preselect functions by adding a list of features that pre-specifies features that must be selected during the traversal of some constituent. This list is obtained by 'path augmentation' whereby the feature mentioned in the preselect statement itself is used as a root for collecting entailed features backward, i.e., leftwards, through the systemic network. Path augmentation does not proceed through disjunctive entry conditions, however. Therefore an augmented path is not necessarily complete and further constraints may need to be given (in the form of further preselections) in order to obtain the full constraints desired. How complete an augmented path will be can be simply obtained by the command INSPECTOR:<Show Path To *feature*> (Section 6.5.3.4).

2. Agreement -

   ```
   (Agreement F1 F2 (f11 f21)
                    (f12 f22)
                    (f13 f23)
                    ...))
   ```

   defines an agreement/prosody domain to hold over the grammatical functions F1 and F2, such that the selection of the grammatical features f1i during the realization of function F1 constrain the automatic selection of corresponding features f2i during the realization of function F2. Note: if a dependency chain is broken (i.e., X depends on Y depends on Z, but Y does not appear, then the indirect dependency X depends on Z is not enforced.[5]

3. Classify - (`Classify Function Lexical-Feature`) - associates the lexical feature with the function. This is similar to preselection; however, whereas preselect operates between different ranks of the grammar, classify sets up an association between a grammatical constituent and features drawn from the lexicon.

4. Outclassify - (`OutClassify Function Lexical-Feature`) - is similar to Classify except that 'not lexical feature' is associated with the function. Thus, (*Outclassify* Finite negative) means that the function Finite may *not* come to possess the lexical feature Negative.

5. Inflectify - (`Inflectify Function Inflectional-Feature`) - associates the inflectional feature with the function. This is again similar to Classify, but is operative at the level of morphological organisation rather than at that of lexical items. Note that when systemicized morphology is being used, this realization statement is largely equivalent to 'preselect'.[6]

6. Lexify - (`Lexify Function Word`) - realizes the grammatical function as the particular lexical item WORD. This is the limiting case of a classify operation; rather than specifying some set of lexical features that constrain the possible lexical items that may realise the selected function, a single lexical item is specified. WORD is the name of a lexical entry defined in the lexicon.

The realization statements used in Penman-style linguistic resources are also described in Matthiessen & Bateman (1991, pp95-97). Proposals for their respecification in terms of unification and classification formalisms can be found in, for example, (Kasper 1987, Kasper & O'Donnell 1990, Bateman, Emele & Momma 1992).

KPML provides two modes of graphing systemic networks where the realization statements associated with particular features are shown in the graph (Section 6.2.1). Realization statements can either be shown in the definition form, as described here, or using the more compact, standard systemic notation. This latter is the default. The realization statement notation is summarized in Table 12.1.[7]

[5] Note that this is an additional realization operator over those defined in Penman-style resources; reports of experience with its use would be appreciated.

[6] Although one difference is that use of inflectify allows use of the lexicon to check for idiomatic realizations of features: i.e., irregular forms. Theoretically there is no reason why this should not apply to higher ranks for idioms in general, but this is not currently supported in KPML.

[7] The notation actually extends the standard somewhat, since not all the realization statements supported here are standard.

Realisation Statement	Systemic Notation
Insert	+ Function
Conflate	Function1 / Function2
Expand	Function1 (Function2)
Partition	Function1 ... Function2
Order	Function1 ^ Function2
OrderAtFront	# ^ Function
OrderAtEnd	Function ^ #
Preselect	Function : feature
Agreement	(no standard)
Classify	Function :: feature
Outclassify	¬ Function :: feature
Inflectify	Function ::: feature
Lexify	Function ! LEXEME

Table 12.1: Realization statements and systemic notation

12.2.5.3 User-defined realization operators

It is possible to define new realization operators. The user needs simply to define a function of the same name taking the appropriate number of arguments. In addition, however, new realization statements should always be defined along with a declaration of the form:

```
(define-realization-operators NEW-OP)
```

This is necessary so that internal interpretation routines can appropriately decompose system definitions and to set up internal records.

12.2.5.4 Morphological realization constraints

KPML provides additional realization operators for working within the word and morpheme ranks of a grammar. These operators are more experimental than the standard operators described above and may change as more experience is gained with their use with a wider range of languages.

The morphological realization operators divide into two classes:

- operators that associate a grammatical constituent (typically a word or its subparts) with some linguistic material (lexeme or morpheme),
- operators that perform morphologically motivated perturbations of the selected linguistic material.

The definitions of the latter class are for the present left deliberately simple and user-extensible.

The first class consists of the operators: `preselect-substance`, `preselect-substance-as-stem`, and `preselect-substance-as-property`. These all act in an identical manner and are used in realization constraints of the form:

```
(preselect-substance Function morpheme-name)
```

This serves to associate the identified morpheme (`morpheme-name`) with the identified grammatical unit (`Function`). It is the morphological equivalent to `lexify` described above. The morpheme name refers to a lexical entry. The term 'preselect substance' is intended to be reminiscent of the fact that this realization is in effect an inter-stratal preselection from lexicogrammar down into the phonology or graphology—even though KPML does not yet support these lower strata explicitly.[8]

Since it is often the case that a recognized unit within the lexicogrammar can have several distinct renderings in terms of phonological/graphological forms (cf. the notion of 'stems'), provision is made in the `preselect-substance-as-...` forms of the realization operator for selecting differing forms depending on specified features. Whereas `preselect-substance` takes its definition of the linguistic material from the string held in the `:spelling` slot of the named morpheme, `preselect-substance-as-stem` takes instead the string held under the `:stem` slot. The `preselect-substance-as-property` generalizes upon this and takes its linguistic material from an identified element from the value of the `:properties` slot. It is used in constraints of the form:

```
(preselect-substance-as-property Function property)
```

This constraint indicates that the form of the linguistic material to become the realization of the grammatical function 'Function' is the value of the property 'property' as found in the `:properties` slot of the lexical item associated with 'Function'. This latter association will typically have been established upstream in the generation process on semantic grounds.

The second class of morphological realization operators form an open class of, currently, string operations. They are used in grammar definitions by giving a single argument specifying a grammatical function. The effect of the operation is then to alter the then current realization associated with the grammatical function (which will be a string) in some regular fashion.

Examples are as follows; their operation is indicated by transforming the input string `"abcde"`.

- chop: removes the last character (producing `"abcd"`) – used for some English graphological alternations (e.g., "use"/"using"),
- strengthen: doubles the last character (`"abcdee"`) – used for some English graphological alternations (e.g., "run"/"running"),
- weaken: changes the last character to an "i" (`"abcdi"`) – used for some English graphological alternations (e.g., "ease"/"easily"),
- span: removes the penultimate character (`"abce"`) – used for some Dutch graphological alternations.

[8] For early experiments in this direction, see, for example, Sefton (1990).

```
;;; Definition for the realization operator 'strengthen'.
;;; This is used in grammatical systems by writing realization
;;; constraints such as (strengthen Head).

;;; First, announce the operator to KPML...
(define-realization-operator STRENGTHEN)

;;; second, define the desired transformation...
(defun STRENGTHEN-MORPH (spelling)
  (string-append
    spelling
    (subseq spelling (1- (length spelling)) (length spelling))))

;;; finally, define the realization operator...
(defun STRENGTHEN (Function-name)
  (morphose Function-name #'strengthen-morph))
```

Figure 12.5: Example definition of a morphological realization operator

This list is clearly not complete, nor particularly theoretically driven. Hence it is to be expected that user might need to extend this list, and that a more theoretically complete treatment will be developed. In the meantime, new morphological transformations of the above sort can be readily defined using the KPML function:

`morphose` ***`grammatical-function`*** ***`transformation`*** [*function*]

`Transformation` is a user-defined Lisp function operating on a string to produce the desired change. `Grammatical-function` is the name of a grammatical function used in a systemic network specification. Realization operators are, in general, the names of Lisp functions that take arguments exactly as they appear in the grammatical system definitions. The definition of the realization operator `strengthen` above could then be given as illustrated in Figure 12.5.

12.2.6 Choosers

An example chooser definition is shown below; this is the definition form corresponding to the graphical version given in Figure 6.9.

```
(chooser
   :name TYPE-OF-BEING-CHOOSER
   :definition
            ((ASK (EXISTENTIAL-Q PROCESS)
                  (EXISTENTIAL
                     (IDENTIFY EXISTENT (EXISTENT-ID PROCESS))
                     (COPYHUB EXISTENT SUBJECT)
                     (COPYHUB EXISTENT DIRECTCOMPLEMENT)
                     (CHOOSE EXISTENTIAL))
                  (NONEXISTENTIAL
                     (ASK (IDENTITY-Q PROCESS)
```

```
                        (IDENTITY
                         (IDENTIFY IDENTIFIED (IDENTIFIED-ID PROCESS))
                         (IDENTIFY IDENTIFIER (IDENTIFIER-ID PROCESS))
                         (IDENTIFY TOKEN (SYMBOL-ID PROCESS))
                         (IDENTIFY VALUE (SYMBOLIZED-ID PROCESS))
                         (COPYHUB IDENTIFIED SUBJECT)
                         (COPYHUB IDENTIFIER DIRECTCOMPLEMENT)
                         (CHOOSE IDENTIFYING))
                        (NONIDENTITY
                         (CHOOSE RELATIONAL-OTHER)))))
              (* Christian "22-Jan-86 18:32:49"))
  :comments ""
  :editor ""
  :date "")
```

This is a Lisp form that may be evaluated; it is also the form that is printed by the inspector command <Print Chooser> when the graphical chooser display is not activated. The chooser actions that are used here, i.e. ask, identify, choose, and copyhub, may be described as follows.

- Ask - puts an Ask type of Inquiry (a Q-inquiry) to the environment. The set of possible responses is predefined and closed.
- Identify - takes a grammatical function and an Identify Inquiry (an ID-inquiry) and puts that Inquiry to the environment. The set of possible responses is open ended. The actual response becomes associated with the grammatical function specified. This association is maintained in a *function association table*; the form and use of this table is described further below.
- Choose - specifies a grammatical feature to choose in the system to which a chooser is attached. If the point in the chooser's decision tree at which the choose operation is situated is reached, then the appropriate choice of grammatical feature to make is the one specified.
- Copyhub - copies the association that exists between one grammatical function and a hub onto another grammatical function.
- * - introduces a comment.

Two additional chooser operations not used in the present example are:

- Pledge - declares that a specified hub is to be considered 'expressed'; subsequent passes through the grammar should not then attempt to re-express already expressed information since responsibility for that expression has already been taken.
- TermPledge - declares that a specified hub is to be considered 'expressed', but by a lexical item rather than by another pass through the grammar.

Actually, all these operations do is place the term mentioned on a list of pledged items. This can be checked in inquiry implementations with the predicate `pledged-p`. Most users need not bother with this possibility.

12.2.7 Inquiries

There are two kinds of inquiries, branching inquiries (Q-inquiries) and identifying inquiries (ID-inquiries). There are also two principal modes of operation for inquiries: implemented and de-implemented (see Section 7.4.7). A typical Q-inquiry is shown below; as usual, this is a Lisp form that gets evaluated when it is loaded.

```
(askoperator
   :name ATTRIBUTE-Q
   :domain KB
   :mode IMPLEMENTED
   :parameters (MODIFYINGRELATIONAL)
   :english
     ("Does " MODIFYINGRELATIONAL
      " represent an attribute, i.e. a modification without an operand?")
   :operatorcode ATTRIBUTE-Q-CODE
   :parameterassociationtypes (CONCEPT)
   :preselectionguidance
     ((ADJECTIVAL-GROUP . ATTRIBUTE)
      (PREPOSITIONAL-PHRASE . OPERANDRELATION))
   :answerset (ATTRIBUTE OPERANDRELATION))
```

The role of a Q-inquiry is to guide generation through a chooser in order that an appropriate grammatical feature be selected. This is normally done, as described below, either by user intervention or an 'implementation' of the inquiry.

In deimplemented mode the English version of this inquiry, as specified in the *English* slot, is put to the user.[9] The possible responses to this inquiry are *attribute* and *operandrelation* as specified in the answerset slot. The user must select the response which most nearly corresponds to the intended semantics of the linguistic unit being generated.

In implemented mode, the operation of an inquiry is more complex. A gloss for above definition would be something along the lines of: the inquiry called *Attribute-Q* interrogates the partition of the environment called the `KB` (knowledge base, in implemented mode this usually refers to the upper model) by invoking the Lisp function specified as the operator code (`Attribute-Q-Code`), which is a function of one argument (called `modifyingrelational` as specified in the parameters slot) of type `concept` (as specified in the 'parameterassociationtypes' slot.)

When the grammatical feature being considered is subject to a preselection—i.e., the outcome of the grammatical system choice has already been constrained by an explicit realization statement, then the *preselection guidance* slot of an inquiry definition is used to check that any prevailing semantic conditions (as revealed by the answers to inquiries) are consistent with the preselection. The preselection guidance is a list of pairs, the first element of which is a grammatical feature that may be preselected, the second the response to the inquiry that is appropriate given the feature preselected. This is an old mechanism which avoids the necessity of a potentially computationally expensive backward-chaining search for entailed paths through the network and the choosers. The user is informed automatically if preselection guidance is required and what that preselection guidance would be.

[9] That this is called `:english` is a hangover from the Penman system; it will be changed to `:gloss` in the near future.

When the inquiry function representing the inquiry implementation is called, the value passed to that function is the 'concept-aspect' of the information associated with the grammatical function used as parameter—i.e., in this case, `MODIFYINGRELATIONAL`. Information can *only* be passed to inquiries in this way, i.e., via some specified aspect of a grammatical function. Aspects are stored in the *function association table* (FAT) and are entered by means of identifying inquiries. The particular information aspects that are supported currently are:

- `concept`: the 'semantico-conceptual correlate' of the grammatical function at issue—for example, its propositional content.
- `modificationspecification`: the textually specific view of the semantic correlate of the grammatical function—for example, the particular propositional content that has been selected as sufficient for some concrete referring expression to be used at a given point in a text.
- `terms`: the set of lexical items that could appropriately realize the grammatical function at issue.
- `term`: the particular lexical item selected for the grammatical function at issue.
- `function`: the label of the grammatical function at issue.

The use of these is illustrated in the example id-inquiry definition given below, in which the `createdassociationtype` slot is used to specify in which field the return result of the inquiry implementation (the Lisp function `dimension-id-code`) is to be placed.

```
(identifyoperator
   :name DIMENSION-ID
   :domain KB
   :mode IMPLEMENTED
   :parameters (PROPERTY1)
   :english ("What is the dimension in terms of which the property "
             PROPERTY1 " is specified?")
   :operatorcode DIMENSION-ID-CODE
   :parameterassociationtypes (CONCEPT)
   :createdassociationtype CONCEPT)
```

When inquiries are interpreted with respect to SPL, the `domain` that the inquiry definition specifies influences where information concerning a hub, or SPL term, may be found. KB-type knowledge is assumed to hold constant for the duration of a sentence; it is therefore possible to use KB knowledge about an entity no matter where in the SPL specification it is given. However, TP type knowledge refers to the textual organization of the sentence and so this type of knowledge may change from instance to instance even within a single SPL specification. For TP inquiries, therefore, the SPL interpreter is only licensed to look in the immediately local term.

12.2.8 Lexicons

A typical lexical item is shown below.

```
(lexical-item
  :name FEED
  :spelling "feed"
  :sample-sentence  "The data is fed into the computer."
  :features  (VERB INFLECTABLE UNITARYSPELLING S-IRR PASTFORM
              EDPARTICIPLEFORM LEXICAL NOT-CASEPREPOSITIONS
              NOT-TOCOMP NOT-QUESTIONCOMP NOT-MAKECOMP NOT-ADJECTIVECOMP
              DOVERB DISPOSAL EFFECTIVE NOT-SUBJECTCOMP NOT-PARTICIPLECOMP
              NOT-BAREINFINITIVECOMP OBJECTPERMITTED
              NOT-OBJECTNOTREQUIRED NOT-COPULA PASSIVE INDIRECTOBJECT
              NOT-THATCOMP)
  :properties  ((PASTFORM "fed" )(EDPARTICIPLEFORM "fed" ))
  :date  "Monday the twenty-third of February, 1987; 4:51:40 pm"
  :editor  "Smith")
```

The features that appear under the `features` slot depend on the concrete linguistic resources defined to the system. The information under the `properties` slot is used for holding idiosyncratic exceptions to general morphological processes. The remaining slots are self-evident.

It is usual that a mapping be provided from sets of lexicogrammatical features to single morphological features such as those that appear in the `:properties` slot. This is necessitated by the fact that property names must still be single atoms and no logical combinations of lexicogrammatical features are permitted. The mapping functions then take specified combinations of lexicogrammatical features (e.g., `present-form` and `first-person-form`) and produce single property names (e.g., `firstpresentform`) such as appear in lexical items. The name of the mapping function can be specified for each language variety as indicated in Section 12.2.2.1. A mapping function must be a function of two parameters: (i) the set of lexical features that have been applied to the currently considered constituent by realization constraints in the grammar; (ii) a flag that is used to indicate to the mapping whether the currently considered constituent is a noun or not.[10] The function should return the name (a single symbol) that represents the lexical *property* that corresponds to the conjunction of the separate lexical features. As an example, the mapping function for English establishes connections such as:

(Firstperson Singular Presentform) → Firstsingularpresentform
(Pastform Plural) → Pluralpastform

Lexicon files from a particular language resource directory (see Section 12.1) can also be loaded independently of other resource objects by means of the command ROOT:<Load Lexicon Files> or by loading linguistic resources with appropriate object focusing (Section 5.6.1).

The current set of lexicon entries loaded can be cleared by the command ROOT:<:Clear Lexicon>.

KPML provides an additional slot (`:stem`) for holding morphological information. How this is used is described in Section 12.2.5.4.

[10] This is a hangover from the Penman system, it will probably be generalized somewhat sometime.

12.2.9 Examples

Linguistic resources may come with files containing example sentences. These examples can be used to test out a grammar without providing semantic specifications or domain models. They also provide a convenient form for test suites showing the coverage of a set of linguistic resources. Special operations are provided for using a set of examples in this way (see Chapter 10).

Examples are typically of two forms:

- an example logical form, or SPL, that generates appropriately given the grammar, domains, and lexicons that are loaded for a given resource set;
- an example set of inquiry responses, that generates appropriately given just the grammar and lexicons loaded for a given resource set.

The two can be combined into a single example record. Typically examples of the former kind are kept in files with extensions '`.spl`', while examples of the latter kind are kept in files with extensions '`.ex`'. The latter kind can be created from the former simply by generating the example with the 'Update environment record' option activated (see Section 7.5.2 and Figure 7.3).

A typical example of the former kind is the following:

```
(example
 :name Behrens4
 :targetform "He studied in Munich in 1890 with Kotschenreiter."
 :logicalform
    ((V-591 / (STUDY)
           :SPATIAL-LOCATING
              (V-585 / (MUNICH THREE-D-LOCATION NAMED-OBJECT OBJECT)
                     :NAME MUNICH)
           :TEMPORAL-LOCATING (V-587 / (THREE-D-TIME) :NAME |1890|)
           :INSTRUMENTAL
                (V-589 / (KOTSCHENREITER NAMED-OBJECT MALE OBJECT)
                      :NAME KOTSCHENREITER)
           :ACTOR (V-590 / (BEHRENS NAMED-OBJECT MALE OBJECT)
                         :NAME BEHRENS)
           :TENSE PAST)))
```

When this example is loaded, `Behrens4` appears in the menus for candidate example generation and, if selected, the logical form under the slot `:logicalform` is used to constrain generation. Since such expressions can rely freely on domain concepts (e.g., `study`, `munich`, `behrens`, etc.), they can only successfully generate when the appropriate domain models have been loaded.

A typical example of the second kind follows; it is the sentence "Yes".

```
(Example
        :name EG88
        :targetform "yes"
        :rootnode EG88
```

```
:includedhubs (EG88)
:selectionexpressions
      ((EG88 RESPONSE-POSITIVE POLARITY
        ELLIPTICAL CLAUSE CLAUSES START))
:editor "BATEMAN"
:date "07/14/88 20:12:42"
:KBenvironment
      ((EG88ACT-POLARITY (POLARITY-VALUE-Q POSITIVE) NIL)
       (EG88ACT (POLARITY-ID EG88ACT-POLARITY) NIL)
       NIL)
:TPenvironment
     ((-TOP--
        (EG88ACT-POLARITY
               (MODIFICATION-SPECIFICATION-ID EG88ACT-POLARITY-PS)
                NIL)
        (EG88ACT
                (MODIFICATION-SPECIFICATION-ID EG88ACT-PS)
                NIL)
        (EG88
                (MODIFICATION-SPECIFICATION-ID EG88-PS)
                NIL)
        NIL)
      (WHERE-AM-I-ID -TOP--)
      (EG88ACT-POLARITY
        (MODIFICATION-SPECIFICATION-ID EG88ACT-POLARITY-PS)
        NIL)
      (EG88ACT
        (POLARITY-ANSWER-Q POLARITYANSWER)
        (ANSWER-Q ANSWER)
        (PROPOSITIONALNESS-Q PROPOSITIONAL)
        (MODIFICATION-SPECIFICATION-ID EG88ACT-PS)
        NIL)
      (EG88
        (SPEECH-ACT-ID EG88ACT)
        (EXIST-SPEECH-ACT-Q SPEECHACT)
        (HEARER-ID READER)
        (SPEAKER-ID PC)
        (SERIOUS-Q SERIOUS)
        (MODIFICATION-SPECIFICATION-ID EG88-PS)
        NIL)
      NIL)
:discoursecontext
        (:speaker I
         :hearer YOU
         :speaking-time TIMENOW
         :realm-of-speech HERE ))
```

The most important components of this structure are the `KBENVIRONMENT` and `TPENVIRONMENT`. These partition the knowledge assumed in the environment into the *knowledge base* and the *text plan* respectively. This distinction is also present in the inquiry definitions (section 12.2.7) under the `:domain` slot. This determines which partition the inquiry is to interrogate for its response. Both KB- and TP-environment slots hold information of the same form: an association list of hub names and the inquiry-inquiry response pairs that are appropriate for those hubs. For example, the first entry of the `TPENVIRONMENT` slot states that when the inquiry *Modification-Specification-ID* is asked of the hub *eg88ACT-POLARITY* then the response appropriate for this example is *eg88ACT-POLARITY-PS*.

The `:included-hubs` field maintains a record of all the hubs that have been realized by rank-level structures, i.e., clause, nominal-group, etc.; and `:selectionexpressions` holds the lists of all the grammatical features selected for each of those rank-level structures, or grammar network traversals.

An additional slot, `:structure`, not shown here for reasons of space, holds the structurally rich version of the generated string that is used to create the mouse-sensitive generated string presentations that appear in the interface. This is also the structure that can be used to good effect by applications that want a more sophisticated presentation of the generated results than the simple strings that result. The internal structure of the 'mouseable structures' is described in Section 14.5.

Note that for the linguistic resources to generate from such an input specification, they need to be run in *de-implemented* mode for this to work (see Chapter 10). Exercise set examples will not run in implemented mode (the normal mode for generating from semantic specifications) and, similarly, examples that are intended to run in implemented mode will not succeed in deimplemented mode. The example runner will automatically switch into de-implemented mode if it is asked to generate an example that does not contain a logical form.

Such example records can either be edited directly or, more usefully, indirectly via the grammar interface by setting the *Update Example Record* flag (see Section 7.5.2). Setting this flag ensures that all relevant information created during generation is preserved in the appropriate slots of the example record. This can be used, for example, for converting an example of the first kind introduced above, containing only logical form, to one of the second kind, where a complete record of the grammatical traversal and semantic inquiry responses is also available.[11] Only such complete examples can be used to support the operations for selecting examples on the basis of which grammatical features or functions they use.

Examples are also multilingual objects, and can be loaded, written, and merged in all the usual modes for multilingual objects generally (i.e., monolingual, contrastive, and multilingual).[12]

12.2.10 Punctuation

KPML provides the same method of controlling punctuation as does the Penman system; the only difference being in the definition syntax and the ability of KPML to maintain different punctuation rule sets for different languages. This method allows functionally and structurally motivated punctuation to be defined drawing on the descriptions generated by a grammar.

The punctuation rules for a language variety are kept in the file `punctuation.gram`. There are three kinds of rules:

- 'pre' punctuation rules,
- 'post' punctuation rules,
- 'post-self' punctuation rules.

[11] In such cases, the logical form is also, of course, preserved.

[12] There are some exceptions in the structure slots that are merged: information that is purely bookkeeping for generation is not merged.

All punctuation rules consist of a set of entries of the form:

```
(<grammatical feature> <grammatical function> <punctuation mark>)
```

The grammatical function identifies the constituent where the punctuation mark (a string) is to be placed.

For 'pre' and 'post' punctuation rules, the grammatical feature applies to a grammatical unit that *includes the designated grammatical function as a subconstituent*. That is, the rule:

```
(NON-THEMATIC-DEPENDENT-BETA DEPENDENT ",")
```

states that when a grammatical unit is generated using the feature `non-thematic-dependent-beta`, then the subconstituent `Dependent` of that unit should be punctuated by a comma. If the rule is a 'pre' rule, then the comma comes before the designated constituent; if it is a 'post' rule, then the comma follows the designated constituent. The above rule is responsible for the fact that hypotactically related dependent clauses following their matrix clauses (this is what the feature `non-thematic-dependent-beta` in the grammar of English means) are separated from that matrix clause by a comma. It is, accordingly, in the English punctuation rules defined as a 'pre' rule, since the structure desired is of the form:

```
[INDEPENDENT "," DEPENDENT]
```

For 'post-self' punctuation rules, the grammatical feature refers to the same grammatical unit as is indicated by the grammatical function. For example, the 'post-self' rule:

```
(INTERROGATIVE SENTENCE "?")
```

states that if a grammatical unit labelled `Sentence` (the top node in a constituent structure) is generated using the grammatical feature `interrogative`, then it should be *followed* by the indicated punctuation. Similarly, the rule:

```
(IMPERATIVE PROJECTED "!")
```

states that the grammatical unit labelled `Projected` should be followed by an explanation mark just in the case that it is realized by a grammatical unit possessing the feature `imperative`. 'Post-self' rules therefore differ from 'post' rules in the positioning of the grammatical feature specified.

Finally, it is possible in 'post-self' rules to use a '`*`' as a wildcard for the grammatical function. This means that it is possible to indicate that a grammatical unit containing a specified grammatical feature is to be punctuated regardless of what grammatical function it is realizing.

The syntactic form of the punctuation definitions is simply:

```
(define-X-punctuation :language :LANGUAGE
  :punctuation-rules '(RULE-1
                       RULE-2
                       ...
                       RULE-N))
```

where the rules have the form indicated above and `X` can be either `pre`, `post`, or `post-self`.

12.2.11 Non-systemic system dependencies

In the case that multiple systems can be entered during traversal of the network, it is possible to control the order in which these candidates are in fact entered. This information in maintained in the file: `ordering-constraints.gram`. The syntax differs again from that used in the Penman system, although automatic conversion (Penman to KPML, not *vice versa*) is provided.

Note: making deliberate use of this information is not recommended, as it compromises the declarative integrity of the resource definitions.

The form of these specifications is as follows (adapted from those for the English grammar):

```
(define-dependency-set
  :language :english
  :name CLAUSE-AREAS
  :systems-list '(ATTITUDE CIRCUMSTANTIAL CLAUSE-COMPLEX CONJUNCTION
                 CULMINATION DEPENDENCY MOOD NONRELATIONAL-TRANSITIVITY
                 POLARITY RANKING RELATIONAL-TRANSITIVITY TAG TENSE
                 THEME VERBAL-GROUP VOICE)
)

(define-dependency-set
  :language :english
  :name  DETERMINATION-AREA
  :systems-list '(SPECIFIC-TYPE PARTIAL-TYPE TOTAL-TYPE)
)

(define-dependency-set
  :language :english
  :name  LEXVERB-CONFLATIONS
  :systems-list '(TRANSITIVITY-UNIT VOICE-LEXVERB
                 LEXVERB-VOICEDEPENDENT AUXSTEM-LEXVERB
                 LEXVERB-FINITE)
)

⋮

(define-system-dependencies
  :language :english
  :dependencies
      '((THING-TYPE-AREA MODIFICATION-AREA-I
         DETERMINATION-AREA (QUANTIFICATION) MODIFICATION-AREA-II)
        (THING-TYPE-AREA ... )))
```

Systems that are potentially candidates for parallel entry are defined by the form `define-dependency-set`. Each set of candidates is named. Thus, for example, one candidate set of systems is the `DETERMINATION-AREA`. In this set, the grammatical systems SPECIFIC-TYPE, PARTIAL-TYPE and TOTAL-TYPE will become available for entry simultaneously. The specification here requires, however, that they will actually be entered in the order given in the list.

The candidate sets are collected together and are used in the definition system dependencies overall. This takes place in the form `define-system-dependencies`, which specifies the relative ordering of the sets of candidates. Thus, in the example above, first systems belonging to the group `THING-TYPE-AREA`, then those belong to the group `MODIFICATION-AREA-I`, and then those of the `DETERMINATION-AREA` are entered. Unnamed sets of alternatives can also be used here by enclosing them in parentheses. The system `QUANTIFICATION`, for example, is given above as following all system sof the `DETERMINATION-AREA` and preceding all systems of `MODIFICATION-AREA-II`.

Definitions of this form can be merged freely during contrastive loading. Multilingual resources use a slightly different form that simply echoes the internal structure of the values of the variables where the system dependency information is maintained. Since the frequent use of this kind of information is not recommended, multilingual support is kept to a minimum.

12.2.12 Default orderings

Default orderings are also specified in the file: `ordering-constraints.gram`. These orderings take the form of lists of preferred sequences of grammatical functions. They are defined for a particular language by specifications of the form:

```
(define-default-orders :language :DUTCH
        :orders '((SUBJECT NON-FINITIVE)
                  (SUBJECT NEGATOR LEXVERB QUALITY ...)
                  (REPRESENTATIVE RSELECTOR PART PSELECTOR ...)
                  (DEICTIC COMPARATOR QUALITY STANDARDINDICATOR STANDARD)
                  (PRE-CARDINAL MILLION THOUSAND HUNDRED SUPRATEN SUBTEN)
                  (PRE-CARDINAL TEMPERER APEX)
                  (PRE-CARDINAL THOUSANDDIGIT HUNDREDDIGIT TENDIGIT
                   UNITSDIGIT DIGIT)
                  (SUBJECT TO-INFINITIVE TEMPO0 FINITE REALITY TEMPO1
                   TEMPO2 TEMPO3 VOICE LEXVERB)
                  (STRUCTURAL TEXTUAL INTERPERSONAL TOPICAL)
                  (IDENTIFIED PROCESS IDENTIFIER)))
```

When the grammatical functions of any sublist occur together at the same rank in a generated structure, then their order—*if not specified otherwise by the explicit ordering constraints present in the grammar*—will be as given in the sublists of the defined default order for the language in question.

Definitions of this form can be merged freely during contrastive loading. Multilingual resources use a slightly different form that simply echoes the internal structure of the values of the variables where the default ordering information is maintained. Since the frequent use of this kind of information is not recommended, multilingual support is kept to a minimum.

12.2.13 Domain concepts and links with the lexicon

The default assumption made by KPML is that the LOOM knowledge representation language is being used. Domain concepts are then typically defined in LOOM and, under the Penman model for interfacing with a domain, are subordinated to concepts defined in the 'upper model'.[13] Macros are also provided for linking such domain concepts with lexical items.

A typical domain concept definition is the following:

```
(defconcept illness
 :is (:and penman-kb::object :primitive))
```

This defines the domain concept `illness` to be a subtype of the upper model concept `object`. Upper model concepts are maintained in the Lisp package `penman-kb`; domain model concepts can be placed in any package (including `penman-kb`) as long as the user knows how to manage the various interactions between Lisp packages and Loom knowledge bases, etc. The simplest incantations for setting up conditions for a domain definition file are the following.

Loom 2.0:

```
(in-package "PENMAN-KB")
(in-kb 'penman-kb)
```

Loom 2.1:

```
(in-package "PENMAN-KB")
(in-context 'ideation-base-0)
```

This makes both the Lisp package and the knowledge base be the same as those of the concepts of the upper model.

Links with the lexicon are then created by the following:

```
(kpml::annotate-concept  illness
  :lex-items
     (ziekte krankheit illness))
```

The elements in the `:lex-items` list are names of lexical items as defined by appropriate lexical item definitions (Section 12.2.8). These names can be conditionalized for individual languages to give more specific definitions in the normal way (Section 12.3). A more appropriate version of the above (which states that the three lexical items `ziekte`, `krankheit`, and `illness` are available for the concept regardless of language and regardless of the individual language conditionalizations of these lexical items) would therefore be:

```
(kpml::annotate-concept illness
:lex-items
(:dutch ziekte
:german krankheit
:english illness))
```

Lexical annotations are stored internally in the hash-table `*concept-annotations*`. The keys to the values are the print names of the concepts, as returned by the knowledge base access function `kb-getconceptname`.

[13] For alternative, more flexible, models of relating domain to upper model concepts, see, e.g., Bateman & Teich (1995).

12.2.14 SPL macros and defaults

SPL macros can be used to simplify the input specifications given to the generator.[14] An SPL macro definition consists of a macro name followed by possible slot values. The definition provides for each slot value a set of inquiry and inquiry responses that are to be placed in the SPL where the SPL macro is used. Coreference of inquiry responses and parameters is indicated in the inquiry/inquiry response sets by variables.

An example definition is the following:

```
(defspl-macro :determiner
  ((a
      :identifiability-q  notidentifiable
      :multiplicity-q     unitary
      :amount-attention-q minimalattention)
   (all
      :current-representative-id ?s1
      :potential-representative-id ?s2
      :set-totality-q (?s1 ?s2) total
      :multiplicity-q multiple
      :singularity-q nonsingular
      :set-totality-individuality-q collection
      :amount-attention-q nonminimalattention)))
```

Following evaluation of this definition, it becomes possible in an SPL specification to specify simply, for example:

```
:determiner all
```

This will be expanded into the set of inquiry and inquiry responses indicated in the definition, producing in this case (with the Nigel grammar of English) a nominal group with determiner 'all'.

The use of SPL macros is provided for compatibility with inputs designed for the Penman system; since in a full generation scenario SPL specifications would themselves be generated automatically, the utility of SPL macros is somewhat weakened. They can also too easily disguise the purely *semantic* nature of the SPL input—as in the above example where it makes it appear that the SPL input contains rather syntactic information concerning choice of grammatical determiners, although this is not the case since this is of necessity expanded into inquiries which are semantic. A further problem is that it is possible with more complex SPLs to choose macro combinations where the inquiries entailed are in partial conflict: this problem is hidden from the SPL-writer by the macros themselves, and so can cause consternation when a macro suddenly stops having its usual effect. For these reasons, SPL macros are not particularly strongly supported or recommended in KPML; they cannot be conditionalized for particular languages. Should problems with macros occur, the user is recommended to replace the macros with the expanded inquiries and to check for possible bad interactions.

[14] Both the SPL macro and default facilities were written by Bob Kasper for the Penman system. This is taken on virtually unchanged in KPML.

A basic set of SPL macros is usually to be found in a set of resources in the file: `basic-spl-macros.lisp`.

SPL defaults, or default environments, provide a way of simplifying SPL input specifications still further. An SPL default environment defines a set of inquiries and their responses which are to be added to all SPL specifications processed while the environment is 'active'. For example, if we wish to specify that, until further notice, all SPLs given should act as if the specifications for present tense were also present, then we can use the definition:

```
(defspl-default present-tense
  :speaking-time-id
    (DEFAULT-ST / time
                :time-in-relation-to-speaking-time-id DEFAULT-ET
                :same-as-q (DEFAULT-ST DEFAULT-ET) same
                :precede-q (DEFAULT-ST DEFAULT-ET) notprecedes)
  :event-time
     (DEFAULT-ET / time
                :precede-q (DEFAULT-ET DEFAULT-ST) notprecedes))
```

This defines a possible SPL default environment named `present-tense`. The inquiries and responses given are those necessary for specifying the semantic temporal relations involved when present tense is used in English.

Since it has often been the case that sets of SPL specifications have been prepared in the context of the Penman system assuming that some set of standard defaults holds, KPML provides a way of declaring that a particular language variety will use a particular set of SPL default environments. This is described in Section 12.2.2.

Note that the KPML provision of default environments is more restricted than that of the Penman system since the interactions of multilinguality and stacked default environments have not been implemented. The remarks given above for SPL macros apply similarly to SPL default environments however, and so their use is not strongly supported in KPML.

12.3 Language variety conditionalization

KPML provides full conditionalization of the linguistic units defined according to language as described in Bateman, Matthiessen, Nanri & Zeng (1991*a*). Any component (represented by either a single symbol or a single list of symbols or further components) present in the definitions of systems, choosers, inquiries, SPL specifications, lexical items, and lexical annotations to concepts can be conditionalized to belong to some specified set of named language varieties. Language varieties are named by a Lisp keyword such as `:english`. A sequence of language varieties states that the following component of the specification is applicable to all the language varieties mentioned in the sequence. The conditionalization applies to the immediately following component only. Thus, the following variation on the definition of the first output feature for the system APPARENT-REALITY illustrated in Section 12.2.4 above:

⋮

```
:english   :dutch (0.5 REAL)
⋮
```

specifies that the feature [real] is only relevant for language varieties `:english` and `:dutch`. If a set of language variety conditionalizations is to apply to more than a single component, then the required components are joined by the '&' symbol. Thus, a slightly more cumbersome way of stating the same conditionalization on the feature [real] would be:

```
⋮
     (:english :dutch 0.5 & REAL)
⋮
```

The conditionalization of an entire unit (i.e., system, chooser, inquiry, example, or lexical item) is achieved by adding the required language varieties into the `:name` slot of the unit. Thus the following would define APPARENT-REALITY to be a system only of the grammar of English.

```
(system
   :name (:english APPARENT-REALITY)
   :inputs
⋮
)
```

A more complex example is shown in Figure 12.6. This represents the possible features concerning grammatical 'gender' as recommended in the working draft of the Eagles (a European Union, LRE project) report on morphology for European languages (Monachini & Calzolari 1994). Whereas the single definition for 9 languages (it is stated in the report that no gender features apply to English) may appear complicated, once loaded into KPML such definitions can be readily decomposed and viewed for subsets of the languages covered. Further, since such definitions can be constructed internally when merging descriptions of different languages, it is possible that no user ever seeks to view the entire multilingual definition. Individual users could focus on particular languages in the overall set without needing to consider the full set.

Figure 12.7 shows the graphical views of the systems from the noun morphology for Spanish, Danish and French, while Figure 12.8 shows a explicitly contrastive view of German and Greek; the distinct views afforded of the GENDER system shown in Figure 12.6 can be directly compared. This is a very direct computational instantiation of the notion of multilingual 'views' inherent in multilingual systemic resources.

```
(system
 :name (:italian :german :dutch :spanish
        :french :portuguese :danish :greek      GENDER)
 :inputs noun
 :outputs
    (:spanish                                   (1  trns)

     :dutch                                     (1  fem-masc)
                                              & (1  context)

     :italian :german :dutch :spanish
     :french :portuguese :greek                 (1  masculine)

     :italian :german :dutch :spanish
     :french :portuguese :greek                 (1  feminine)

     :italian :german :dutch :danish
     :portuguese :greek                         (1  neuter)

     :italian :spanish :portuguese
     :danish :greek
          (:italian :spanish :portuguese :danish 1 & common

           :greek                                1 & masc-fem))

 :chooser GENDER-CHOOSER
 :selector CHOICE-MASTER
 :region NOUN-TYPES
 :metafunction LOGICAL)
```

Figure 12.6: Example highly multilingual system

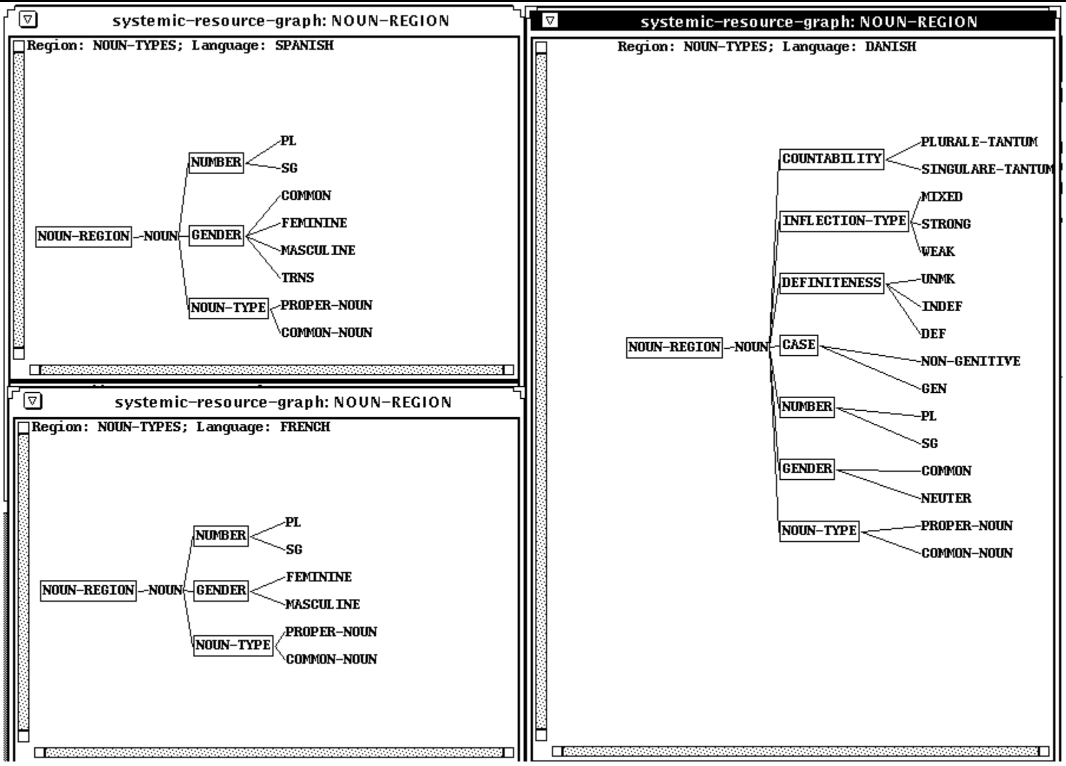


Figure 12.7: Distinct views on a multilingual resource (contrastive)

12.4 Requirements for resource definitions

There are a few constraints that hold for all resource definitions. These should either be met or some action should be taken to defuse the consequences of their not holding.

12.4.1 Special inquiries

The following inquiries should probably always be defined.

`where-am-i-id` This inquiry allows the current place in the constituent structure to be ascertained.

`term-resolve-id` This inquiry selects a lexical item matching the lexicogrammatical and semantic constraints holding for its parameter.

`modification-specification-id` This inquiry provides the connection between the experiential (propositional content) information maintained in the `concept` slot of an entry in the function association table and the textual view of that content maintained in the `modificationspecification` slot.

Definitions of these are to be found in the released resources; standardized implementations of the latter two are also to be found there in the inquiry implementation files; the implementation of the former is a KPML-internally defined function.

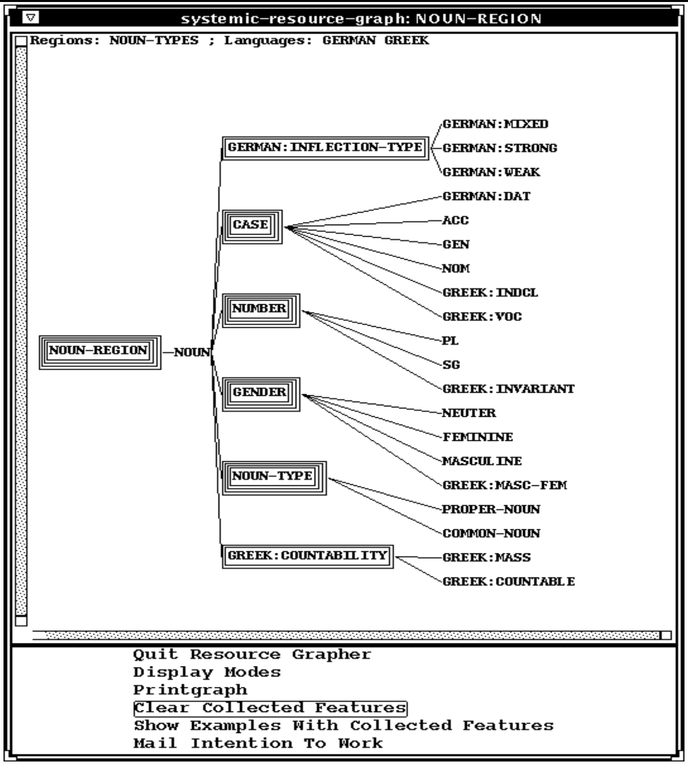


Figure 12.8: Distinct views on a multilingual resource (multilingual)

12.4.2 Special semantic concepts and relations

A few of the upper model concepts are relied upon in internal code; for example, the interpretation of SPL relies upon the distinction between semantic relations (which can stand as roles in SPL expressions) and objects (which can not). In addition, the SPL interpreter uses some upper model concepts for constructing lists of semantic entities. The upper model concepts/relations which should, therefore, always be defined are:[15]

`um-set` refers to a set of objects,

`disjunctive-set` refers to a disjunctive set of objects,

`two-place-relation` refers to two place relations.

These are described in the upper model documentation. The SPL interpreter code does not refer to these concepts name directly; instead it uses the values of the variables: `*spl-set-type*`, `*spl-disjunctive-set-type*` and `*spl-relational-types*` respectively. The latter is a list of concepts/relations which are all taken to root semantic relations.

The set concepts are used in the interpretation of SPL forms of the kind:

[15]These are mostly maintained in lists internal to KPML so customization would also be straightforward if they are not to be defined in the upper model adopted.

- `(spec1 spec2 ...)`
- `(:and spec1 spec2 ...)`
- `(:or spec1 spec2 ...)`

The first two are equivalent to one another and rely on `*spl-set-type*`; the third relies on `*spl-disjunctive-set-type*`. In each case an SPL term graph (see Appendix B) of the set semantic type is constructed with the arguments stored as a list under the `:symbol` slot. Inquiry implementations that wish to use such SPL expressions should therefore be written appropriately if they are to succeed.

Chapter 13

Accessing external information sources

13.1 Semantic information from inquiry implementations

The principal way of interfacing from a body of linguistic resources to external information is by means of the *inquiries* and their implementations. There are very many inquiries, which means that this kind of interface is of necessity of a very broad bandwidth. For (meta-)functionally diverse resources such as systemic-functional resources, this seems to be essential. There are, however, several ways in which this kind of interfacing is simplified. The most significant of these is the provision of an Upper Model for organizing the experiential semantics that linguistic resources, particularly grammars, presuppose. The upper model currently used within KPML is the *merged upper model* motivated in Henschel (1992). A further, *generalized upper model* is under development.

The most immediate interface to the grammatical component is provided by the *Sentence Plan Language* (SPL: Kasper 1989). This notation relies on the existence of an upper model for its interpretation, but not on any particular upper model. Inquiry implementations as usually defined obtain their information from an SPL expression provided as input. However, inquiry implementors are, of course, free to write those inquiries so as to obtain information from any source, not just from the SPL.

Interfacing with any particular knowledge representation language is simplified by means of a very restricted set of access functions. These are the functions by which inquiry implementations access SPL input expressions or underlying upper or domain concepts and relations (currently represented in Loom). Appendix B contains extracts from Bob Kasper's description in the *Penman Reference Manual* concerning these access functions and other internal aspects of the SPL implementation.

13.2 External information from the lexicon

At present this is often done (when necessary for, for example, languages with lexically specified information such as lexical gender) by using inquiry implementations that lift the information from the lexicon directly. This is certainly not particularly elegant, and it also loses the real theoretical difference between accessing semantic information and moving information around within the lexicogrammar. Improved mechanisms for this will probably be made available at some stage.

The KPML function `access-lexical-information` takes a grammatical micro-function (such as 'Subject', 'Actor', etc.) as argument[1] and returns three values: the lexical features defined by the lexical entry, the associated lexical item's identifier, and the lexical entry itself.

This permits the ready definition of inquiry implementations such as the following, which checks whether a particular grammatical function is being realized by a lexical item with the lexical feature [neuter].

```
(defun neuter-gender-q-code (item)
  (multiple-value-bind (features name entry)
    (access-lexical-information item)
      (if (member 'kpml::neuter features)
          'neuter-gender
          'not-neuter-gender)))
```

Or, of course, following the Common Lisp treatment of multiple values, simply:

```
(defun neuter-gender-q-code (item)
  (if (member 'kpml::neuter (access-lexical-information item))
      'neuter-gender
      'not-neuter-gender))
```

The package of the lexical feature `neuter` is given explicitly because of the possibility provided by KPML of moving inquiry implementations across different Lisp packages. The lexical features are always in the Penman package and so this information should be preserved. This is obviated by the functions defined below.

Two additional support functions for handling lexical information are the following:

- `(lexical-feature-present-p Name Feature)`

  This checks whether the lexical item of name `Name` possesses the feature `Feature`.

- `(lexical-class-ascertainer Name Feature-list)`

  This returns the feature of `Feature-list` that the lexical item with name `Name` possesses (if any).

[1]An inquiry defined as taking a parameter of type `Function` provides such objects appropriately (see Section 12.2.7).

Since the most common kind of lexical access inquiry by far needs only to access the lexical item associated with some grammatical function in order to ascertain lexical features that are present, the following two functions provide a convenient combination of the above two functions and `access-lexical-information`.

- `(lexical-feature-present-in-association-p Item Feature :yes Yes :no No)`

  This combines the work of `access-lexical-information` and `lexical-feature-present-p` enabling information to be obtained directly from the grammatical function that is typically provided to a lexically concerned inquiry operator as argument.

- `(lexical-class-of-association-ascertainer Item Feature-list)`

  This similarly combines the work of `access-lexical-information` and `lexical-class-ascertainer`.

The above example inquiry for `neuter-gender-q-code` can now, therefore, be simplified still further to the following definition:

```
(defun neuter-gender-q-code (item)
  (lexical-feature-present-in-association-p
    item 'neuter :yes 'neuter-gender :no 'not-neuter-gender)
```

This standard form also supports automatic conversion to other possible forms (e.g., a typed feature representation) more readily and so is recommended over the use of straight Lisp code.

All four of these support functions use those of their arguments that refer to lexical features as if they were symbols in the `kpml` package. They can therefore be used in any package and nevertheless provide the appropriate lexical access.[2]

13.3 Morphological information from external components

It is in principle straightforward to interface a set of grammatical resources with external morphological components. This does, however, require that the kinds of constraints given in the grammar (normally in terms of classify or inflectify realization statements are directly relatable to the specifications required by the external component. As with all access to external components, it is necessary to provide suitable Lisp definitions of the relevant KPML interface functions.

Users interested in this possibility are invited to contact the author.[3]

[2] In fact, for the very lazy, `lexical-feature-present-in-association-p` assumes as default for its `:yes` case, a symbol identical to the feature sought (given as second parameter), and for its `:no` case, a symbol constructed by prefixing either the yes case, or if this is also missing, the second parameter, with the string held in the variable `*default-negation-prefix*` (which is in turn by default the string "NON").

[3] The relevant KPML/Penman internal function for this is `realize-classify`. This function is called whenever a constituent has had lexical constraints specified for it in terms of 'classifications', i.e., preselections of lexical features. It returns information specifying a lexical item that is appropriate for the constraints specified. If, however, a lexical item has already been selected on semantic grounds (by use of the `term-resolve-id` inquiry), then that is accepted without further investigation.

Chapter 14

Using KPML without the window interface

In this section, we describe the Lisp functions that enable KPML to be driven directly without going via the commands provided by the window interface. Many of these functions are those that underly the window interface commands; some are provided additionally to make operation without the window interface more comfortable. Unless otherwise noted, all symbols are in the `Kpml` Lisp package.

14.1 Blackbox operation as a tactical generator

The standard KPML function for initiating generation is `say`.[1]

`say` *`logical-form-or-name`* `&key :details :full-structure :language` [*function*]

`Logical-form-or-name` must have as a value either the name (a symbol) of a loaded example (Section 12.2.9) or an SPL semantic specification. Generation proceeds for the language specified by `:language`, which defaults to the current language (which is always maintained in the global variable `curlan`).

If `:details` is true, additional information about the generated structure and string are printed (at the window interface if it is present, on `*standard-output*` if not).

The result of the function depends on the flag `:full-structure`. If this flag is false, then only the generated string is returned as result. If the flag is true, then two results are returned. The first is the string as before; the second is a list of string and 'mouseable structure' pairs. The 'mouseable structure' is a structured representation of the generated string that follows the generated linguistic structure. Precisely how closely it follows the generated structure can be fine-tuned by setting the `*mouse-sensitive-constituents*` and `*mouse-sensitive-terminals*` variables as described in Section 14.5. For applications with

[1] Note, this is a generalization of the Penman functions `say` and `express`. It takes several additional keyword parameters and returns results that are not available from the corresponding Penman functions. Code relying on these features is not interchangeable with the Penman system.

more complex requirements than simply echoing the generated string, it will generally be the second value that is of more use.

If the systemic network connectivity for the requested language has not been established prior to the call to `say`, it will automatically be established for that language before commencing generation (by calling the function `reset-system-network`: Section 14.2.2).

If inquiry definitions have been loaded, but no defaults initialized prior to the call of `say`, default initialization will be automatically triggered before generation proceeds (by calling the function `ml-activate-defaults`: Section 14.2.3).

If the language selected by `:language` represents a switch of language from that previously used for generation, then the standard language switching actions will be triggered (see Section 7.11).

14.2 Bookkeeping functions

14.2.1 Switching languages

The command DEVELOPMENT: <Set Language> is realized by the Lisp function:

`switch-language &key :language :load-patches` *[function]*

This changes the current language to be `:language` and, if `:load-patches` is true, loads in any language specific patches for the new current language. This includes inquiry implementations, default orderings and punctuation. All are triggered by a function of the appropriate name being found in the resource directory of the specified language.

Note, however, that if an example is defined for a unique language variety, then this language takes precedence when generation of the example is attempted.

14.2.2 Establishing network connectivity

Prior to using linguistic resources for generation, the connectivity of the defined systemic needs to be checked and internal data structures representing that connectivity built up. KPML usually re-establishes connectivity prior to generation whenever new system definitions have been loaded. The function to set up connectivity is:

`reset-system-network &optional current-language` *[function]*

The `current-language` parameter, when set, restricts the resetting connectivity operation to the current language. Otherwise connectivity is established for all languages known to KPML.

14.2.3 Inquiry default initialization

The sequence of operations to be performed to activate the necessary defaults for a language is as follows. This can also be used if there is some suspicion that defaults are not being adequately set up automatically.

1. Load the standard default configuration for the resource set with `(load-properties` *:set-name*`)`.

2. Load the macro and default definitions for the resource set with `(load-spl-defaults-and-macros` *:set-name*`)`.

3. Activate those defaults and macros with `(ml-activate-defaults` *:language*`)` for each language present in the resource set for which generation is desired.

14.2.4 General initialization

The KPML function

`do-all-kpml-initializations` [*function*]

provides a convenient way of performing all initializations that are required without doing any generation. This could be used, for example, after loading and before generating when giving a demonstration.

14.3 Multilingual behaviour flags

The internal flags for controlling the behaviour of loading and saving operations are:

- `*loading-saving-profile*`: contains the objects that are effected during loading or saving operations (cf. Section 5.6.2).

- `*ml-saving-mode*` : should be either `:monolingual`, `:contrastive`, or `:multilingual` in order to parameterize the action of the saving functions in the way described in Section 5.9.1.

- `*ml-loading-mode*` : should be either `:monolingual`, `:contrastive`, or `:multilingual` in order to parameterize the action of the loading functions in the way described in Section 5.7.

14.4 Development tools

In this section some of the internal Lisp function calls for the multilingual operations supported by the KPML window interface are given. This permits their embedding in further code and their use when the window interface cannot, for some reason, be used. Unless otherwise noted, all functions and symbols are in the `Kpml` Lisp package.

The sequence of operations that will be described are as follows:

1. loading a resource set,

2. generating the example set,

3. modifying the resources,

4. saving the modified resource set.

In the immediately following example, we set out how one can load a set of resources, generate examples, use the example runner, and save out modified resources. This gives the minimal information for using the system. In the sections following, more details of each of the available functions is given, providing for more sophisticated use approaching that reachable from the window interface.

In this example, we presume that KPML has been installed, an appropriate set of resources are accessible (via the variable `user::*root-of-resources*`), and we want to generate examples in German. In this case, it is sufficient to type:

```
(in-package "KPML")
(load-linguistic-resources :german)
```

in order to load all the resources associated with the language variety German (including lexicons, domains, grammar, examples, etc.). If these resources include an example called '`Behrens3`', then the function call:

```
(say 'behrens3)
```

is sufficient to generate this example. All bookkeeping such as establishing defaults and network connectivity will be triggered automatically.

14.4.1 Linguistic Resource Loading Operations

A set of functions is provided for loading linguistic resources. With these functions one can either load an entire resource set or particular types of linguistic objects. The smallest granularity of concern is the grammatical region. The structuring of the loading functions can be envisioned thus:

```
load-linguistic-resources
    load-properties
    load-grammar
        load-region
              load-systems
              load-choosers
              load-inquiries
        load-default-orderings
        load-punctuation
        load-language-patches
    load-inquiry-implementations
    load-domains
    load-lexicons
    load-examples
    load-spl-defaults-and-macros
    load-kpml-lg-specific-patches
```

All functions take as first parameter the name of the linguistic resource set from which they want to load resources; e.g.:

```
(load-grammar :english)
```

The functions operating on regions (i.e., `load-region`, `load-systems`, `load-choosers`, and `load-inquiries`) take an obligatory second parameter that identifies the region of concern.

When the loading mode is contrastive, the single variety name must be replaced by a list of variety names.

Finally, all functions allow three further optional keyword parameters as follows:

`load-linguistic-resources`	*`variety-designation`* `&key :root-directory :merge :clear`	[*function*]

or

`load-region`	*`variety-designation region`* `&key :root-directory :merge :clear`	[*function*]

The functions thus load the designated objects of the set of resources for language(s) `variety-designation` (keyword or symbol) from the directory of the same name that is located under the specifed resource `root directory`. The remaining keywords have the following effects:

- `:clear` – when `nil` no resources are cleared;
- `:merge` – when `t` resources are loaded in merging mode (Section 5.7.2.2); clearing is disabled when this mode is selected.

The defaults are that resources are cleared and merging is not activated.

Following application of the `load-linguistic-resources` function, the current language is left set to the language of the last set of resources loaded.

14.4.2 Generating the example set

The following simple function call activates the example runner.

```
(example-runner Examples-list
                :data-file-name File)
```

This runs through the examples whose names are found in the `Examples-list` writing the results of generation in the file `File`.

Several more sophisticated uses of the example runner are possible. The full parameter description of the `example-runner` function is as follows.

`example-runner`	*`example-list`* `&key :data-file-stream :data-file-name :say-p :message-stream :check :compare :closeness`	[*function*]

The parameters have the following effects:

- `:data-file-stream` allows the specification of a stream directly for the results of the example run, instead of going via a file name.
- `:data-file-name` provides a file that the results stream will print to.
- `:say-p` indicates that examples are to be generated in *implemented mode* (the default: see Section 12.2.9).
- `:message-stream` provides the stream where example runner messages (such as the example currently being generated, warnings, errors, etc.) will appear.
- `:check` requires that the results of the example runner will be compared with the results of some former run of the example runner. Checking is disabled by default. When activated, the precise check made is determined by the subsequent two parameters.
- `:compare` specifies whether the example runner check is made with the value of the `:target` slot (the default) of the example records generated or with the previously `:generated` form.
- `:closeness` specifies whether a string compared during example checking has to be an `:exact` match, including spacing, or a `:loose` match (the default), where spacing is not significant (i.e., any sequence of blanks is equivalent to a single space).

When checking is activated, a summary of mismatches is given following execution of the example runner. This allows speedy recognition of whether some changes in a linguistic resource have deleteriously affected previous generation behaviour.

14.4.3 Modifying the resources

Since all the resource definition forms as described in Chapter 12 are also Lisp expressions, evaluating them in, for example, an Emacs buffer, or loading files containing them is sufficient to modify the loaded resources accordingly. Note that if the resource patching capability is activated (Section 11), then all evaluations/loading of systems, choosers, and inquiries successive to a call of *load-linguistic-resources* will be marked as patches.

Patching can be activated from Lisp by pushing the symbol `:resource-patches` onto the list `*loading-saving-profile*` and by setting the flag `*in-ml-region*` to `T`.

14.4.4 Saving the resources

A set of functions is provided for saving linguistic resources. With these functions one can either save an entire resource set or particular types of linguistic objects. The smallest granularity of concern is the grammatical region. The structuring of the saving functions can be envisioned thus:

```
save-linguistic-resources
```

```
save-properties
save-grammar
    save-region
          save-systems
          save-choosers
          save-inquiries
    save-default-orderings
    save-punctuation
    save-language-patches
save-lexicons
save-examples
save-spl-defaults-and-macros
```

This is largely the mirror image of the functions provided for loading, with the exception that information that is not represented in some KPML-specific, or systemic, form cannot be automatically saved. Thus there is no provision for saving inquiry implementations—since these are straight Lisp—and nor for the domain model definitions—since these are represented in LOOM. Finally, note that none of these functions performs any other actions on the directory to which they are saving resources. It is the user's responsibility when using these functions to ensure that new and old resources do not become mixed.

As with loading functions, all saving functions take as first parameter the name of the linguistic resource set from which they want to save resources; e.g.:

```
(save-grammar :english)
```

The functions operating on regions (i.e., `save-region`, `save-systems`, `save-choosers`, and `save-inquiries`) take an obligatory second parameter that identifies the region of concern.

When the saving mode is contrastive, the single variety name must be replaced by a list of variety names.

In addition, all saving functions may take an optional keyword parameter `:inheriting-from`. This permits the construction of new resource sets that are simply copies of the existing language definition specified as the `inheriting-from`, reconditionalized for the language given as first parameter. For example,

```
(save-linguistic-resources :french :inheriting-from :english)
```

creates a new resource set definition, identical to that for language variety English, but conditionalized for French. (See Section 5.9.3).

14.5 Using the mouseable structures for mousing and mark-up

When the `:full-structure` parameter to `say` (Section 14.1) is set, a list of pairs of generated strings and 'mouseable structures' is produced as second result. These mouseable structures can be used as the basis for mouse sensitive presentations of the string (as they are in the window interface) or for establishing hyper-text links, etc. Such structures are also stored into the `:structure` slot of example records (cf. Section 12.2.9). The value of this slot is actually a *list* of such structures, corresponding to the fact that multiple results could be generated from a single input specification (if, for example, the final ordering is not sufficiently well constrained to produce a single result).

14.5.1 The structure produced

The general form of a mouseable structure is as follows:

(PRINTABLE-CONSTITUENT

ID	*a unique label*
CONCEPT	*corresponding concept*
NODE-TYPE	*either* NIL *or* :terminal
ANNOTATION	*user defined*
FUNCTIONS	*list of grammatical functions*
SPELLING	*list of subconstituents*

The `spelling` slot's list of subconstituents is made up either of *strings*, indicating no further represented substructure, or further `printable-constituent` structures.

Each printable constituent node corresponds to some node in the grammatical structure generated (but not vice versa: see below). The `functions` slot contains the grammatical functions describing that node, and the `concept` slot contains the semantics (if any) associated with those functions. The `annotation` slot is intended for associating arbitrary user provided information with part of the generated linguistic result (Section 14.5.3).

An example of a complete mouseable structure is shown in Figure 14.1.[2]

14.5.2 Conditionalization of mouse sensitivity

The mouseability of the resulting generated strings can be further tuned by the user as follows. Two variables are provided that provide for conditionalization of mouse sensitivity.

- `*mouse-sensitive-constituents*` can contain a list of *systemic features*, i.e., features that occur in systemic networks. Whenever a constituent is generated possessing a feature on this list in its selection expression, then it will be made mouse sensitive.

- `*mouse-sensitive-terminals*` can contain a list of *grammatical functions*, i.e., the functional labels of elements of structure. This is used for conditionalizing the mouse sensitivity of *terminal elements* in the generated structure since these do not have any selection expression—normally because they are lexical elements directly inserted into structure rather than by being generated by a traversal through some systemic network.

[2]Each printable constituent object also has a unique identifier (under the slot `:id`); these have been ommited from the figure to save space.

```
(#S(PRINTABLE-CONSTITUENT
    CONCEPT LEAD-124525 NODE-TYPE NIL ANNOTATION NIL
    FUNCTIONS (SENTENCE)
    SPELLING (#S(PRINTABLE-CONSTITUENT
                 CONCEPT DIFFER-124541 NODE-TYPE NIL ANNOTATION NIL
                 FUNCTIONS (TOPICAL#1 CARRIER#1 SUBJECT#1)
                 SPELLING
                 (#S(PRINTABLE-CONSTITUENT
                       CONCEPT  NIL SPELLING ("The ")
                       NODE-TYPE :TERMINAL ANNOTATION NIL
                       FUNCTIONS (DEICTIC#2))
                  #S(PRINTABLE-CONSTITUENT
                       CONCEPT DIFFER-124541 SPELLING ("difference ")
                       NODE-TYPE :TERMINAL ANNOTATION NIL
                       FUNCTIONS (THING#2))))
                  #S(PRINTABLE-CONSTITUENT
                    CONCEPT ST295-124527-124550-124556 SPELLING ("has ")
                    NODE-TYPE :TERMINAL ANNOTATION NIL
                    FUNCTIONS (TEMPO1#1 TEMPO0#1 FINITE#1))
               #S(PRINTABLE-CONSTITUENT
                    CONCEPT  LEAD-124525 SPELLING ("led to ")
                    NODE-TYPE :TERMINAL ANNOTATION NIL
                    FUNCTIONS (VOICE#1 TEMPO1DEPENDENT#1 CIRCUMSTANCE#1
                               LEXVERB#1 PROCESS#1))
               #S(PRINTABLE-CONSTITUENT
                    CONCEPT BEHAVE-124542
                    SPELLING
                    (#S(PRINTABLE-CONSTITUENT
                          CONCEPT S1300-124543
                          SPELLING ("some ")
                          NODE-TYPE  :TERMINAL ANNOTATION NIL
                          FUNCTIONS  (DEICTIC#3))
                     #S(PRINTABLE-CONSTITUENT
                          CONCEPT SCHIZOID-124548
                          SPELLING
                          (#S(PRINTABLE-CONSTITUENT
                             CONCEPT  SCHIZOID-124548
                             SPELLING ("schizophrenic ")
                             NODE-TYPE :TERMINAL ANNOTATION NIL
                             FUNCTIONS (QUALITY#4)))
                          NODE-TYPE NIL ANNOTATION NIL
                          FUNCTIONS (STATUS#3))
                     #S(PRINTABLE-CONSTITUENT
                          CONCEPT BEHAVE-124542
                          SPELLING ("behaviour ")
                          NODE-TYPE :TERMINAL ANNOTATION NIL
                          FUNCTIONS (THING#3)))
                   NODE-TYPE NIL ANNOTATION NIL
                   FUNCTIONS (DIRECTCOMPLEMENT#1 ATTRIBUTE#1))
       ".")))
```

Figure 14.1: Example of mouseable structure for the sentence: ‘The difference has lead to some schizophrenic behavior’

In addition, both variables may take the values `:all` or `:none`. Setting the former variable to `:all` means that all non-terminal constituents, regardless of their selection expressions, will be made mouse sensitive; setting the latter variable to `:all` means that all terminals, regardless of which grammatical function that are realizing, will be made mouse sensitive. The `:none` options are in both cases equivalent to setting the variables to the null list.

Since the structures supporting mouse sensitivity are passed on to the user or application program, they can form the basis for further mouse-driven options that an application can offer. For this purpose, it may then be preferred to conditionalize the mouse sensitivity beforehand so that only pruned structures need be processed by the application. The default setting for linguistic resource development is that all constituents and terminals are made mouse sensitive.

14.5.3 Specifying additional links in the SPL: annotations

KPML provides the SPL keyword `:annotation` for specifying additional links between the generated strings and user-given information. The value of the `:annotation` keyword for a given SPL term is placed in the annotation slots of the constituents of the mouseable structure corresponding to the realization of that SPL term. This makes it straightforward, for example, to interpret the generated strings as components of a hypertext, where the annotations specify hyperlink addresses or URLs: the application need only to traverse the generated mouseable structure (Figure 14.1) and insert appropriate markup when a hyperlink annotation is found on some node.

14.6 Window startup functions

It is possible for the user to start different components of the window interface directly from Lisp. This only makes sense, of course, when KPML is being used with the window interface included.

The individual Lisp functions for starting up the main new-style KPML interface windows are as follows. Each takes an optional parameter which, when set (`T`), causes any existing instances of the relevant window to be replaced. If unset, a new window is created only when there is no such window already existing. The default is always that no replacement occurs. Only the first two functions would normally be of relevance for a user: particularly for restarting interface windows if they become broken.

`kpml-i::startup-resource-development-frame` *&optional reset* [*function*]

Starts up the resource development window as described in Chapter 7.

`kpml-i::startup-resource-inspector-frame` *&optional reset* [*function*]

Starts up the resource inspector window as described in Chapter 6.

The remaining functions would only be of use for further interface extensions or tighter integration into applications.

`kpml-i::startup-cumulative-gh-frame` *&optional reset* [*function*]

Starts up a cumulative generation history frame as described in Section 7.5.5.

`kpml-i::startup-fat-frame` *&optional reset* [*function*]

Starts up a function association table display window as described in Section 7.5.2.13.

`kpml-i::startup-generation-history-frame` *&optional reset* [*function*]

Starts up a generation history window as described in Section 7.5.2.2.

`kpml-i::startup-a-traversal-frame` [*function*]

Brings up a dynamic network traversal display window as described in Section 7.7.2. Successive calls to the function bring up new windows.

`kpml-i::startup-a-results-window` [*function*]

Starts up a results display window; this contains will show the last generated string if any. Display respects the normal results of generation flags as described in Section 7.10. Successive calls to the function bring up new windows.

Finally, the following function starts up the main root KPML window only if one does not already exist.

`kpml-i::startup-resource-management-frame` [*function*]

Forcing the creation of a new window can be done using the standard user startup function (`kpml-i::startup`: Section 5.2).

14.7 Accessing Linguistic object slots

This section summarizes the Lisp accessor functions that return values of the various slots of the internal data structures used for representing linguistic objects in KPML. Use of these functions is never necessary for normal use of KPML for tactical generation. They might however be useful for various extensions to the basic KPML functionality.[3] All of these functions are in the `KPML` Lisp package.

14.7.1 Systems

The main accessor functions for systems are as follows. Each takes the name of a system as an obligatory parameter and an optional specification of the language of the system to be accessed. The default language is always the current language (as maintained as the value of the variable `curlan`).

`fetch-system-chooser` *system-name &optional ls* [*function*]

This returns the *name* of the chooser associated with the system.

`fetch-system-inputs` *system-name &optional ls* [*function*]

[3]This is not an exhaustive list. Some of the internal data structure slots reflect information that is either used internally or is no longer used.

This returns the entry condition of the system in the definition format described in Section 12.2.4.

`fetch-system-outputs` *system-name &optional ls* [*function*]

This returns the output features of the system in the definition format described in Section 12.2.4.

`fetch-system-region` *system-name &optional ls* [*function*]

This returns the region of the system.

14.7.2 Choosers

A single accessor function for choosers is provided.

`fetch-chooser-info` *chooser-name slot-name &optional ls* [*function*]

The language required is specified as for systems; the default is always the current language (the value of `curlan`). The particular information returned is specified by the `slot-name` parameter. The slot-name is a single symbol (in the `KPML` Lisp package). The most important slot is `definition`. This causes the body of the chooser definition to be returned for the specified language in the format described in Section 12.2.6.

14.7.3 Inquiries

A single accessor function for inquiries is provided.

`fetch-inquiry-info` *inquiry-name slot-name &optional ls* [*function*]

The language required is specified as for systems; the default is always the current language (the value of `curlan`). The particular information returned is specified by the `slot-name` parameter. The slot-name is a single symbol (in the `KPML` Lisp package) selected from the following list. For further details of the functions of all these constructs, see Section 12.2.7.

- `operatorcode`: returns the name (if any) of the Lisp function that is the inquiry's implementation.
- `preselectionguidance`: returns any preselection guidance that the inquiry currently has.
- `trivialdefault`: return the general trivial default for the inquiry.
- `activedefault`: return the *currently active* default for the inquiry.

14.7.4 Examples

A single accessor function for examples is provided.

`fetch-example-info` *example-name slot-name &optional ls* [*function*]

The language required is specified as for systems; the default is always the current language (the value of `curlan`). The particular information returned is specified by the `slot-name` parameter. The slot-name is a single symbol (in the `KPML` Lisp package) selected from the following list. For further details of the functions of all these constructs, see Section 12.2.9.

- `targetform`: the string(s) that the the example should generate.
- `generatedform`: the string(s) that were actually generated last time the example was run.
- `logicalform`: the SPL for the example.
- `structure`: the generated structure (this is the structuer that is returned by the function `say` when `:full-structure` is requested (cf. Section 14.5).
- `selectionexpressions`: the list of selection expressions for all constituents generated from this example. The list has the format described for example runner results in Section 10.2.9.5.

Chapter 15

Faster Generation

KPML maintains very extensive information during generation which is used for the many options provided for inspecting the process and results of generation. It also provides for the interpretation of multilingual resources throughout. These features result in a certain run-time overhead which reduces the speed of generation. This is usually not a problem when debugging and maintaining resources. However, if the resources are to be used simply for generation and are considered, for some purpose, sufficiently debugged, then it can be desirable to have as fast a generation process as possible. The ideal solution here would be to have a dedicated kernel generator for systemic resources that takes the basic generation algorithm (as described, for example, in Matthiessen & Bateman 1991, pp100–109), implementing this in a run-time efficient manner and programming language. All of the debugging and maintenance overheads could then be spared. Unfortunately, such a kernel generator is not yet available.

As an interim solution, however, the methods described in this chapter can be adopted. These significantly increase the speed of generation with KPML at the cost of partially disabling the debugging facilities and, for one method, fully disabling multilinguality. On faster machines short texts of 10-15 sentences can be generated in a few seconds: generally fast enough for demonstration purposes.

Several different methods can be combined to reduce generation time. These are detailed here since they have differing side-effects, some of which may be important for particular applications. The main methods are:[1]

- deactivation of multilinguality,
- knowledge-base package reduction,
- compilation of the inquiry implementations.

Having the window interface active also brings a small run-time overhead that can be avoided by not bringing the interface up.

The approximate improvements in generation time that these methods achieve are indicated

[1]The knowledge-base package reduction methods, as well as several internal speed-ups, were worked out by John Wilkinson (University of Waterloo, Canada).

`Reuters1`:

"The European electronics industry has made a lot of noise in public about keeping Europe safe from Japanese competitors, but in private they are saying that if you can't beat them, you should join them."

KPML configuration			Machines		
MO	PR	CI	SB3	S10-64	S20-320
−	−	−	13.4s	6.3s	3.2s
+	−	−	11.5s	5.3s	2.7s
−	+	−	10.5s	4.9s	2.2s
+	+	−	8.3s	3.8s	1.8s
−	−	+	9.9s	4.9s	2.8s
+	−	+	7.8s	3.9s	2.0s
−	+	+	7.1s	3.5s	1.6s
+	+	+	5.0s	2.4s	1.1s

Key. MO: KPML running strictly monolingually; PR: KPML running with package reduction; CI: KPML running with compiled inquiries.

Machines. SB3: Tadpole Sparcbook 3 (32Mb RAM; ACL4.3); S10-64: Sun Sparcstation 10 (64Mb RAM; ACL4.3); S20-320: Sun Sparcstation 20 (320Mb RAM; ACL4.3).

Table 15.1: Timings for differently configured KPML generation

by example in Table 15.1. This table shows the average generation time[2] on various machines for the sentence `Reuters1` from the ISI Reuters example set for the Nigel grammar. The particular quirks and side-effects of the speed-up methods are described in the following sections.

A general speed-up can also be achieved by compiling KPML with non-default values for the compiler flags of the Lisp system used: for example, by setting the speed flag to 3. When creating a generation server or demo system, for example, where the resources used are fully debugged, compilation can be redone with the compiler flags set according to:[3]

```
(proclaim '(optimize (speed 3) (safety 1) (space 0) (debug 0)))
```

All timings shown in Table 15.1 were made with the default options (safety: 1; space 1; speed 1; debug 2). An example of the speed-up possible for the REUTERS1 example sentence is from 13.5s (full monolingual KPML running on a Tadpole Sparcbook 3, 32MB RAM with ACL4.3) to 12.2s under the same configuration but with the non-default compiler flag settings. Compiling the inquiry implementation with speed at 3 then brings the generation time down further to around 9s without any loss of multilingual functionality. On significantly faster machines, the

[2] This is the time excluding, for example, any garbage collection, KPML once-only set up activities (such as establishing network connectivity), and Loom classifications.

[3] Franz strongly recommend that `safety` never be set to zero for their Allegro Common Lisp.

time overheads for full multilingual generation are now often acceptable without needing to trade-off on functionality.

Similarly, the German example BEHRENS4, which produces the two strings:

> 1890 studierte er bei Kotschenreiter in Muenchen und war 1893 ein Mitbegruender der Muenchner Sezession.
> 1890 studierte er in Muenchen bei Kotschenreiter und war 1893 ein Mitbegruender der Muenchner Sezession.
> (*In 1890 he studied with Kotschenreiter in Munich and in 1893 was a co-founder of the Munich Secession*)

is speeded up under the same change in configuration details from 6.25s to 5.25s, simply by changing the speed compiler flag.

15.1 Strictly Monolingual Generation

In order for this method to be applied, KPML must be configured for a single language and the resources for that language must be loaded. This can also be achieved by setting the variable `all_languages` to a list containing just the desired language, e.g., `(:dutch)`. This establishes an internal representation where most of the possible language conditionalizations do not occur. Then, setting the flag `*rigidly-monolingual*` to a non-nil value (e.g., `T`) will disable multilingual conditionalization interpretation. This will, of course, fail gracelessly if attempted with resources containing multilingual conditionalizations, so it is essential that the above configuration step be carried out.

15.2 Knowledge base package reduction

The KPML access function for checking knowledge base subsumption relationships (`kb-superp`) that is provided for Loom allows knowledge base concepts to reside in any Lisp package (as long as the function knows which). It's definition includes a considerable number of calls to the Lisp function `intern` which is fairly slow. This can be avoided by placing all knowledge base concepts in a single package. Further speed-ups can be achieved by simplifying the kind of concepts that are in fact sought; John Wilkinson provides the following comment in his speed-up code:

> "using evaluate-identifier instead of find-concept or find-relation saves considerable time by not checking for Loom extended identifiers (identifiers which contain the context name and concept name, separated by a '^'). Some users may desire to use this feature, so perhaps a parameter should be included which is checked initially to determine if find-concept should be used instead."

At present no such parameter is provided and so users placing their domain concepts in various packages should probably inspect their re-definitions of `kb-superp` to see how much of this speed-up method can be applied in their bown cases.

This kind of speedup can be installed by calling the function `kb-package-reduction-speedup`. Note that this is a *destructive* operation and it is not then possible to return to a non-speeded-up configuration without restarting KPML. Because of this, the function typically asks the user for confirmation before proceeding; this interaction can be suppressed by providing the value `T` to the keyword parameter `:force-p`.

The basic knowledge base package reduction method, which places all known Loom concept and relation symbols in the Kpml package, can be activated by calling the function `kpml::update-kb-package-reduction`. This function should be called whenever new concepts/relations in differing packages have been loaded. Note that this function does not exist unless in package reduction mode.

Clearly, if LOOM is not being used in a particular KPML configuration, then this speedup method is not applicable.

15.3 Compilation of inquiry implementations

This is the simplest method: all inquiry implementations used should simply be compiled in the usual manner and then loaded into the Lisp environment directly from a Lisp listener. Note that KPML never loads compiled inquiry implementations itself, since this makes the source definitions of the inquiries difficult to inspect.

The sc kpml Lisp function `load-compiled-inquiry-implementations` will, however, load compiled inquiry implementations for a single language variety (compiling the inquiry implementation files if necessary).

`load-compiled-inquiry-implementations` *`variety-designation`* `&key :root-directory` [*function*]

Chapter 16

Using templates in generation

It is traditional to distinguish between 'linguistically'-based generation techniques that rely on more or less general linguistic descriptions of some language and 'template'-based generation models. It is commonly argued that template-based generation is necessary for real applications. When viewed from within a suitably broad linguistic perspective, the division between template and non-template generation is artificial: both can be considered 'linguistic' when linguistic *instances* (realization patterns) can be specified alongside linguistic potential.

Whether a generation system employs templates or not should be decided flexibly according to the particular needs of the texts being generated: some text are more restricted according to 'pre-generated' patterns than others. It is not realistic to always generate from scratch all parts of a required text. Any real text will usually contain parts that have been written previously, or are fixed externally (by style, convention, or legislation). A full text generation account must therefore naturally include such a possibilities, although the most appropriate form for such a facility still requires further research.

KPML provides a basic template-generation functionality. This is offered on an experimental basis since it is clear that template generation has several disadvantages—especially when used for multilingual generation; further experience with the use of templates is necessary.

Templates are used by means of a further extension to SPL. They are introduced by the new pseudo-semantic type `template` and the SPL keyword `:pattern`. The format of the pattern is either a string, which is then the sole body of the template, or a list. The elements of the list are either strings or symbols representing SPL variables. Slots may be thus included in the template for further full generation. A simple example is the following:

```
((s / template
    :pattern ("The first of the " o1 " always sails to " o2))
 (o1 / ship :singularity-q nonsingular :multiplicity-q multiple)
 (o2 / port))
```

This involves a simple pattern "The first of the ... always sails to ...". With a simple domain for ships and ports, the Nigel grammar for English, and the standard inquiry defaults loaded, this SPL generates the string *The first of the ships always sails to a port.*

The template facility may be freely used recursively with combinations of embedded templates and items generated normally. The following example SPL takes several liberties in order to generate the string *The first of the ships always sails to a port to the west of an absolutely huge island.*

```
((s / template
    :pattern ("The first of the " o1 " always sails to " o2))
 (o1 / ship :singularity-q nonsingular :multiplicity-q multiple)
 (o2 / template :pattern ("a port to the west of" o3))
 (o3 / island
     :size-property-ascription
         (o4 / template :pattern ("absolutely huge"))))
```

Perhaps noteworthy here is the fact that the patterns become a part of the generated structure prior to final graphological processing and so can effect preceding and following elements of the structure as specified in the string cleaning up rules (see Section 12.2.2.5).

When the structure of sentences involving templates is graphed with the KPML structure inspection commands, the elements of patterns are shown as pseudo-grammatical functions. String elements received function names of the form **TEMPLATE-SEGM**; SPL variables receive names of the form **TEMPLATE-SLOT**.[1] Figure 16.1 shows the graphed structure for the example preceding.[2] Constituents created by template patterns naturally have no selection expressions.

Finally, Figure 16.2 is an example that combines multilingual conditionalization, annotations and templates in order to generate phrases of the form “Host X is unreachable” and the corresponding German “Host X ist nicht erreichbar”. The expression for X is in both cases generated normally and may be presented as a hyperlink (see Section 14.5.3).

Implementation limitation: It is not currently possible to use templates that are both associated with *non-embedded* SPL variables and which are the values of SPL slots. That is, the following SPL:

```
((s / sail :actor d1 :destination p)
 (d1 / template :pattern  "My favorite ship" )
 (p / template :pattern "my favorite port"))
```

will not have the result that might be expected. However, since it is always possible to express the same desired effect by embedding the relevant SPL terms at their point of use, this limitation is maintained at this time. Thus, the following SPL:

```
(s / sail
   :actor (d1 / template :pattern  "My favorite ship" )
   :destination (p / template :pattern "my favorite port"))
```

does generate the expected *My favorite ship sails to my favorite port.*

[1] These names are strings forming the values of the global variables `*template-segment-name*` and `*template-slot-name*` respectively. They can therefore be changed should this be required for some obscure reason.

[2] Note that, as usual, the structure graph shows the selected lexemes *before* some of the final string operations are performed: cf. Appendix D. This is why “a” appears as the indefinite article in the graph although “an” appears in the final string.

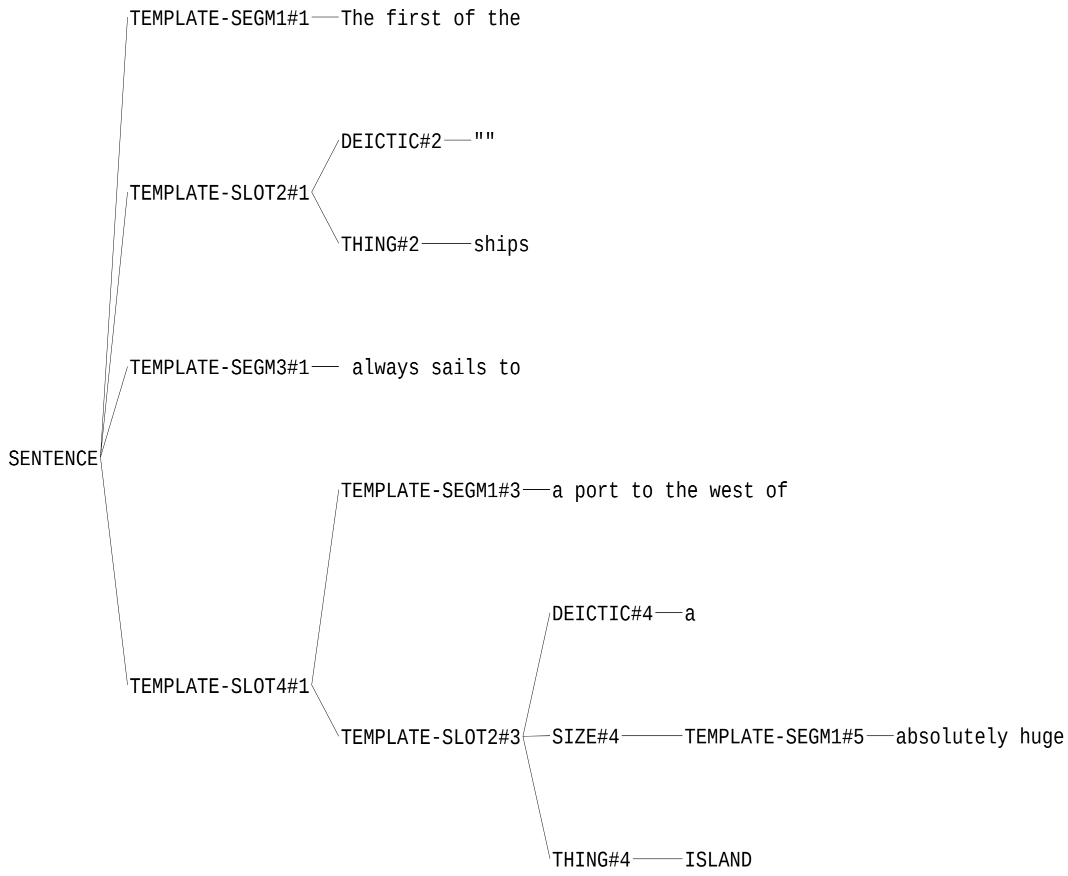


Figure 16.1: Graphed template structure

```
((e / template
      :pattern :english ("Host " actor " is unreachable")
               :german  ("Host " actor " ist nicht erreichbar"))
 (actor / host ...
      :annotate "http://www...."))
```

Figure 16.2: Example multilingual template generation specification

Chapter 17

Establishing and using a generation server

When used with Allegro Common Lisp (version 4.2 and newer under Unix), KPML includes basic methods for creating a generation server that can accept input specifications from other processes and return the generated string to those processes. The basic functions are described here, although it is still likely that particular applications of these methods will need to be tailored individually.[1]

This chapter describes the basic method for creating a KPML generation server, basic methods of for creating a Lisp KPML client, and an example usage of such a client: providing semantic generation as a World Wide Web server.

17.1 Creating a KPML generation server

Starting a KPML server is done by configuring and starting KPML as required, and then issuing the function call:

```
(kpml::start-kpml-server)
```

This establishes a connection to a Unix port: the host and the port number are left in a file identified in the variable `kpml::*com-file*`. This takes by default the value of the variable `kpml::*default-com-file*`, initially set up as the file:

```
<user-home-directory>/KpmlCom.tmp
```

This communication file is deleted when the KPML server is closed; the existence of the file can therefore be used as a test as to whether a KPML server is running or not.

Note that for a server it will normally be the case that the window interface is not loaded or started (see the relevant installation steps in Chapter 3) , and that some set of the speedups

[1] A CORBA-compliant protocol is being considered.

described in Chapter 15 will have been activated. Otherwise the server will have slower than necessary response time.

The server start-up function takes an addition optional parameter which, if set (`T`), initiates logging of the server's operations. The file name is created by appending the date and time to the string held in the variable `kpml::*kpml-log-file*`. This is initially by default the string `"/tmp/kpml-log"`. An example is therefore the following: `"/tmp/kpml-log-19960811-140247"`.

It is also possible to create KPML Lisp images which are specifically for acting as servers. This is done by a call to the Lisp function:

`kpml::make-kpml-server-image` *`server-name`* [*function*]

The function call creates a Lisp image containing the current generation functionality. When started fresh from Unix, the resulting image will automatically start up a KPML server that is ready to process client requests. The server image as called from Unix takes two optional command line parameters:

- `-f` introduces a file name that is to be used as the communication file identifying the server host and port number (see above); this simply sets the value of the variable `kpml::*com-file*`;
- `-l` specifies that logging should proceed; possible values of the parameter are `T` (indicating logging should proceed to the default place: see above) or a file name prefix to be used as the value of `kpml::*kpml-log-file*` (see above).

Thus, as an example, after issuing from Lisp the function call:

```
(kpml::make-kpml-server-image "/tmp/kpml-server")
```

It is then possible to give as a Unix command commands such as:

1. `kpml-server`
2. `kpml-server -f /tmp/ComFile`
3. `kpml-server -f /tmp/ComFile -l T`
4. `kpml-server -l /home/fred/my-kpml-log`

Note that the server image will not load any additional patches or site/user-specific information: it is an exact copy of the functionality of KPML at the point when the image is created and is not subsequently altered in any way.

17.2 Creating a KPML client from Lisp

This section describes how to establish a KPML client from Allegro Lisp. Clients from other types of process can be defined in the individual ways each programming environment allows for accessing Unix ports. The Lisp KPML client is created by loading into a running Lisp image the file:

<KPML root directory>`/PROCESSES/server/client.lisp`

No other KPML-specific files are necessary.

The most basic way of then connecting the client to a KPML server is by issuing the call:

```
(kpml::start-kpml-client)
```

This returns a stream to the port identified in the global variable `kpml::*com-file*`.

Information can then be sent to this stream using `ipc::send-to-socket`—a function of two arguments, the item to be sent and the stream. The function returns the value returned from the server.

SPL specifications can be sent to the KPML server with the function call: `(kpml::server-say-string <SPL> stream)`. This returns the string generated by the server in response to the SPL or, if generation failed for some reason, the string "...".

For more sophisticated use of the KPML server, the following form is provided.

`kpml:with-server-access` `(&key :stream :com-file :language :close-stream :terminals :constituents) &rest` ***body*** [*macro*]

This provides a program body within which various KPML server-specific variables are bound, including variables determining the language in which the server is to generate, and the degree of structure preserved in the resulting strings generated (cf. Section 14.5). The form normally starts a connection to the KPML server identified by the `com-file` (which defaults to the default communication file described above) and closes that connection when the form is finished. The keyword parameters `:terminals` and `:constituents` allow values to be set for the global KPML server variables `*mouse-sensitive-terminals*` and `*mouse-sensitive-constituents*` (Section 14.5.2) respectively.

The `:stream` keyword parameter can be used to maintain a single server session. A new server session is only started when the `:stream` parameter is `nil`; otherwise, this parameter should contain a stream for a KPML server. Such a stream can be obtained either by the function `kpml::start-kpml-client` or by setting the `:close-stream` parameter for the `with-server-access` form to `nil`. Whenever this latter parameter is not set, the form as a whole returns the server stream used within its body—regardless of this was for a newly created server connection or was passed in at the outset as the value of `:stream`.

Within the scope of the `with-server-access` form, server-based generation can be triggered by the function `kpml:say`. This takes one obligatory argument, an SPL form from which to generate or an example name. The result of the function call is the full presentation structure described in Section 14.5. The declaration of the function is as follows:

`kpml:say` ***spl*** `&key :stream :com-file :language :stream :m-terms :m-consts` [*function*]

The default values of the keyword parameters are taken from the values established by the `with-server-access` form. Thus, another way of sending an SPL to a KPML server and doing something with the result is the following:

```
(kpml:with-server-access ()
  (print
    (kpml:say <SPL-form>)))
```

Normally there is more to be done within the scope of the access to the server than simply printing out the structure of course; an example is given in the following section.

The current language of the server and the presentation structures can be further altered within the body of the `with-server-access` form by the following functions:

`kpml:set-mouse-sensitive-constituents` *constituent-list* [*function*]

`kpml:set-mouse-sensitive-terminals` *terminal-list* [*function*]

`kpml:set-language` *language-variety* [*function*]

It is also possible to interrogate the server concerning the examples it currently has loaded and the languages for which the server is configured. The first is retrieved by means of the function call: `(kpml:get-kpml-example-list)`; this returns the full example list. The second is retrieved similarly by the function call: `kpml:get-language-range`.

Finally, the following function provides client access to the full example records produced during generation (cf. Section 12.2.9). This is probably only useful for clients that also attempt to perform some resource development and/or maintenance; for normal applications the results of the `kpml:say` function should be sufficient.

`kpml::get-full-example-structure` *spl* `&key :language :m-consts :m-terms :stream` [*function*]

The result returned is the example record structure.

17.3 An example of a KPML Lisp client: a WWW-KPML server

This section gives a simple example of using the facilities for creating Lisp clients for a KPML server. The example is artificially simple, but nevertheless serves as an illustration of certain techniques that could be applied more generally. The code shown here makes available a World-Wide Web-based server for generating examples from resource sets loaded into KPML. The user submits a form selecting some set of examples to be generated, and this causes the corresponding generated strings to be displayed. The WWW-facilities are provided by the MIT Common Lisp hypermedia server (CL-HTTP: Mallery 1994).

We assume that a multilingual KPML server has been established and is running. We call the server `kpml-server`.

We assume further that a Lisp image including the CL-HTTP server is available. Rather than install the entire CL-HTTP system on top of KPML, or the entire KPML system on top of CL-HTTP, we make use of the KPML server-client functionality in order to create a small KPML client that can be loaded into the CL-HTTP server. The resulting program (which we will call `www-kpml`) provides the service of generating strings from selected examples to the web, but

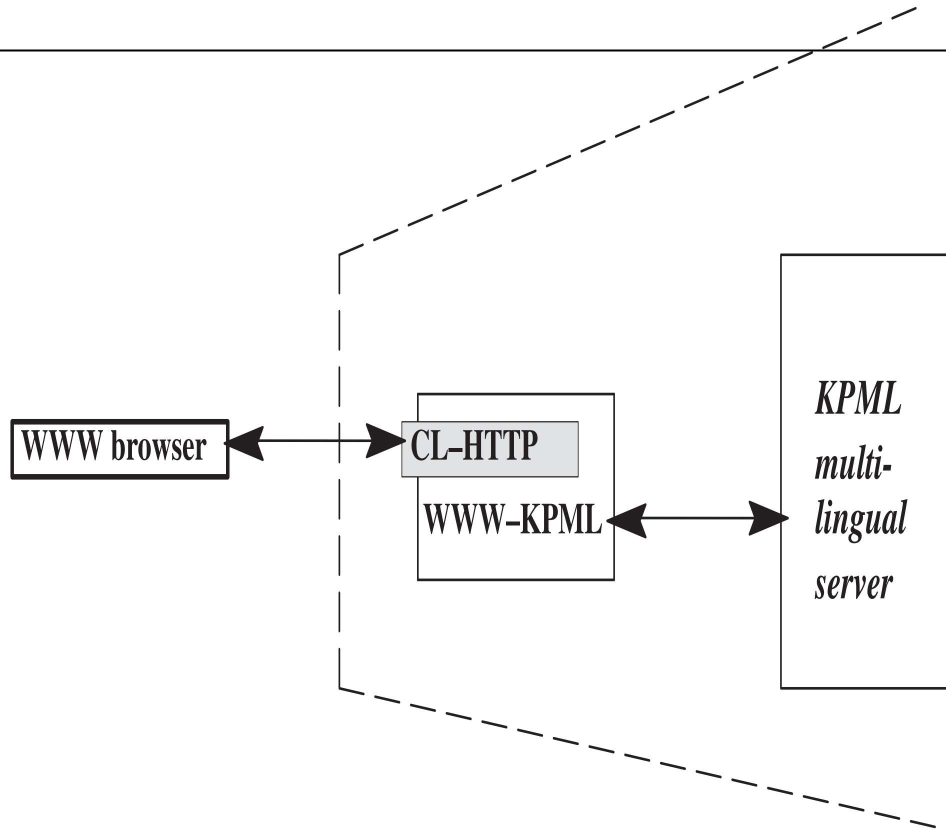


Figure 17.1: Program configuration of the example WWW server

does so by sending requests to the separate KPML server. This configuration is shown graphically in Figure 17.1; the file <KPML root directory>`/PROCESSES/server/www-kpml.lisp` contains the code described below.

The precise functionality of the provided web server is somewhat trivially to accept selections from the available SPL examples for a language (in this case German) and to present the generated string back to the user. The main code is accordingly straightforward and consists of three components: (i) a method that creates the HTML form which accepts user requests for generation, (ii) a response method that is activated whenever the user submits the generation form, and (iii) a declaration to the web server of where the generation form is to be located—i.e., which URL the generation form is to have.[2]

The first component is the most complicated of the three and is as follows. Most of the content of the method is concerned with setting up the HTML appropriately for the displayed webpage. The KPML-specific parts of the definition are shown in boldface.

```
(defmethod COMPUTE-GENERATION-REQUEST ((url url:http-form) stream)
  (with-successful-response
      (stream :html :expires (url:expiration-universal-time url))
    (html:with-html-document (:stream stream)
      (html:with-document-preamble (:stream stream)
        (html:declare-base-reference url :stream stream)
```

[2]The code is adapted directly from John Mallery's CL-HTTP dynamic forms examples. The example shown was run with CL-HTTP version 58.12, ACL4.2 and Netscape 3.0.

```
        (html:declare-title "KPML generation server" :stream stream))
      (kpml:with-server-access (:language :german)
(html:with-document-body (:stream stream)
  (html:with-section-heading ("KPML generation server" :stream stream)
    (image-line :stream stream)
    ;;
    ;; Make a form containing the available examples as a menu...
    ;;
    (html:with-fillout-form (:post url :stream stream)
      (html:with-paragraph (:stream stream)
(with-rendition (:bold :stream stream)
  (fresh-line stream)
  (write-string "SPL examples: " stream))
                (html:accept-input
 'html:select-choices
 "CHOICES"
 :choices (mapcar #'first (kpml:get-kpml-example-list))
 :default
 *generation-requests* :sequence-p t :stream stream))
      ;;
      ;; When there are examples to generate, do so...
      ;;
      (html:with-paragraph (:stream stream)
  (loop for example in *generation-requests*
      do
(html:with-paragraph (:stream stream)
  (format stream "~A: " example)
  (with-rendition (:bold :stream stream)
    (write-string (caar
  (kpml:say
   (intern
    (string-upcase example)
    "KPML")))
  stream)))))
      (submit-and-reset-buttons stream))
    (image-line :stream stream)
    (cl-http-signature stream)))))))
```

The second component simply picks up the example selections that have been made by the user and regenerates the web page of the original form:

```
(defmethod RESPOND-TO-GENERATION-REQUEST
    ((url url:http-form) stream query-alist)
  (bind-query-values
   (choices)
   (url query-alist)
   (let ((*generation-requests* choices))
     ;; generate another version of the form with the new values.
     (compute-generation-request url stream))))
```

Finally, the generation form and the response method are declared to the web server and allocated a URL.

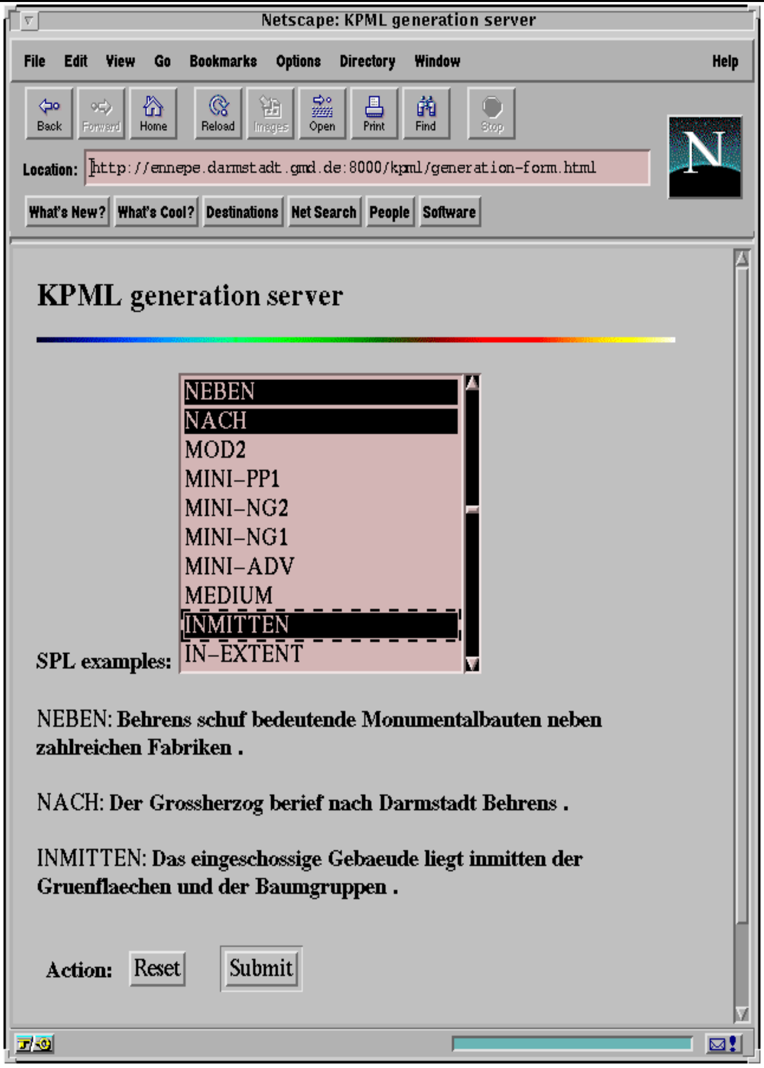


Figure 17.2: Example generation server in use

```
(export-url #u"/kpml/generation-form.html"
            :html-computed-form
            :form-function #'compute-generation-request
            :expiration '(:no-expiration-header)
            :response-function #'respond-to-generation-request
            :keywords '(:kpml :generation :demo)
            :documentation "KPML example WWW server demo.")
```

An example of the generation server in use is given in Figure 17.2. This shows the state of the KPML server web page after the user has selected some examples and clicked on *submit*. The strings following the example names (NEBEN, etc.) were generated by the KPML server and then returned to the WWW page.

Clearly, very much more complicated (and useful!) servers could be readily constructed with the client functions defined above.

Bibliography

Bateman, J. A. (1991), Language as constraint and language as resource: a convergence of metaphors in systemic-functional grammar, Technical report, Gesellschaft für Mathematik und Datenverarbeitung – Institut für Integrierte Publikations- und Informationssysteme, Darmstadt, Germany. Written version of paper presented at the International Workshop on Constraint-based Formalisms for Natural Language Generation, November 27-30, 1990, Bad Teinach.

Bateman, J. A., Emele, M. & Momma, S. (1992), The nondirectional representation of Systemic Functional Grammars and Semantics as Typed Feature Structures, *in* 'Proceedings of COLING-92', Nantes, France.

Bateman, J. A., Matthiessen, C. M., Nanri, K. & Zeng, L. (1991*a*), Multilingual text generation: an architecture based on functional typology, *in* 'International Conference on Current Issues in Computational Linguistics', Penang, Malaysia. Also available as technical report of the department of Linguistics, University of Sydney.

Bateman, J. A., Matthiessen, C. M., Nanri, K. & Zeng, L. (1991*b*), The re-use of linguistic resources across languages in multilingual generation components, *in* 'Proceedings of the 1991 International Joint Conference on Artificial Intelligence, Sydney, Australia', Vol. 2, Morgan Kaufmann Publishers, pp. 966 – 971.

Bateman, J. A., Matthiessen, C. M. & Zeng, L. (1996), A general architecture of multilingual resources for natural language processing, Technical report, Institute for Integrated Publication and Information Systems (German National Research Centre for Information Technology, Darmstadt) and Macquarie University, Sydney.

Bateman, J. A. & Teich, E. (1995), 'Selective information presentation in an integrated publication system: an application of genre-driven text generation', *Information Processing and Management* **31**(5), 753–768. (Special Issue on Summarizing Text).

Brachman, R. J. & Schmolze, J. (1985), 'An overview of the KL-ONE knowledge representation system', *Cognitive Science* **9**(2).

Brew, C. (1991), 'Systemic Classification and its Efficiency', *Computational Linguistics* **17**(4), 375 – 408.

Carpenter, B. (1992), *The Logic of Typed Feature Structures*, Cambridge University Press, Cambridge, England.

Degand, L. (1993), Dutch grammar documentation, Technical report, GMD/Institut für Integrierte Publikations- und Informationssysteme, Darmstadt, Germany.

Devlin, K. (1990), Infons and types in an information-based logic, *in* R. Cooper, K. Mukai & J. Perry, eds, 'Situation Theory and its applications', Vol. I, CSLI: Center for the Study of Language and Information, Stanford University, California, pp. 79 – 96. CSLI Lecture Notes Number 22.

Emele, M., Heid, U., Momma, S. & Zajac, R. (1992), 'Interactions between linguistic constraints: Procedural vs. declarative approaches', *Machine Translation* **6**(1). (Special edition on the role of text generation in MT).

Finkler, W. & Neumann, G. (1988), MORPHIX: A fast realization of a classification-based approach to morphology, *in* 'Proceedings of the 4th. ÖGAI: Wiener Workshop Wissensbasierte Sprachverarbeitung', number 176 *in* 'Informatik Fachberichte', Springer Verlag, Berlin.

Götz, T. & Meurers, W. (1995), Compiling HPSG type constraints into definite clause programs, *in* 'Proceedings of the 33rd. Annual Meeting of the Association for Computational Linguistics'.

Halliday, M. A. (1961), 'Categories of the theory of grammar', *Word* **17**, 241–292. Reprinted in abbreviated form in Halliday (1976) edited by Gunther R. Kress, pp 52-72.

Halliday, M. A. (1976), The English verbal group, *in* G. R. Kress, ed., 'Halliday: system and function in language', Oxford University Press, London.

Halliday, M. A. (1978), *Language as social semiotic*, Edward Arnold, London.

Halliday, M. A. (1985), *An Introduction to Functional Grammar*, Edward Arnold, London.

Henschel, R. (1992), A proposal for merging the english and german upper models, Technical report, GMD/Institut für Integrierte Publikations- und Informationssysteme, Darmstadt, West Germany, Darmstadt, Germany.

Henschel, R. (1994), Declarative representation and processing of systemic grammars, *in* C. Martin-Vide, ed., 'Current Issues in Mathematical Linguistics', Elsevier Science Publisher B.V., Amsterdam, pp. 363–371.

Henschel, R. (1995), Traversing the Labyrinth of Feature Logics for a Declarative Implementation of Large Scale Systemic Grammars, *in* Suresh Manandhar, ed., 'Proceedings of the CLNLP 95'. April 1995, South Queensferry.

Kameyama, M., Ochitani, R. & Peters, S. (1991), Resolving translation mismatches with information flow, *in* 'Annual Meeting of the Assocation of Computational Linguistics', Association of Computational Linguistics, Berkeley, California, pp. 193–200.

Kasper, R. T. (1987), Systemic grammar and functional unification grammar, *in* J. D. Benson & W. S. Greaves, eds, 'Systemic Perspectives on Discourse, Volume 1', Ablex, Norwood, New Jersey. Also available as USC/Information Sciences Institute, Reprint Report ISI/RS-87-179, 1987.

Kasper, R. T. (1989), A flexible interface for linking applications to PENMAN's sentence generator, *in* 'Proceedings of the DARPA Workshop on Speech and Natural Language'. Available from USC/Information Sciences Institute, Marina del Rey, CA.

Kasper, R. T. & O'Donnell, M. (1990), Representing the Nigel grammar and semantics in LOOM, Technical report, USC/Information Sciences Institute, Marina del Rey, California.

Kay, M. (1979), Functional grammar, *in* 'Proceedings of the 5th meeting of the Berkeley Linguistics Society', Berkeley Linguistics Society, pp. 142 – 158.

Kumano, T., Tokunaga, T., Inui, K. & Tanaka, H. (1994), Genesys: an integrated environment for developing systemic functional grammars, *in* 'Proceedings of the Workshop on Sharable Natural Language Resources', Nara, Japan. Also available as: Department of Computer Science, Tokyo Institute of Technology, Technical Report: 94TR-0028.

Mallery, J. C. (1994), A Common LISP Hypermedia Server, *in* 'Proceedings of the 1st. International Conference on the World-Wide Web', CERN, Geneva.

Mann, W. C. (1983*a*), 'The anatomy of a systemic choice', *Discourse Processes* . Also available as USC/Information Sciences Institute, Research Report ISI/RR-82-104, 1982.

Mann, W. C. (1983*b*), An overview of the PENMAN text generation system, *in* 'Proceedings of the National Conference on Artificial Intelligence', AAAI, pp. 261–265. Also appears as USC/Information Sciences Institute, RR-83-114.

Mann, W. C. (1985), 'The anatomy of a systemic choice', *Discourse Processes* **8**(1), 53–74. Also available as ISI/RR-82-104.

Mann, W. C. & Matthiessen, C. M. (1985), Demonstration of the Nigel text generation computer program, *in* J. D. Benson & W. S. Greaves, eds, 'Systemic Perspectives on Discourse, Volume 1', Ablex, Norwood, New Jersey, pp. 50–83.

Matthiessen, C. M. (1984), Choosing tense in English, Technical Report ISI/RR-84-143, USC/Information Sciences Institute, Marina del Rey, CA.

Matthiessen, C. M. (1987), Notes on the organization of the environment of a text generation grammar, *in* G. Kempen, ed., 'Natural Language Generation: Recent Advances in Artificial Intelligence, Psychology, and Linguistics', Kluwer Academic Publishers, Boston/Dordrecht. Paper presented at the Third International Workshop on Natural Language Generation, August 1986, Nijmegen, The Netherlands.

Matthiessen, C. M. (1995), *Lexicogrammatical cartography: English systems*, International Language Science Publishers, Tokyo, Taipei and Dallas.

Matthiessen, C. M. & Bateman, J. A. (1991), *Text generation and systemic-functional linguistics: experiences from English and Japanese*, Frances Pinter Publishers and St. Martin's Press, London and New York.

Matthiessen, C. M., Nanri, K. & Zeng, L. (1991), Multilingual resources in text generation: ideational focus, *in* 'Proceedings of the 2nd Japan-Australia Joint Symposium on Natural Language Processing', Kyushu Institute of Technology, Kyushu, Japan.

Mellish, C. S. (1988), 'Implementing systemic classification by unification', *Journal of Computational Linguistics* **14**(1), 40–51.

Monachini, M. & Calzolari, N. (1994), Synopsis and comparison of morphosyntactic phenomena encoded in lexicons and corpora. a common proposal and applications to European languages, Technical report, Istituto di Linguistica Computazionale. Draft version of EU-LRE Project Eagles deliverable EAG-LSG/IR-T4.6/CSG-T3.2.

Nerbonne, J. (1992), 'Representing grammar, meaning and knowledge'. (Papers from KIT-FAST Workshop, Technical University Berlin, October 9th - 11th 1991).

Penman Project (1989), PENMAN documentation: the Primer, the User Guide, the Reference Manual, and the Nigel manual, Technical report, USC/Information Sciences Institute, Marina del Rey, California.

Pollard, C. & Sag, I. A. (1987), *Information-based syntax and semantics: volume 1*, Chicago University Press, Chicago. Center for the Study of Language and Information; Lecture Notes Number 13.

Rösner, D. & Stede, M. (1994), Generating multilingual documents from a knowledge base: the TECHDOC project, *in* 'Proceedings of the 15th. International Conference on Computational Linguistics (COLING 94)', Vol. I, Kyoto, Japan, pp. 339 – 346.

Sefton, P. M. (1990), Making plans for Nigel (or defining interfaces between computational representations of linguistic structure and output systems: Adding intonation, punctuation and typography systems to the PENMAN system), Technical report, Linguistic Department, University of Sydney, Sydney, Australia. Batchelor's Honours Thesis.

Simpkins, N. K. (1992), ALEP-0 Version 2.2: Prototype Virtual Machine, Technical report. (User guide for the Advanced Language Engineering Platform). BIM.

Steele Jr., G. L. (1990), *Common Lisp: the language*, (2nd. edition) edn, Digital Press.

Teich, E. (1992), Komet: grammar documentation, Technical report, GMD/Institut für Integrierte Publikations- und Informationssysteme, Darmstadt, West Germany.

Tomita, M. & Carbonell, J. G. (1986), Another stride towards knowledge-based machine translation, *in* 'Proceedings of COLING 86', pp. 633–638. 11th. International Conference on Computational Linguistics; Bonn, August.

Uszkoreit, H. (1991), Strategies for adding control information to declarative grammars, *in* 'Proceedings of the 1991 Meeting of the Association for Computational Linguistics', Association for Computational Linguistics, Berkeley, California.

Zajac, R. (1992*a*), 'Inheritance and constraint-based grammar formalisms', *Computational Linguistics* **18**(2), 159 – 182. (Special issue on inheritance: 1).

Zajac, R. (1992*b*), Towards computer-aided linguistic engineering, *in* 'Proceedings of COLING-92', Vol. II, pp. 828 – 834.

Zeng, L. (1992), ML-Penman: implementation notes, Technical report, GMD/IPSI and University of Sydney.

Appendix A

Information display flags

When running in teletype mode, without the benefit of the window interface, it is necessary to control the amount of detail given during tracing by means of the actual flags maintained within the system. These are the flags the mode option menus set internally. The flag names are listed below in Section A so they can be used directly from a Lisp listener. The value `nil` is unset, the value `t` set. All the flag variables are in the Lisp package *kpml*.

A further list of flags and internal variables useful for some debugging situations is given in Section A.1. **This latter list includes some internal flags that are *not* available from a user menu. These are flags that are more for internal system debugging than resource debugging, although they might prove useful in exceptional circumstances. In addition, some other resource debugging possibilities that have not yet been incorporated in the user interface are also given below.**

A third list given in Section A.2 contains those internal variables that control the various modes for loading and storing linguistic resources as described in Section 5.7.2.2.

Finally, Section A.3 lists some of the global variables that might impact on the KPML user.

Modes and internal flags

The format of these descriptions is: first the name of the mode as it appears in the mode menu, then a brief description of the function of the flag, and finally the name of the flag itself.[1]

Create Boundary At All Choosers: Allows manual choice of several convenient debugging tools whenever a chooser is reached by forcing a *boundary condition* at the end of the execution of a chooser. Normally by the time execution of a chooser is complete, the chooser is able to choose one of the alternatives that the system calling the chooser is required to make. If the chooser is unable to make a choice we say that a boundary has been reached. KPML then provides several options to help debug why the chooser was

[1]The reader might wonder as to why there is an inconsistency in the naming; some of the flags have names of the from `*...*`, most do not. This is a relic of the old Penman code still underlying much of KPML. The flags with stars are in areas that have been reworked more recently.

not able to choose. If the *Create Boundary* flag is set, a boundary condition is *always* declared at the end of running a chooser, even if the chooser was able to make a choice. This therefore provides one fairly low-level way of breaking into the generation process. In the case that a choice was made, this information is displayed prior to reporting the boundary condition. *create-boundary-at-all-choosers-flag*

Store Implemented Values Into Example Record: Stores implemented value into an example record, queries if example record value already exists. *domain-implemented-value-compare-flag*

Make New Choosers: If an entered system has no chooser allows you to specify one. Such a situation could occur during grammar development; for established resources it should not of course occur. *make-new-choosers-flag*

Manual Guidance For Entailed Inquiry Responses: Allows choice between the entailed response, the environment's response or a new value. *manual-guidance-flag*

Realize Selectively: Allows option of realizing or skipping each grammar constituent. *realize-selectively-flag*

Show Associations: Prints the association table at the end of each pass through the grammar. *display-association-flag*

Show Cautions: Prints Cautions in lisp listener window. *show-cautions-flag*

Show Constituent Starts: Prints what function bundle is being realized at the start each pass through grammar. *show-constituent-starts*

Show Dependency Choices: Prints the systems ready for entry and the system entered. *show-dependency-choices*

Show Disabled Systems: Prints names of any disabled systems that would have been entered had they not been disabled. **show-disabled-systems-flag**

Show Hubname Selection-Expression Discrepancy: Choice of rejecting a run if its values differ from environment, otherwise retains run. *show-hubname-discrepancy-flag*

Show Immediate Realizations: Prints each realization operator as it is invoked. *show-immediate-realizations-flag*

Show Lexical Selections: Prints information on how each lexical item is chosen. *show-lexical-selection*

Show Ordering Constraints: Prints ordering constraint information when computing orderings. *ordering-dump-flag*

Show Ordering Events: Prints each ordering relation as it is inserted in ordering relations table. *show-ordering-events-flag*

Show Ordering Results: Prints each function structure and its resultant ordering. *show-orderings*

Show Pledges: Prints each pledge realization operator as it is invoked. *show-pledges-flag*

Show Entailed Inquiry Response: Prints message whenever a response to a query operator entailed by preselection is used. *show-preselected-response-flag*

Show Preselections: Prints the preselected grammatical features at the start of each pass through the grammar. *show-preselections-flag*

Show Selection Expression: Prints the grammatical feature selections at the end of each pass through the grammar. *show-selection-expression-flag*

Show System And Inquiry Activity: Displays System, System Choice, Inquiry Question, and Inquiry Response Activity. Also the changing entries in the Function Association Table when running *english-trace-flag*

Show Why System Is Entered: Prints the feature that caused a particular system to be entered. *show-why-system-is-firing-flag*

Single Step: Pauses after each query, prompting the user to hit the end key to continue. *stop-action-flag*

Realize until constituent number: Expects a positive integer as value; the system will then pause during generation when a constituent of the specified number is reached and offer the user the opportunity of setting generation display flags (either in a break if the window interface is not present or with the usual menu as described in Section 7.5.2). The number of a constituent can be read directly from the output form of the grammatical functions: e.g., the grammatical function `FINITE#45` was generated during the 45th. cycle through the grammar. **trace-this-constituent**

Update Example Record: Stores user responses to the example record if not in Verify Every Response mode. *store-to-environment-flag*

Note that in order to collect the inquiry responses one needs also to set the variable `Domain-Implemented-Value-Compare-Flag` to T. This occurs automatically when this option is selected in the window interface.

Stop On Warnings: Invokes the debugger whenever a warning occurs. *warning-stop-flag*

Verify Every Response: Asks if an environmental response is ok or if a user response should be stored to the environment. *manual-response-mode-flag*

A.1 More detailed tracing and display modes

The following variables also provide enhanced debugging facilities; they may be incorporated at some stage in general menu options available from the user interface directly.

`*break-on-no-target*` – when set causes generation to be suspended whenever generation is attempted on an example record for which there is no target string defined.

`*traced-systems*` – contains a list of systems and provides details during generation for the specified grammatical system analogous to that produced for all systems when *Show System and Inquiry Activity* is set.

`*inquiries-to-pause-upon*` – contains a list of inquiries; when execution of any inquiry on the list is required, the system enters the debugger and gives control back to the user.

`*show-lexical-selection-flag*` – either `t` or `nil`; when `t` more information is given during lexical selection, including the domain concepts investigated for possible sources of lexical information, etc.

`*show-morphology-selection-flag*` – either `t` or `nil`; when `t` more information is given during morphological selection, including the complete constraints given from the grammar and those which are considered relevant for morphology.

`*show-loading-actions*` – either `t` or `nil`; when `t` the system reports on each linguistic unit being loaded plus some diagnostics concerning the status of that loading.

`*show-merging-actions*` – either `nil` or a list; depending on the members of the list the system reports on its attempts to merge newly loaded linguistic units with previously existing ones. If the list contains the symbol `:systems`, then details of merging systems are given; if the list contains the symbol `:choosers`, then details of merging choosers are given; if the list contains the symbol `:inquiries`, then details of merging inquiries are given; and if the list contains the symbol `:lexemes`, then details of merging lexical items are given. These symbols can, of course, be used in any combination. Information is given as to whether the newly defined unit completely replaces previously existing units, whether language gaps have been created, whether merging was not possible, etc. This flag can also be used in conjunction with the following `*step-by-step-merging-behavior*` for selectively giving very fine detail.

`*step-by-step-merging-behavior*` – either `t` or `nil`; intended primarily for internal system debugging since it presents the finest granularity possible of KPML merging behaviour. The values identified as `KPML>MERGING:ARG-LIST` are the successive parameters that are collected for passing to the internal function `merge-lx`. The values identified as `KPML>MERGING:RESULT` are the results of merging for the given parameters. If problems are suspected with the merging, a bug report should be sent containing the values printed with this flag set for the linguistic units where the result of merging is in doubt.

`*example-display-desired-mode*` – either `t` or `nil`; toggles the information displayed in the menu for selecting an SPL for generation brought up by the *Generate Sentence* command. When `t` the desired sentence is displayed, as given in the `:englishform` slot of the example. Otherwise, the actually generated strings are shown.

`*example-differences-mode*` – either `t` or `nil`; when `t` only SPLs whose actually generated strings differ from their desired results (as indicated in the `:englishform` slot) are offered for generation under the *Generate Sentence* menu. The comparison is simple string comparison. An example use of this would be to load a set of examples, use the example runner to run through all SPLs, and then to inspect those SPLs that did not generate as expected.

`supplement-preselections-flag` – either `t` or `nil`; when `t` the user is asked prior to the generation of each grammatical unit whether additional preselections are to be considered active for that unit.

A.2 Loading and storing modes

`*merging-active*` – either `t` or `nil`; when `t` newly loaded resources are *merged* with existing resources rather than overwriting them.

`*acquire-lexical-items-mode*` – either `t` or `nil`; when `t` undefined lexical items mentioned in `:lex` or `:name` slots in SPL expressions are created on the fly (cf. Section 5.9.4). Such newly created lexical items are recorded on the list `*new-lexical-items*`.

A.3 Miscellaneous global variables

There are several global variables functioning as flags and switches for controlling the behaviour of KPML that are in addition to the tracing and debugging flags shown above. Many of these are not available directly from menus. This section gives a list of those that might occasionally impact on the user of KPML, including those that enable the system's behaviour to be customized somewhat according to individual preferences or needs.

The following variables are in the `user` Lisp package.

`*root-of-kpml*` Maintains the directory where KPML is installed.

`*root-of-resources*` Maintains the directory that currently serves as the root of all linguistic resource definitions for the language varieties being developed or used.

`*kpml-pathname-default*` Maintains the root directory of the KPML system.

`*loom-pathname-default*` Maintains the root directory of the Loom knowledge representation system (which must have been previously compiled if it is to be used). Loom 2.0 and 2.1 are supported as default by the present release of KPML.

`*kpml-binaries-root*` Maintains the root directory for the placement of binaries produced when KPML is compiled. This is normally set by the installation process transparently to the user, but can be usefully manipulated if required. A value given to this variable *prior* to installing KPML will take precedence over the KPML default.

The following variables are in the `kpml` Lisp package.

`complexity-maximum` This contains an integer that limits the number of constituents that will be generated for a given call to the generator. Its purpose is to avoid erroneous infinite regressions. It's default value for Penman releases was 40; for KPML releases this has been increased to 100 since when morphology is included it is very easy to have more than 40 constituents present.

`all_languages` Maintains the list of language varieties that KPML knows about at any time. If language varieties not on this list are encountered in resource definitions, those resource definitions will probably cause an error.

`curlan` Maintains the current language for which KPML is providing information or generating.

`*demo-mode*` When set `T` suppresses all warnings during generation.

`*in-ml-region*` When set `T` default language conditionalizations are merged into any resources evaluated; when not set, no default language conditionalizations are considered.

`*new-lexical-items*` Holds the list of newly created lexical items when the auto-create flag for lexical items is set (cf. Section 5.9.4).

`*package-for-inquiry-implementations*` Must be set to either a string denoting the package or a (dotted pair) association list of languages (as specified in `all_languages`) and such strings; used for changing packages for inquiry implementations in inquiry definitions when saving resources (Section 5.9.6).

The following variables are in the `kpml-i` Lisp package.

`*auto-print*` When `T`, postscript files depicting resource graphs, etc. are immediately sent to a printer when first created.

`*global-font-switching*` When `T`, changes in language may change the font for the main KPML windows—particular the *Interaction results* panes.

`*hardcopy-structure-orientation*` Controls the orientation of graphs; may be either `:vertical` or `:horizontal`.

`*show-collecteds*` When `:always`, always puts a pane showing the list of collected features in a resource graph; when `nil`, never put a collected features pane in a resource graph; and when `T`, put a collected features pane in only when there are collected features.

Appendix B

Inquiry Operator Data Access Functions

(Extract from Bob Kasper's description in the *Penman Reference Manual*: references to Penman and to English generalize in the KPML context to KPML and all supported language resources.)

The design of Penman allows the developers of applications to define their own implementations of Penman's inquiry operators. Although many applications should be able to use the inquiry operator implementations that are provided with Penman, some applications may achieve the best results by customizing some of the inquiry operator implementations according to the kind of knowledge that is available. Customization may be appropriate for several reasons:

- the application uses a knowledge representation framework that is significantly different from that assumed by Penman;
- the application has very specific kinds of knowledge that can be used to answer Penman's inquiries (i.e., kinds of knowledge that might not be used in other applications to answer the same inquiries);
- the application developer would like to use some features of English in a way that is inconsistent with Penman's standard implementation of an inquiry.

The inquiry implementations provided by Penman are written as Lisp functions. Although it is possible to use any Lisp code, most implementations use a small collection of access functions to find information contained in the SPL specification or the application's knowledge-base. These access functions can (and should) be used in developing customized inquiry implementations, or equivalent functions can be written if the application requires using a different programming language, instead of Lisp.

Term-Graph structures

When Penman is invoked using an SPL sentence plan, the plan is parsed and stored in internal data structures. Each term of an SPL plan is stored in a structure called a `term-graph`. The following functions can be used to access individual fields of a `term-graph` structure:

```
(term-graph-id term-graph)
(term-graph-symbol term-graph)
(term-graph-type term-graph)
(term-graph-features term-graph)
(term-graph-parent term-graph)
```

Each `term-graph` has a unique identifier (stored as its `term-graph-id`), which is a Lisp symbol that yields the `term-graph` when evaluated. Such term-id symbols are generally returned as the responses to Penman's identifying inquiries. The `term-graph-symbol` field of the structure contains the actual variable or constant that appears in the SPL plan; the values in this field may then be EQ across several `term-graph` structures when the plan has co-referential terms. When a term is a set, this field will contain a list of the elements of the set. The `term-graph-type` field is a knowledge-base concept or a list of such concepts. The `term-graph-features` field is an association-list in which the keys are feature names, and the values are (typically) `term-graph` structures. The `term-graph-parent` field is a pointer back to the term in which this term was embedded, or `NIL` for terms occurring at the top-level of an SPL plan.

Note: some aspects of SPL interpretation rely on particular concepts being available in the version of the upper model that is loaded. This enables the SPL interpreter to state that it has recognized, for example, 'sets', and to distinguish upper model 'relations' from inquiry preselections. Changing the upper model so that it does not include the following concepts can, therefore, lead to unexpected consequences. The required concepts are: `um-set`, `disjunctive-set`, `two-place-relation`: see Section 12.4.2.

Other Access Functions

The following functions are also used to gain access to various kinds of information with respect to SPL term-graph structures.

`(fetch-atomic-feature feature-name term-id)`
Returns the symbol that is the value of the `feature-name` feature of `term-id`. It is used with features, such as `:lex`, which have atomic values.

`(fetch-feature feature-name term-id)`
Returns the `term-graph-id` of the term that is the value of the `feature-name` feature of `term-id`.

`(fetch-feature-symbol feature-name term-id)`
Returns the `term-graph-symbol` of the term that is the value of the `feature-name` feature of `term-id`.

`(fetch-minimal-relation relation-name term-id)`
Returns either the value of the `relation-name` feature of `term-id`, or a reified relation of type `relation-name` from the value of the `:relations` feature of `term-id`. Only reified relations which have a minimal set of features (nothing other than `:domain` and `:range`) are returned. The caller will not know whether the returned value is the relation's range or a reified relation.

`(fetch-non-minimal-reified-relation relation-name term-id)`
Returns a reified relation of type `relation-name` from the `:relations` feature of `term-id`. Only reified relations which have a non-minimal set of features (something in addition to `:domain` and `:range`) are returned.

`(fetch-reified-relation relation-name term-id)`
Returns a reified relation (either minimal or non-minimal) of type `relation-name` from the `:relations` feature of `term-id`.

`(fetch-relation relation-name term-id)`
Returns either the value of the `relation-name` feature of `term-id`, or a reified relation of type `relation-name` from the value of the `:relations` feature of `term-id`. The caller will not know whether the returned value is the relation's range or a reified relation.

`(fetch-relation-spec term-id relation-name new-term-id)`
Returns a reified relation of type `relation-name`, reifying a non-reified relation if necessary. The relation may be found either

- **non-reified:** as the `relation-name` feature of `term-id`, or
- **reified:** as a value of the `:relations` feature of `term-id`.

When a non-reified relation is found, then `new-term-id` provides an identifier to be used for a newly constructed reified relation.

`(fetch-relation-range relation-name term-id)`
Returns either the value of the `relation-name` feature of `term-id`, or the value of the `:range` feature of a reified relation of type `relation-name` from the `:relations` feature of `term-id`.

`(fetch-subc-feature feature-name term-id)`
Returns the `term-graph-id` of the term that is the value of the `feature-name` feature of `term-id`, or any feature which specializes `feature-name`. If `term-id` is not bound, look for `feature-name` in some co-referential term.

`(get-global-terms term-id)`
Returns the ids of all terms from `*plan-graphs*` (the current sentence plan) that are co-referential with `term-id`.

`(get-symbol-term term-id)`
Returns the id for a term that is co-referential with `term-id`.

`(global-fetch-feature feature-name term-id)`
Returns the value of a feature of type `feature-name` from `term-id`, or from some term that is co-referential with `term-id`, if no such feature is found in `term-id`.

`(term-eq-p term1 term2)`
Predicate is true if `term1` and `term2` are co-referential (i.e., either they are identical atoms,

or they are terms having the same `term-graph-symbol`).

`(term-role-p term-id1 term-id2 role)`
Predicate is true if `term-id2` participates in a `role` relation with `term-id1`.

`(term-type-p term-id given-type &optional (non-local-test? t))`
Predicate is true if `given-type` is the same as or a superc of the type of `term-id`. When `non-local-test?` is true, then look also at the types of any terms that are co-referential with `term-id`.

Appendix C

Knowledge representation interface functions

All interaction between KPML and a supporting knowledge representation language is managed via the following interface functions. Using a knowledge representation other than Loom therefore requires these functions to be redefined.[1] A version of the upper model should then also be prepared in the target knowledge representation.

Loom makes a distinction between *concepts* and *relations*; if an alternative knowledge representation language does not make this distinction, then the corresponding pair of functions `kb-getnamedconcept` and `kb-getnamedrelation` can receive the same implementation. The linking between domain concepts and lexical items described above (Section 12.2.13) is defined solely in terms of these interface functions and so does not need additional adjustment.

- KB-CONCEPTDISJOINT?, Function (C1 C2)

  Returns true if concepts `C1` and `C2` belong to disjoint classes.

- KB-ENTITY?, Function (INSTANCE)

  Returns true if `instance` is a knowledge representation instance.

- KB-GETCONCEPTNAME, Function (CONCEPT)

  Returns a print name for the knowledge representation concept `concept`.

- KB-GETNAMEDCONCEPT, Function (CONCEPT)

  Returns the knowledge representation concept that has the print name `concept`.

- KB-GETNAMEDRELATION, Function (RELATION)

  Returns the knowledge representation relation that has the print name `relation`.

- KB-IMMEDIATESUBCS, Function (CONCEPT)

  Returns the immediate subconcepts of the concept `concept` (ignoring any internal system-defined concepts that may appear).

[1] There are two additional functions used in the old Penman experimental nominal phrase planner; this code is not normally used. The functions are: `kb-relations` and `kb-identifier`.

- KB-IMMEDIATESUPERCS, Function (CONCEPT)

  Returns the immediate superconcepts of the concept `concept` (ignoring any internal system-defined concepts that may appear).

- KB-PACKAGE, Function

  Returns for LOOM 2.0, the Lisp package of the current knowledge base and for LOOM 2.1, the current knowledge base context name. This simply ensures that the concept access functions try placing any symbols with which they are presented as arguments in appropriate packages.

- KB-SUPERP, Function (C1 C2)

  Returns true if the concepts or relations `C1` and `C2` stand in a super-type relationship.

Appendix D

Locations of finer user control for generated strings

Here we summarize the locations where a user or external component can effect the strings that are generated by KPML. These range from locations that would be theoretically expected given a linguistically motivated generation component to locations that are more practically motivated in fine (and possibly arbitrary) rule-based manipulations of the strings generated. For practical generation purposes it may be necessary or useful for an application to consider direct alteration of the strings generated in order to fine-tailor them for particular needs. All points of possible control readily accessible to the user are listed here.

We first present an overview of the generation process in terms of the *structures produced* at each stage, detailing the possibilities for influencing these structures directly. This is then summarized graphically in Figure D.1.

D.1 Structures produced during generation

The final string as it is produced by the generation process is only the last in several intermediate stages on the path from semantic input to string. In KPML these stages are open to user control in various ways. The structures include:

1. the constituent tree
2. the punctuated mouseable constituent structure
3. the phonologically/graphologically edited mouseable structure
4. the final 'cleaned-up' string

The rationale for these levels of structure is as follows.

The first structure, the constituent tree, is the grammatical structure as constructed by the realization statements in the system network. It is the prime linguistically motivated result. The constituent tree already includes the results of any ordering constraints applicable.

The second structure is then the result of applying the punctuation rules (Section 12.2.10 to the first structure. Punctuation is therefore seen as a process that is based on the functional grammatical structure. The structural units defined by the grammar still define the units of concern.

The third structure allows the application of various rules for enforcing phonologically or graphologically motivated alterations: these include the conversion of the indefinition article “a” to “an” in English when necessary, the conversion of “de le” to “du” in French, etc. Note that it would be possible to include these variations in the ‘grammar’, but if it is not practically worth the effort then a simple replacement rule can be provided. The phonological conversions still respect to the units of the constituent tree: essentially pairs of adjacent words are compared and the replacement is made if some condition is met. As with any of the replacement rules, they should not be overused. The Lisp function `phonological-processing` is currently the interface to this level of replacements. This takes a single parameter: this is the usually punctuated mouseable result structure. The format of this structure is as described in Section 14.5; when `phonological-processing` is called, the final string clean-ups have of course not yet been carried out. The function directly alters any necessary strings used in the mouseable structure. It is likely that the interface to this level of string alterations will be improved: if it appears necessary to carry out operations at this point, it would be best to contact the author so that an appropriate extension be provided.

The final structure consists of pure string operations and need not respect any grammatical structure. This is used, for example, to remove accumulated punctuation marks (such as “,.”) that may be the result of the functional structure-driven punctuation rules. The format of these rules is described in Section 12.2.2.5.

D.2 Overview of accessible hooks

Here we summarize the intermediate structures and points of possible control. Naturally, the initial input to the generation process is also a source of control for how the sentence appears! And, therefore, the lexical items defined, the inquiry defaults, and SPL macros are similarly places where the final string result can be shaped as required. These are all set out in Figure D.1.

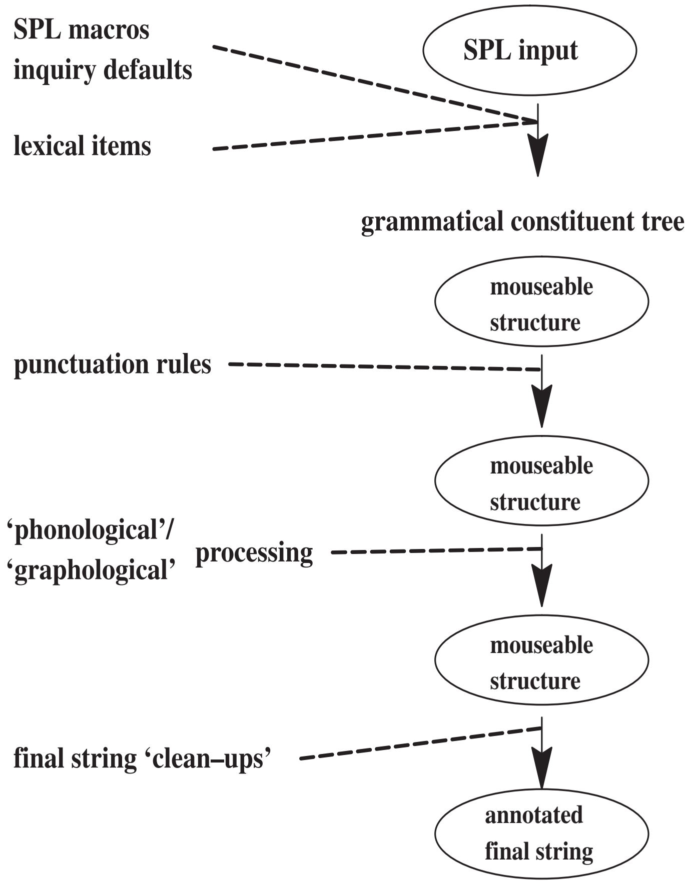


Figure D.1: Points of control for generation strings

Index

D