The RAGS Reference Manual

The RAGS project

August 2000. Reissued with minor corrections, July 2002

At the end of the project, March 2001, the RAGS project is Lynne Cahill[1], Roger Evans[1], Chris Mellish[2], Daniel Paiva[1], Mike Reape[2]and Donia Scott[1].

[1]Information Technology
Research Institute
University of Brighton
Lewes Rd, Brighton
UK

[2]Division of Informatics
University of Edinburgh
80 South Bridge, Edinburgh
UK

rags@itri.brighton.ac.uk
http:/www.itri.brighton.ac.uk/projects/rags

Previous members of the project, who have contributed to this document are:
Christy Doran, now at Mitre Corp, Boston, USA.
Rodger Kibble, now at Goldsmiths College, London, UK.
Neil Tipper, now at Motorola, Sydney, Australia.

Contents

Abstract

This is a slightly reduced version of the reference documents produced after years one and two of the project. It is intended to provide users with a reference manual for using RAGS datatypes and any associated tools/modules produced by the RAGS project. It does not aim to attempt to justify the decisions made in the RAGS project – this is done in a separate document.

Chapter 1

Introduction

The RAGS architecture is a developing specification of a reference architecture for natural language generation (NLG) systems. The goals of the architecture are (in order) to:

> Provide support for applications-oriented system development
> Allow for the general provision of standard interfaces and datasets
> Allow for the reuse of existing ideas and software
> Form a basis for possible evaluation, specialisation, etc.

The RAGS architecture is largely based on our conception of current practice in the building of applications-oriented NLG systems, which we have attempted to rationalise and which we have made partly open-ended in order to facilitate what we consider to be potentially useful extensions to current practice. It follows that we hope that, even though the terminology and notation cannot easily reflect all existing work, nevertheless the substance of the detailed specifications that appear here will not be particularly surprising or novel to workers in NLG.

The RAGS architecture is defined in a set of stages in such a way that researchers can "buy into" parts of the model without having to embrace the whole thing. This is possible through the architecture having three separate parts:

1. An *abstract data model* which specifies the types of data manipulated by an NLG system, how they can be represented in concrete terms and what operations are *possible* on them.
2. Suggested *instantiations* of the types in the abstract data model (which are *parameterised*) which tie the representations to particular theories or sets of theories.
3. A set of possible architectures constructed by adding further constraints to the ordering of operations in the data model.

This document presents primarily a reference description of the abstract data model. A companion document [4] provides more general discussion and motivation. The present document is relatively formal, precise and concise, in order that somebody wishing to use RAGS can know exactly what is involved.

The abstract data model allows for the following principal levels of representation within an NLG system:

1. **Conceptual**, i.e. non-linguistic representations constructed by external agents in accordance with their own purposes

2. **Rhetorical**, i.e. structures organised for a rhetorical purpose
3. **Document**, i.e. structures pertaining to layout and function of a text as a document
4. **Semantic**, i.e. linguistically-oriented representations of meaning
5. **Syntactic**, i.e. conventional syntactic structure
6. **Quote**, i.e. fixed output for some medium that can be incorporated into a generated document

There are certain natural principles of precedence/ordering between these levels which are assumed:

- Everything is eventually derived from some conceptual representation.
- Rhetorical and document structures need to reference semantic and syntactic information.
- The final output of the NLG system will be syntactic structures and certain kinds of document structures, possibly including some fixed "quotes".

The main chapters of this document describe the levels of representation in turn. These proposals are augmented by a set of more detailed implementation requirements set out in Chapter 9, which also describes how combinations of the different levels can be used. Each chapter usually has the following parts:

1. **Executive Summary** – serves to briefly outline the idea of the representation, where it came from, how it is different from others, etc.
2. **Abstract Type Definitions** – defines the types of entities involved in this level of representation and their parts in an abstract way, together with a simple English gloss. The notation used in the type definitions is explained in Appendix A.
3. **Example Instantiations and Surface Syntaxes** – indicates some values for the parameters of the representation that correspond to well-known or recommended theories that groups of researchers may want to adopt (see Chapter 2). Shows some (hopefully fairly familiar) surface syntaxes that could be used for representations at this level. Also, for completeness, shows some examples in the graphical "objects and arrows" notation of Chapter 9.

The concrete syntaxes given in these chapters are supposed to be illustrative examples only. The only concrete syntax that RAGS explicitly endorses is the encoding in XML given in Chapter 10. We have chosen XML [6] for the concrete syntax of an interchange language for the RAGS levels of representation, in the expectation that XML will shortly become very widely used in the publication of electronic documents, given the existing use of SGML/XML in large scale datasets and interfaces [22] for NLP.

We greatly value feedback from the NLG community, because a reference architecture can only be of any use if it genuinely reflects a consensus of the community. The RAGS architecture is still incomplete in a number of ways and there are likely to be deficiencies in what is defined here. This document describes the final specification that has emerged as the result of the funded RAGS project involving the Universities of Brighton and Edinburgh, but this should be thought of as only the first step in a much longer discussion process.

Chapter 2

Parameterisation

NLG systems are built assuming many different theories of syntax, semantics, rhetoric, etc. It would be impossible for RAGS to produce a generally-acceptable complete definition of representations at any of the six levels as there is so much theoretical diversity. Instead, RAGS defines the different levels of representation primarily in terms of *abstract type definitions* which specify the components out of which these representations are expected to be made. Thus, for instance:

$$\begin{aligned} \mathit{RhetRep} &= \mathit{RhetRel} \times \mathit{RhetRepSeq} \\ \mathit{RhetRepSeq} &= (\mathit{RhetLeaf} \cup \mathit{RhetRep})^+ \end{aligned}$$

(from chapter 4) defines[1] a Rhetorical Representation (*RhetRep*) as consisting of a *RhetRel* (rhetorical relation) and a *RhetRepSeq* (sequence of *RhetRep*s) for the children. The latter is itself defined in terms of a sequence of elements, each of which is a *RhetLeaf* or a *RhetRep*.

The abstract type definitions "bottom out" in *primitive* types, which are indicated as follows:

$$\mathit{RhetRel} \in \mathit{Primitives}$$

RAGS has nothing further to say about the type *RhetRel*, which is the set of rhetorical relations. Individual groups of researchers are invited to *instantiate* this set as they see fit according to any theoretical position they wish to adopt. That is, RAGS does not specify the *subtypes* of primitive types (here, the particular relations, or classes of relations, that some theory specifies), apart from to acknowledge that they may exist. Different implementations may make different assumptions about the subtypes of primitive types, though of course most effective reusability of code or data will be possible between researchers that do agree.

In that they are all expressed in terms of primitive types, all the definitions of RAGS datatypes are thus *parameterised* – they define the overall format of a set of structures but do not stipulate the exact set of values that can appear at the leaves of these structures. This is because, although researchers often agree about the overall nature of a particular kind of representation, very often they do not agree on the actual set of rhetorical relations, concepts,

[1] The notation used for the abstract type definitions is explained in Appendix A.

semantic case roles, etc. that are used to give substance to the structures, or they need a domain specific set of the above.

Someone who wishes to use or make available RAGS datasets of a particular kind needs to make it clear what instantiation of the generic definition they are using. In general, this involves producing a listing of the elements in certain sets (e.g. a list of rhetorical relations), together with some indication of the semantics of these elements. As well as this, in some circumstances further structure may need to be provided (for instance, a subsumption relation between elements in a list of concepts). Our interface definitions should eventually make it quite clear what an instantiation has to provide and make it simple for people to declare such instantiations in a way that permits easy determination of whether data and processing components can be reused for a given task or not. There does not necessarily have to be an exact match between the information provided in an example dataset and the information required by a given processing component. For instance, one could imagine a general module for referring expression generation which only needs to know about certain very coarse distinctions among facts in a knowledge base, even though the contents of that knowledge base would probably be made available in a way that contained a lot more information (needed, e.g. for a more general content-determination module).

In RAGS we are trying hard to avoid making commitments at a detailed level to any particular theory of a part of NLG. This is why we have adopted this parameterisation approach. Of course, sharing of resources will be maximally efficient between researchers that agree on the instantiation as well as the general form of a representation. It is to be hoped that some instantiations will become generally popular, and we intend to make suggestions about possible instantiations (based on our implementations) in order to stimulate this process.

Chapter 3

Conceptual Representation Datatype Specification

3.1 Executive Summary

The *Conceptual* level of representation allows pointers to non-NLG representations, which nevertheless satisfy a general API (Application Programmer Interface), to be included in RAGS representations.

Conceptual Representations differ from Semantic ones in that they are outside the control of the NLG system and hence may not be designed with language in mind. For instance, whereas a table of numerical data might be Conceptual, a summary of this in terms of concepts like `increase`, `slow` and `significant` (which have simple linguistic realisations) is more likely to be Semantic. In the RAGS model, Conceptual Representations have to provide an API that allows them to be related to the Semantic terms in which an NLG system operates. RAGS takes no position on what Conceptual Representations are "really like" though.

In deciding whether a given representation should be Conceptual or Semantic, the following questions might be useful. Answering "yes" to any of these would suggest the use of Semantic, rather than Conceptual.

- Are the entities in the representation at the same kind of granularity as words and other constructs in the kind of language the NLG system is to produce?
- Is it possible to have two different representations that express the same "content" but which suggest (possibly rather indirectly) two different subclasses amongst the set of possible realisations?
- Do you want to point at certain kinds of complex representations (e.g. conjunctions, quantifications) as corresponding to the "meaning" of linguistic constructions (such as NPs, VPs and Ss)?
- Do you want to be able to build new representations as part of the NLG process (rather than just select from what is provided) or transform/modify existing representations?

The API described below is very close to the abstract interface between the PENMAN system and its knowledge base. Indeed, our decision to specify conceptual representations via an API appears to be adopting the PENMAN *inquiry-based* approach to semantics (i.e. suggesting grammar-driven, rather than message-driven, generation). The fact that Conceptual representations can contain more or less direct guidance about non-content issues (e.g. salience) and that an NLG system can receive input using other RAGS representation levels, however, means that RAGS does not take a position on which decisions are made within the NLG system and which outside. The following is also compatible with the view [23] that an NLG system's interaction with an external knowledge source should be through a controlled interface that provides some protection from deficiencies and irregularities in the underlying data.

3.2 Abstract Type Definitions

Conceptual Representations can only be manipulated through pointers that have no explicit meaning to the NLG system. These are called "Knowledge Base IDs" or `KBIds`.

$$KBId \in Primitives$$

`KBId`s are required to satisfy the following API (which may require the "knowledge base" to make arbitrarily complex inferences):

3.2.1 Basic Type Checking Functions

$$\begin{aligned} kb\_isstring : &\quad KBId \to Bool \\ kb\_isnumber : &\quad KBId \to Bool \\ kb\_equivalent : &\quad KBId \times KBId \to Bool \end{aligned}$$

(where $Bool$ is the set `{true,false}`).

These check whether a $KBId$ is actually a string or number and whether two $KBId$s denote the same thing.

3.2.2 Data Access/Testing

The following functions make use of a set of possible "type/predicate names" $SemPred$ and a set of possible "role names" $SemRole$. These are the same primitive types defined for Semantic Representations. Thus, although they are not Semantic Representations, Conceptual Representations are required to act (in some respects) as if they can relate themselves to Semantic distinctions.

$$\begin{aligned} kb\_subsumed\_by : &\quad KBId \times SemPred \to Bool \cup \{\texttt{Unknown}\} \\ kb\_role\_values : &\quad SemRole \times KBId \to 2^{KBId} \cup \{\texttt{Unknown}\} \\ kb\_types : &\quad KBId \to 2^{SemPred} \\ kb\_roles\_with\_values : &\quad KBId \to 2^{SemRole} \end{aligned}$$

The first of these is to test whether a given KBId is subsumed by a type, and the second is to access the values of a given role in the view for a KBId. These two functions are

extended to allow arbitrary values in the place of an element of $SemPred$ or $SemRole$. These functions return the special value `Unknown` if the relevant name is not in $SemPred$ or $SemRole$ respectively.

$$Unknown \in Primitives$$

The third function returns the set of types that the KBId belongs to[1]. The final function indicates for a KBId which roles (out of those in $SemRole$) have values.

$SemPred$s are a way of naming types of KBId (e.g. `human`, `animate`) and $SemRole$s (e.g. `father_of`, `designer_of`) are a way of naming roles that may be present for KBIds. Both of these types are used also in semantic representations (chapter 6). A system may decide to use the same $SemPred$s and $SemRole$s at all stages of processing, or it may choose to have distinguished subsets which are used for dealing with KBIds. $SemPred$s allow one to access complex types of data whose components are accessed via $SemRole$s. It is assumed that this would be used to allow KBIds to represent, for instance, sets and sequences if that was needed for a given class of applications.

3.3 Example Instantiations and Surface Syntaxes

The following instantiation of the above primitive types is used in a Prolog interface to a simplified ILEX knowledge base about pieces of modern jewellery at Edinburgh [15].

$$\begin{aligned} SemRole &= \{arg1, arg2, pred, nuc, sat, reln, inv\_arg1, inv\_arg2, inv\_nuc, inv\_sat, name\} \\ SemPred &= \{entity, fact, reln, jewellery, spatio\_temporal\_thing \ldots\} \end{aligned}$$

In fact, what is being modelled here is what the ILEX project calls the "content potential" – a language-oriented repository of the material that could be included in a text. In ILEX, this representation is mostly created by a preprocessing step which works on an underlying database and a set of rules of domain knowledge. That is certainly one way to create a Conceptual representation; another way would be to do this processing on demand when it is needed to evaluate a function specified in the API.

In this instantiation, each KBId represents a domain entity (`entity`; this includes qualities and generic entities), a "fact" that could be expressed (`fact`) or a rhetorical relation that could be expressed (`reln`). In addition, some KBIds (indicating the predicates of facts, rhetorical relations and names of named domain entities) are constants (strings) which have a significance which it is not relevant to expand on here. For `entity` KBIds, there is a large set of other $SemPred$s, representing various distinctions that may be relevant to generation, organised into a complex taxonomy. A user of this knowledge base would need to know the details of this taxonomy.

The $SemRole$s basically implement the following mappings:

$$\begin{aligned} arg1 &: fact \to entity \\ arg2 &: fact \to entity \end{aligned}$$

[1] If the set of types was required to be a partial order, it might be more appropriate to have this only return the maximally specific types.

$$
\begin{aligned}
pred &: fact \rightarrow string\\
nuc &: reln \rightarrow fact\\
sat &: reln \rightarrow fact\\
reln &: reln \rightarrow string\\
inv\_arg1 &: fact \rightarrow 2^{entity}\\
inv\_arg2 &: fact \rightarrow 2^{entity}\\
inv\_nuc &: reln \rightarrow 2^{fact}\\
inv\_sat &: reln \rightarrow 2^{fact}\\
name &: entity \rightarrow string
\end{aligned}
$$

where `pred`, `arg1` and `arg2` correspond to the predicate and arguments of a `fact`, and `nuc` and `sat` correspond to the nucleus and satellite of a `reln`. Here, each `inv_X` role implements the inverse of role `X`.

This example is probably atypical and suboptimal because probably predicates should really be *SemPred*s used to classify `facts` (rather than being strings). Also this way of dealing with rhetorical relations works mainly because with ILEX such relations are purely informational.

The knowledge base of an NLG system, and hence KBIds, could be implemented in any language using constructs of any complexity. Nevertheless, as far as RAGS is concerned, KBIds have no internal structure that can be used (they are to be treated as uninterpreted strings/symbols). In the implementation mentioned above, in fact KBIds are represented by Prolog atoms. Section 10.4 shows how an XML-based interface to a knowledge base could be specified.

3.3.1 Views

The notion of `views` allows multiple knowledge bases which differ in the details of what they provide to, nevertheless, satisfactorily interface to the same processing module. A KB **view** is a pair $< SemPred, SemRole >$ specifying a set of predicates and a set of roles. A knowledge base *supports* a view if it provides the interfaces described above for the particular *SemPred* and *SemRole* given in the view and the values returned for the functions when applied to these sets are never `Unknown`.

For example, the above knowledge base supports the view described by the specification of *SemPred* and *SemRole* given above. It also supports the following more restricted view:

$$
\begin{aligned}
SemRole &= \{\}\\
SemPred &= \{specific\_thing, plural\_thing, countable\_thing, spatio\_temporal\_thing,\\
&\quad nonconscious, female, named\_thing\}
\end{aligned}
$$

because in fact each of the *SemPred*s here is included in the larger set above. This more restricted view might be all that was needed by a module with some specific function (e.g. determiner selection). This comparatively limited requirement would mean that potentially a number of other quite different knowledge bases would be able to satisfy it.

Because the interface functions always return results, it is possible that a particularly resilient processing component may be able to survive when the KB does not support the full view that the component really wants (e.g. it could assume some sensible default value whenever the value `Unknown` is returned).

When someone makes available a KB for general use, a statement of the view(s) supported needs to be supplied. Similarly, a processing component requiring access to a knowledge base needs to specify the minimal view that must be supported. Natural language comments must be provided when one publicises a view, so that a software developer can determine whether a given view makes the distinctions they want/have or not.

Views should be explicitly represented and publicised (e.g. in some textual form) and named so that multiple KBs can support the same view and that people writing different modules can know what functionalities are supported and required by other modules. Hopefully a set of "standard" views will emerge for the requirements of some modules.

Chapter 4

Rhetorical Representation Datatype Specification

4.1 Executive summary

This level of representation is used to represent the *rhetorical* structure of a discourse, defining the relations between the propositions in the discourse. For example, the Rhetorical Representation may specify that two propositions are connected by a CAUSE relation.

Rhetorical Representations differ from Document Representations in that they define the underlying rhetorical relations between parts of a discourse, without specifying anything about how the relations may be realised. Whereas Document Representations define features such as linear ordering and page layout, Rhetorical Representations define what may be termed the *pragmatic* relations.

Following the prevailing computational theories of discourse structure (e.g., Grosz and Sidner [9], Mann and Thompson [13], Hobbs [10]) the RAGS architecture assumes that the rhetorical structure of a discourse is hierarchical. Our data-specification presupposes the use of trees with relations at the nodes and content at the leaves. They are defined in a very general way, and the details are left open to the user. We follow Rhetorical Structure Theory in our example instantiation, but this is only illustrative.

4.2 Abstract Type Definitions

4.2.1 Formal definition

$$\begin{aligned} RhetRep &= RhetRel \times RhetRepSeq \\ RhetRel &\in Primitives \\ RhetRepSeq &= (RhetLeaf \cup RhetRep)^{+} \\ RhetLeaf &\in Primitives \end{aligned}$$

4.2.2 Informal semantics of the representation

Rhetorical Representations (**RhetRep**) are trees with two kinds of node:

1. **atomic (or primitive) node** (**RhetLeaf**) with no descendants (referring to a semantic structure **SemRep**, see section 9.4);
2. **rhetorically complex nodes** with at least two descendents, each of which is itself either a **RhetRep** or a **RhetLeaf**, labelled with a rhetorical relation (**RhetRel**).

No restrictions are made as to the number or type of elements involved in a rhetorical relation. Although we believe that a rhetorical relation that applies to only one proposition is not truly *rhetorical*, but rather *semantic*, we accept that some people may wish to use them in this way. The tree structure also does not represent the left-right text order directly – this information is part of the **DocRep** objects (see Chapter 5). The constituents of the **RhetRepSeq** are, however, *ordered*. Although their order does not necessarily represent their linear order in the final document, it may have other import, such as destinguishing between nucleus and satellite.

4.3 Example instantiations and surface syntaxes

Let us consider a very simple example text:

> Blow your nose so that it is clear

This consists of two propositions, "blow your nose", and "your nose is clear", connected by a rhetorical relation which we shall term "motivation". Let us first consider how a simple RST-style *Rhetorical* representation might look. Figure 4.1 shows a possible representation for this. The semantic representations for the two propositions are labelled *r5* and *r6* respectively.

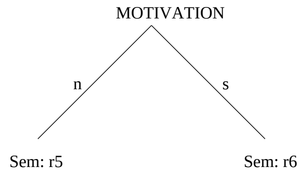


Figure 4.1: Rhetorical tree for "Blow your nose so that it is clear"

In the RAGS object-and-arrow model, this might look something like figure 4.2. This includes reference to two main propositions, the SemReps labelled r5 and r6. The RhetRep here consists of a pair, the first element being the RhetRel, in this case, *motivation*, and the second being the RhetRepSeq, a sequence of RhetReps. In this case, the sequence consists of two RhetReps, each of which is simply a SemRep.

4.3.1 Sample instantiation of example RhetReps

The above simple example might be represented in Prolog (with the semantic representations included) as follows:

```
type(u0, motivation).
attr(nucleus, u0, r5).
  type(u1, blow).
```

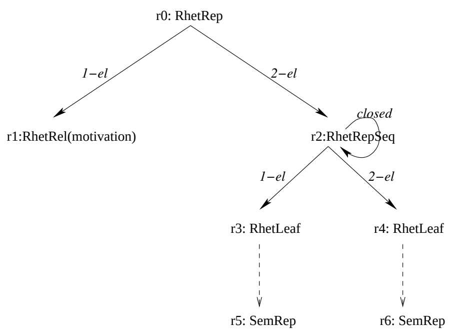


Figure 4.2: RhetRep for "Blow your nose so that it is clear"

```
    attr(actor, r5, u2).
      type(u2, patient).
    attr(actee, r5, u3).
      type(u3, nose).
      attr(owner, u3, u2).
  attr(satellite, u0, r6).
    type(r6, clear).
    attr(subject, r6, u3).
```

The precise nature of the subtype information is not constrained by RAGS, but determined by the implementer. The full implementation that this example comes from includes the following set of rhetorical relations, defined as either nucleus/satellite or multi-nuclear relations:

```
nucleus_satellite_relation(motivation).
nucleus_satellite_relation(evidence).
nucleus_satellite_relation(elaboration).
nucleus_satellite_relation(concession).
nucleus_satellite_relation(purpose).

multinuclear_relation(sequence).
multinuclear_relation(alternative).
```


Chapter 5

Document Representation Datatype Specification

5.1 Executive Summary

Document structure is the level of representation that encodes explicit information about how a *document* is organised, i.e. information relating to the physical layout. It is intended that this level of representation should also be extendable to representation of speech as well as text, although this is as yet untested.

In contrast to Rhetorical representations, which encode the *rhetorical* relations that hold between propositions, the document representations encode the relative position, textual and layout information of the realisation of propositions in a document. This may include things other than text, such as pictures etc..

There is a distinction between *abstract* and *concrete* Document structure, which can perhaps best be illustrated with an example. In preparing a document in LaTeX, we might decide to emphasise a piece of text, by giving the command `{\em ... text ...}`. However, the precise realisation of this, whether in italics, boldface etc. is determined by the style file used. We consider the specification of emphasis to be part of abstract DocRep and the specification of the font type to be part of concrete DocRep. We do not cover the concrete level of representation in RAGS.

There is relatively little previous work that addresses these issues, and our proposals for this level are largely based on our own experiences of representing document structure in the generation process. The proposals for TEXT-LEVEL come more or less directly from Nunberg's text grammar [19].

5.2 Abstract Type Definitions

5.2.1 Formal definition

$$\begin{aligned} DocRep &= DocAttr \times DocRepSeq \\ DocRepSeq &= (DocLeaf \cup DocRep)^{++} \\ DocLeaf &= DocAttr^{1} \\ DocAttr &= (DocFeat \rightarrow DocAtom) \end{aligned}$$

$$\begin{aligned} DocFeat &\in Primitives \\ DocAtom &\in Primitives \\ DocLeaf &\in Primitives \end{aligned}$$

5.2.2 Informal semantics of the representation

The DocRep is a tree structure with two types of node:

1. **atomic node** with no descendants, labelled with a feature structure (and referring to a syntactic structure **SynRep**, see section 9.4).

2. **complex node** with at least two descendents, each of which is itself a **DocRep** or a **DocLeaf**, labelled with a feature structure (and referring to a syntactic structure **SynRep**).

The feature structure (**DocFeat**) consists of features with atomic values. There are no constraints on the features or the sets of values which may be defined.

5.3 Example instantiation of the interface

We propose four features for the **DocAttr**: Text-Level, Layout, Order and Picture.

Order: the relative ordering of the sister nodes in the rhetorical structure. This information informs (or may be informed by) the specification of concrete syntax and choice of discourse marker. As well as specifying ordering of sisters, we allow rhetorical components to 'float', that is, to be unordered with respect to their rhetorical relatives, but positioned according to external layout criteria (for example, figures and footnotes).

Text-level: the text-grammar category of each text element. Following Nunberg [19], we distinguish between a lexical grammar and a text grammar. In RAGS the text grammar categories are specified in the DocRep whereas the lexical grammar categories are properly a part of the Syntactic Representation. A typical text grammar might use categories such as *text, text-sentence, text-clause* and *text-phrase.*

Layout: the properties of the text which will lead to later formatting choices. Typical examples of layout features include: *chapter, section, paragraph, plain-text, vertical-list, list-item, caption, footnote, title* and *picture.*[1]

Picture: the presence or absence of, and location of, a picture associated with a piece of text. The value of this feature is either a filename indicating the location of a picture to be associated with a piece of text, or the value *nil*, indicating that there is no picture associated with that piece of text.

[1]We do not address here the issue of formatting choices at the word level, e.g. small-caps for new terminology or bold face for contrastive emphasis. Again, there is clear overlap here with prosodic annotation for spoken output.

We assume that order and text-level must be obligatorily specified for all nodes in the DocRep, while layout must be obligatorily specified for each leaf node but is otherwise optional. If the picture feature is not present, it is assumed that its value is *nil.*

The sets of possible values are defined as follows:

Text-Level = {text, text-sentence, text-clause, text-phrase,nil[2]}

Layout = {chapter, section, paragraph, title, sub-title, vertical-list, list-item, caption, footnote, picture, wrapped-text}

Order = {1, ..., n, floating}

Picture = {<filename>, nil}

Order is either an integer, in which case lower numbered rhetorical structures must precede higher numbered siblings in the output text, or the atomic value floating, which is unordered with respect to other rhetorical structure[3]. Text-level and Layout have the values given above, which are self-explanatory. Picture has either a filename or nil as its value.

5.4 Surface syntaxes

A very simple document representation for the text discussed in Chapter 4 could look like figure 5.1. This exactly mirrors the (very simple) RhetRep, replacing the values at the nodes with its own feature-value pairs. These define text-level and layout for all nodes and, for the leaf nodes, position.

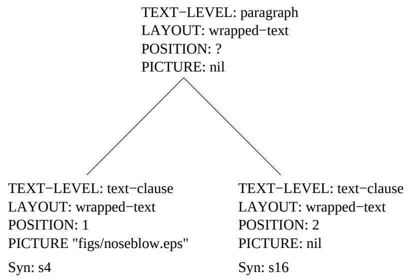


Figure 5.1: Representation for text structure of "Blow your nose so that it is clear"

The same information in RAGS object and arrow form looks, at first glance, to be rather intimidating. However, if we look closely, we can see that most of the extra complexity can be attributed to the expansion of sets of feature-attribute pairs on nodes to additional nodes

[3]Currently floating objects are completely unconstrained. In the future we may wish to introduce a notion of scope that constrains them, for example, to occur within the text of a specific rhetorical ancestor.

and arcs. We believe that the cost of this (in readability at least) is countered by the gain of a more uniform and thus more portable representation. Once we can see, in figure 5.2, that the three sets of pairs (*d1, d8, d14*) are simply the feature definitions, the rest of the tree can quite simply be viewed as representing the connection between the realisations of the leaves referring to the SynReps *s4* and *s16*.

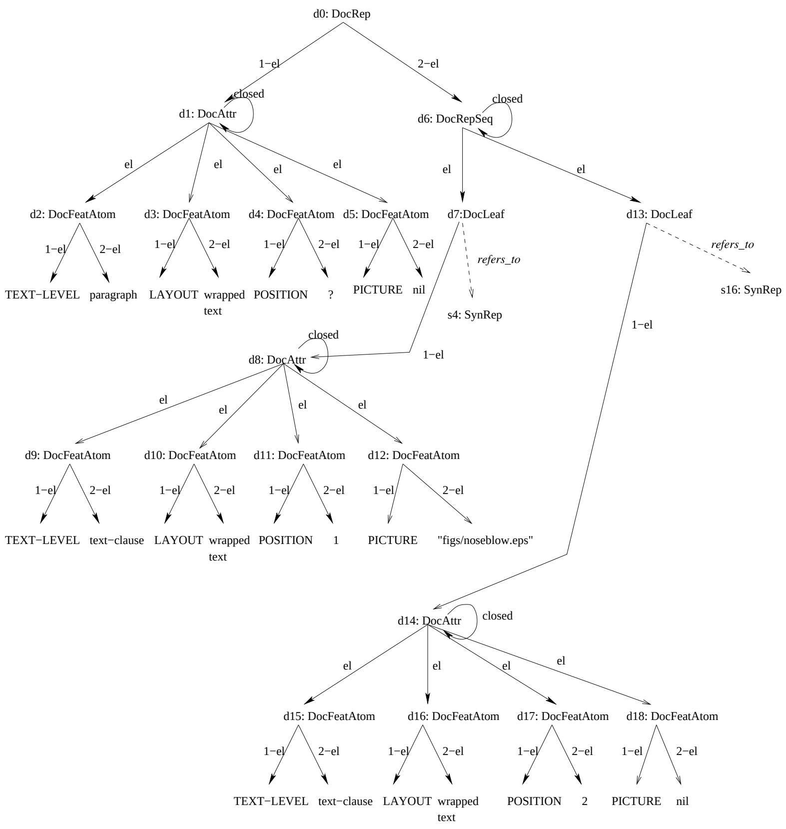


Figure 5.2: DocRep for "Blow your nose so that it is clear"

Chapter 6

Semantic Representation Datatype Specification

6.1 Executive summary

This level of representation can be used to encode both (when mixed with Conceptual Representations) atomic propositions of the kind found in the leaves of rhetorical structures (Sec. 9.6.2) and structured representations suitable for input to semantics-based realisers (for example, Penman [12] and KPML [2]) and as input to the translation to Syntactic Representations. (Cf. Ch. 7.) By "semantic information" we mean propositional content, or "ideational meaning" in the terminology of systemic grammar (see e.g., [1]). Semantic representations are representations of linguistic meaning structured in terms of *lexical semantic predicates* and semantic, i.e., thematic, *roles*. They contrast with syntactic representations, which are structured in terms of lexical *heads* and *grammatical functions* or *relations*. The scheme is based primarily on SPL [12], ESPL [27], Underspecification Discourse Representation Theory (UDRT) [24, 7] and Zeevat's Indexed Language (InL) [28].

6.2 Abstract type definitions

6.2.1 Formal definition

$$
\begin{aligned}
SemRep &= DR \times SemType \times SemAttr \\
ScopedSemRep &= DR \times SemType \times SemAttr \times Scoping \\
SemType &= 2^{SemPred} - \phi \\
SemAttr &= SemRole \rightarrow (SemRep \cup ScopedSemRep \cup DR \cup SemConstant) \\
Scoping &= ScopeConstr^{*} \\
ScopeConstr &= ScopeRel \times DR \times DR \\
DR &\in Primitives \\
SemConstant &\in Primitives \\
SemPred &\in Primitives \\
ScopeRel &\in Primitives
\end{aligned}
$$

6.2.2 Informal semantics of the representations

SemReps are triples consisting of a DR (discourse referent), a non-empty set of predicates and a function from *roles* to *arguments* (which are themselves semantic representations).[1] The arguments of *SemRoles* can be *DRs*, constants or other SemReps (both scoped and unscoped). Constants logically correspond to names of individuals whereas *DRs* are logical variables. The argument of a *SemRole* might be simply a *DR* for two reasons. First, the *DR* might correspond to a pronoun which is coreferential with some formula elsewhere in the *SemRep*. In this case, we wouldn't expect any content to be associated with the *DR*. Second, an intermediate *SemRep* might be missing content which is filled in by a later stage of processing but given the semantics of *SemReps* given above, it would be appropriate to substitute the *DR* of the missing content for that content.

That leaves only *ScopedSemReps* to explain and describe. (Complex) semantic representations (at every level) can include scoping information or not. The extra *Scoping* component of a scoped semantic representation is intended to express constraints on how subparts of that representation relate to others in terms of scope. Quantifier scope can be arbitrarily specified or left unspecified altogether in which case the fourth component of the ScopedSemRep is empty.

6.3 Example instantiations and surface syntaxes

Representations in the above abstract syntax can be translated into a number of more familiar notations, and indeed actual processing modules may operate on notations more oriented towards particular programming languages. It would be straightforward to specify a FUF-like notation, for example. In the following, however, we define and give an unscoped example in the original ESPL notation (taken directly from [27]) and a scoped example in an extension of that notation.

6.3.1 Example without Scoping

The following defines informally how the ESPL notation can be used for (unscoped) SemReps. Here Sem is used to refer to the disjunction of DR, SemConstant and SemRep.

Sem	$\Longrightarrow$	DR | SemConstant | SemRep
SemRep	$\Longrightarrow$	(DR / SemType Attribute*)
SemType	$\Longrightarrow$	SemPred | (SemPred$^+$)
Attribute	$\Longrightarrow$	SemRole Sem

Here is an example semantic representation in this notation:

[1] We intend "discourse referent" here in the sense of Discourse Representation Theory [11], that is, as a variable which denotes an individual (entitiy) or a set, group or collection of individuals (entities).


```
;;; If you got a letter addressed to you with this form, your National
;;; Insurance number is at the top of the letter.
;;;
(lc1 / Logical-Condition
     :domain ni-number-on-letter
     :range gets-letter
     :theme gets-letter)

(gets-letter
 / get
 :actor (h / hearer
          :recoverability recoverable)
 :actee (letter1
        / letter
        :identifiability nonidentifiable
        :actor (a1 / arbitrary)
        :modification (address1
                      / address-action
                      :time e<r
                      :actee letter1
                      :destination (h / hearer
                                    :recoverability recoverable)))
 :time e=r<s
 :inclusive (form1 /form
                   :deixis environmental))

(ni-number-on-letter
 / Spatial-Locating
 :domain (number1
         / number
         :modification (ni / national-insurance)
         :owned-by (h / hearer
                     :recoverability recoverable))
 :range (t1 / top
           :part-of (S1 / letter)))
```

6.3.2 Example with Scoping

Consider the sentence *Every man loves a woman* with *every* taking wide scope over *a* (ignoring tense). This can be represented in modified ESPL notation as in (1).

```
(1)  (e / love
      :agent (m / man :quant (q1 / every))
      :affected (w / woman :quant (q2 / a))
      ((> q1 q2)))
```

Compare this with what would be the corresponding unscoped representation using ESPL notation in (2).

```
(2)  (e / love
      :agent (m / man :quant (q1 / every))
      :affected (w / woman :quant (q2 / a)))
```

We can see that the only substantive difference between (1) and (2) is that in (1) the top-level (*Scoped*)*SemRep* has a fourth component which indicates that the quantifier named by `q1`, i.e., *every*, takes wide scope over the quantifier named by `q2`, i.e., *a*. More generally, the presence of triples $\langle Op, Var_1, Var_2\rangle$ indicates that the quantifier labelled by Var_1 stands in the scope relation Op to the quantifier labelled by Var_2. Absence of any such triple for a pair $\langle Var_1, Var_2\rangle$ indicates that the corresponding quantifiers are unscoped with respect to each other.

Finally, we give an indication of what the scoped example might look like in a FUF-like notation.

```
((DR "e")
 (PRED LOVE)
 (ROLES
  ((AGENT
    ((DR "m")
     (PRED "man")
     (ROLES
      ((QUANT
        ((DR "q1")
         (PRED "every")
         (ROLES ()))))))))
   (AFFECTED
    ((DR "w")
     (PRED "woman")
     (ROLES
      ((QUANT
        (DR "q2")
        (PRED "a")
        (ROLES ())))))))
   (SCOPING
    ((CAT LIST)
     (DISTINCT
      ((CAR
        ((OP ">")
         (OPER1 "q1")
         (OPER2 "q2")))
       (CDR NONE))))))))
```

6.3.3 Example in Objects and Arrows notation

Let us now consider the proposition, "blow your nose", (labelled r4 in the rhetorical example in chapter 5), This might be viewed in SPL-type notation as the following:

(r4/blow :actor(r8/addressee) :actee(r11/nose :owner(r8)))

In RAGS object-and arrow notation, this might look like figure 6.1. Although this appears

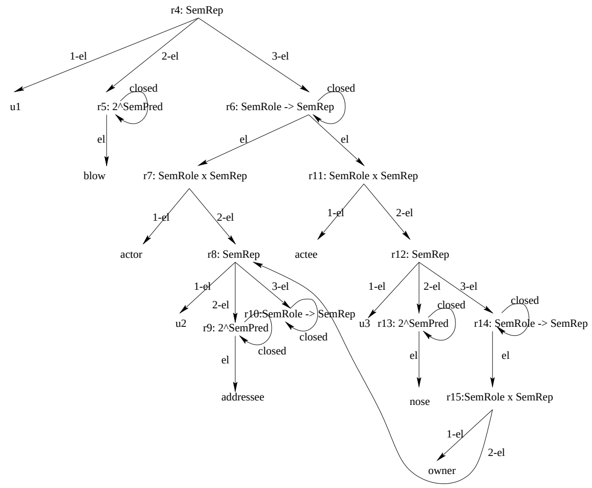


Figure 6.1: SemRep for "Blow your nose"

much more complex than the SPL representation, this cost is at the gain of a more universal notation that can more easily be ported for use in different systems. It also allows for more easy combination of the different levels of representation.

Chapter 7

Syntactic Representation Datatype Specification

7.1 Executive Summary

This level of representation is for planning the input to realisers which are based on some notion of *abstract* syntactic structure which does not encode surface constituency or word order distinct from *concrete* syntactic structure which does. The main point of comparison for this level is semantic representation (SemRep). Whereas SemRep organises information about the semantic translations of words and thematic roles, abstract syntactic representation (SynRep) organises it around words and grammatical functions as specified by heads, specifiers and adjuncts. SynRep is based on the D-structure of GB, the functional structure of LFG, the ARG(MENT)-ST(RUCTURE) of HPSG(4), Meteer's Text Structure and the *D(eep) Synt(actic) R(epresentation)s* of MTT.

7.2 Types and Operations

7.2.1 Abstract type definition

$$\begin{array}{rcl} SynRep &=& FVM \times (SynRep \cup Nil) \times SynArg \times Adj \\ FVM &=& SynFeat \rightarrow (SynAtom \cup FVM) \\ SynArg &=& SynFun \rightarrow SynRep \\ Adj &=& SynRep^* \\ Nil &\in& Primitives \\ SynFun &\in& Primitives \\ SynFeat &\in& Primitives \\ SynAtom &\in& Primitives \end{array}$$

7.2.2 Informal semantics of the representation

The atomic unit of abstract syntactic representations is simply a *feature-value matrix*, FVM, of the usual kind in this model. (Cf. [25].) We envisage that such FVMs might contain five types of information: (a) morpholexical class (and other "syntactic" information usually described as *head features*, see [8], [20] and [21]), (b) morphological markup suitable for input to inflectional morphology, (c) root, stem and orthography, (d) "semantic class" and (e) information for controlling realisation. Although these FVMs basically fulfil the role of *lexemes* in MTT (see [16]) we nonetheless recognise that practical systems will include realisation control information in these FVMs to choose amongst multiple possible realisations.

By "morpholexical class" we mean part of speech information like noun, verb, etc., and verb, noun or prepositional "form" (see [20] and [21]) such as base, infinitive, present participle, gerund, expletive *it* and presentational *there* in English and *to* (in the case of case-marking *to* in English). By "morphological markup suitable for input to inflectional morphology" we mean for example "third person, singular, feminine" or "strong (determiner class), second person singular". By "root" and "stem" we mean those terms as they are commonly used in morphology. By "semantic class" we mean for example object, event, process and culminated process.[1] By "information for controlling realisation" we mean for example "active" or "passive".

So the set of `SynFeat`s which are part of the XML equivalents of the FVMs might include `cat` for part of speech, `nform` for "noun form" and `pform` for "prepositional form" taking `SynAtoms` `n`, `v`, `a`, `p` and `there`, `it`, `norm` and `by`, `to`, etc. respectively. For morphological markup, we might have `SynFeats` `pers`, `num` and `gen` taking `SynAtoms` `1`, `2`, `3` and `sg`, `pl` and `mas`, `fem`, `neu` respectively. For root and stem we might have `SynFeats` `root` and `stem` taking `SynAtoms` `work-` and `worker-` respectively. For semantic class we might have `SynFeat` `semclass` taking `SynAtoms` `object`, `event`, `process`, etc. For information controlling realisation, we might have `SynFeat` `voice` taking `SynAtom`s `active` and `passive`.

Then a (non-atomic) *syntactic representation* (SynRep) is a tuple ⟨Head, Spec, Args, Adjs⟩ where Head is a feature-value matrix, Spec is the SynRep of the specifier of Head, Args is a function from grammatical functions to argument SynReps and Adjs is a tuple of adjunct SynReps. It is a tuple and not a set to accomodate those theories in which order of adjunction or modification is significant.

7.3 Example instantiations and surface syntaxes

7.3.1 Example instantiations of the interface

In this section, we present some simple mappings from SynReps to six levels of abstract syntactic representation in linguistic and NLG theories. These mappings are by no means meant to be definitive and are really illustrative only but we do believe that they do cover the major features of each level of representation. Following each mapping is a list of its limitations apparent at the time of writing.

It should be pointed out that the following mappings are, following standard RAGS methodology, *native*. That is, it is assumed that some aspects of the SynReps to be mapped are parameterised by the particular linguistic theory. In particular, the set of grammatical

[1] Our intention here is that "semantic class" should mean more or less the same thing as *expressive type* in Meteer's Text Structure [17]. We also note that such semantic classes can be as delicate as they need to be.

functions used are those given by individual linguistic theories and associated lexicons and grammars and are not fixed. This means that a sentence planner or microplanner has to plan SynReps using the given set of grammatical functions.

To simplify presentation in what follows, $f_1 \Longrightarrow f_2$ is used to indicate that the occurrence of f_1 in the right hand side of an equation should be textually substituted by f_2. In other words, if $X \Longrightarrow Y$ then $t = f(X)$ should be interpreted as $t = f(Y)$.[2] Prolog list notation is used to represent and construct phrase structure configurations. In particular, $[X\ Y_1\ \ldots\ Y_n]$ denotes a local tree with mother X and daughters $Y_1, \ldots, Y_n$ in that order. $[X \mid f(x)]$ denotes a local tree with mother X whose daughters are given by the value of $f(x)$. Finally, $cat(x)$ is the "category information" of a SynRep x.

7.3.2 GB

Here we present a mapping from SynReps to D-structure in GB [5].

Let $s = \langle \text{Head}, \text{Spec}, \begin{bmatrix} \text{ARG}_1 & A_1 \\ \vdots & \vdots \\ \text{ARG}_n & A_n \end{bmatrix}, \langle \text{Adj}_1, \ldots, \text{Adj}_n \rangle \rangle$ be a SynRep.

Then let

$$\begin{aligned} t_0 &\Longrightarrow [cat(\text{Head})\ \text{Head}\ t(A_1) \ldots t(A_n)] && (7.1) \\ t_1 &\Longrightarrow [cat(\text{Head})' \ldots [cat(\text{Head})\ t_0\ t(\text{Adj}_1)] \ldots t(\text{Adj}_n)] && (7.2) \\ t_2 &\Longrightarrow [cat(\text{Head})''\ t(\text{Spec})\ t_1] && (7.3) \end{aligned}$$

Then let

$$t(\langle \text{Head}, \text{Spec}, \begin{bmatrix} \text{ARG}_1 & A_1 \\ \vdots & \vdots \\ \text{ARG}_n & A_n \end{bmatrix}, \langle \text{Adj}_1, \ldots, \text{Adj}_n \rangle \rangle) = t_2 \qquad (7.4)$$

Then $t(s)$ is a mapping from SynRep s to its corresponding GB $\overline{\text{X}}$-theoretic D-structure.

Limitations of the mapping:

- The mapping assumes that subjects will not be included in the set of input arguments but rather be encoded as the input specifier. This is in accordance with the "native" strategy. Alternatively, the mapping could be redefined so that the input subject, when present as one of the input arguments, is mapped to the specifier.
- Nonlexical cases of functional categories need to be accommodated. This can be done by adding a special clause which adds them in to the mapping.

7.3.3 LFG

The following is a mapping from SynReps to f-structures in LFG [3].

[2] This is similar to the use of *macros* in programming languages.

Let $s = \langle \text{Head}, \text{Spec}, \begin{bmatrix} \text{ARG}_1 & A_1 \\ \vdots & \vdots \\ \text{ARG}_n & A_n \end{bmatrix}, \langle \text{Adj}_1, \ldots, \text{Adj}_n \rangle \rangle$ be a SynRep.

Then let

$$t(s) = \begin{bmatrix} \text{ARG}_1 & t(A_1) \\ \vdots & \vdots \\ \text{ARG}_n & t(A_n) \\ \text{PRED} & \text{Head}(\text{ARG}_1, \ldots, \text{ARG}_n) \\ \text{SPEC} & t(\text{Spec}) \\ \text{ADJ} & \{t(\text{Adj}_1), \ldots, t(\text{Adj}_n)\} \end{bmatrix} \qquad (7.5)$$

Then $t(s)$ is a mapping from the SynRep s to its corresponding LFG f-structure.

Limitations of the mapping:

- Cases where tense and determiners end up as values of TENSE and DEF are not handled. These are subcases of the more general issue of the LFG theory of function words which is not addressed here. In any case, such information would almost certainly be encoded in the lexical head in the SynRep and it would be straightforward to adjust the mapping so that it adds in that information under the appropriate features when it is present on the head in the SynRep.
- The mapping is more general than classical LFG in its treatment of specifiers but this issue is really dependent on whether tense and definiteness (for example) are encoded on lexical heads in the SynRep or as specifier SynReps. See the comment above.
- All grammatical functions are assumed to be *semantic* arguments as well as subcategorised but the mapping could be easily adjusted, for example, for raising verbs which subcategorise for a subject but which do not assign it a thematic role.

7.3.4 HPSG4

Here we present a mapping from SynReps to *extended* ARG-ST (i.e., with adjuncts) of HPSG4 [14].

Let $s = \langle \text{Head}, \text{Spec}, \begin{bmatrix} \text{ARG}_1 & A_1 \\ \vdots & \vdots \\ \text{ARG}_n & A_n \end{bmatrix}, \langle \text{Adj}_1, \ldots, \text{Adj}_n \rangle \rangle$.

Then let

$$t(s) = t(\text{Adj}_n)(\ldots(t(\text{Adj}_1)(\text{Head}(t(\text{Spec}), t(A_1), \ldots, t(A_n))))) \qquad (7.6)$$

Then $t(s)$ is a mapping from the SynRep s to its corresponding HPSG4 ARG-ST.

Limitations of the mapping:

- Nothing special is done about lexical entries which, for example, subcategorise for a nominative NP and a VP which subcategorises for the NP as well (without realising it). The SynReps for the two NPs are simply identical. This issue however depends on what assumptions one is making and which version of ARG-ST one is using. We simply note here that this treatment might have to be adjusted for a given set of assumptions.

7.3.5 Categorial grammar

The following shows how SynReps can be mapped into the functor-argument structures used in versions of Categorial Grammar (e.g. [26], [29]).

Let $s = \langle \mathrm{Head}, \mathrm{Spec}, \begin{bmatrix} \mathrm{ARG}_1 & A_1 \\ \vdots & \vdots \\ \mathrm{ARG}_n & A_n \end{bmatrix}, \langle \mathrm{Adj}_1, \ldots, \mathrm{Adj}_n \rangle \rangle$ be a SynRep.

Then let

$$t(s) \quad = \quad t(\mathrm{Spec})(t(\mathrm{Adj}_n)(\ldots(t(\mathrm{Adj}_1)(\mathrm{Head}(t(A_1), \ldots, t(A_n)))))) \tag{7.7}$$

and rewrite $X(Y_1, \ldots, Y_n)(Z)$ as $X(Z, Y_1, \ldots, Y_n)$. Then the rewriting of $t(s)$ is a mapping from the SynRep s to its corresponding functor-argument structure.

7.3.6 Text Structure

The following indicates how SynReps correspond to the boxes used in Meteer's Text Structure [18].

Let $s = \langle \mathrm{Head}, \mathrm{Spec}, \begin{bmatrix} \mathrm{ARG}_1 & A_1 \\ \vdots & \vdots \\ \mathrm{ARG}_n & A_n \end{bmatrix}, \langle \mathrm{Adj}_1, \ldots, \mathrm{Adj}_n \rangle \rangle$ be a SynRep.

Then let [A B C] represent a Text Structure box $\boxed{\begin{array}{c} \mathrm{A} \\ \mathrm{B} \\ \mathrm{C} \end{array}}$ and [A B] represent a Text Structure box $\boxed{\begin{array}{c} \mathrm{A} \\ \mathrm{B} \end{array}}$. Then let

$$\begin{aligned}
t_0 \quad &\Longrightarrow \quad [\mathrm{Head\ HEAD} && (7.8) \\
&\qquad\quad [\mathrm{ARGUMENT} \mid t(A_1)] \\
&\qquad\quad \ldots \\
&\qquad\quad [\mathrm{ARGUMENT} \mid t(A_n)]] \\
t_1 \quad &\Longrightarrow \quad [\mathrm{COMPOSITE} \\
&\qquad\quad [\mathrm{MATRIX} \mid t_0] \\
&\qquad\quad [\mathrm{ADJUNCT} \mid t(\mathit{Spec})] \\
&\qquad\quad [\mathrm{ADJUNCT} \mid t(\mathrm{Adj}_1)] \\
&\qquad\quad \ldots \\
&\qquad\quad [\mathrm{ADJUNCT} \mid t(\mathrm{Adj}_n)]] \\
t_2 \quad &\Longrightarrow \quad [\mathrm{TOP} \mid t_1] && (7.9) \\
t(s) \quad &= \quad t_2 && (7.10)
\end{aligned}$$

Then $t(s)$ is a mapping from the SynRep s to its corresponding text structure.

Limitations of the mapping:

- As defined here, SynRep only encodes the abstract syntactic aspect of Text Structure. For a proposal which also incorporates Text Structure's semantic aspects, see the next section.

7.3.7 MTT

The following mapping translates SynReps into the DSyntR's of MTT [16].

Let $s = \langle \mathrm{Head}, \mathrm{Spec}, \begin{bmatrix} \mathrm{ARG}_1 & A_1 \\ \vdots & \vdots \\ \mathrm{ARG}_n & A_n \end{bmatrix}, \langle \mathrm{Adj}_1, \ldots, \mathrm{Adj}_n \rangle \rangle$ be a SynRep.

Then let

$$t(s) \quad = \quad [\mathrm{Head}, \left\{ \begin{array}{c} \langle \mathrm{SPEC}, t(\mathrm{Spec}) \rangle \\ \langle G_1, t(A_1) \rangle \\ \vdots \\ \langle G_n, t(A_n) \rangle \\ \langle \mathrm{ATTR}, t(\mathrm{Adj}_1) \rangle \\ \vdots \\ \langle \mathrm{ATTR}, t(\mathrm{Adj}_n) \rangle \end{array} \right\}] \tag{7.11}$$

Then $t(s)$ is a mapping from the SynRep s to its corresponding MTT DSyntR.

Limitations of the mapping:

- It is unclear to us how MTT handles arbitrary specifiers. The mapping might need to be adjusted given a particular set of assumptions.

7.3.8 Example surface syntax

In this section we present a concrete Prolog syntax for the abstract type definition.

Example using DCG

In the first, a modified form of Definite Clause Grammar notation is used in which disjunction is indicated by a semicolon, nonterminals by an uppercase-initial alphabetic string surrounded by '<>', '<atomic>' indicates a Prolog atomic and standard Prolog term syntax is used otherwise.

7.3.9 Definition

```
<SynRep>    --> SynRep(head(<Head>),spec(<Spec>),args(<Args>),adjs(<Adjs>)).
<Head>      --> <FVM>.
<FVM>       --> [] ; [<Feature>=<Value> | <FVM>].
<Feature>   --> <atomic>.
<Value>     --> <FVM> | <atomic>.
<Spec>      --> [] ; <SynRep>.
<Args>      --> {} ; { <A> }.
<A>         --> <B> ; <B>, <A>.
```

```
<B>        --> arg(fun(<Fun>),arg(<SynRep>)).
<Fun>      --> <atomic>.
<Adjs>     --> [] ; [<SynRep> | <Adjs>].
```

`<SynRep>` is the nonterminal for SynReps. An `<SynRep>` has four subterms, one each for the head lexeme (`<Head>`), the specifier SynRep (`<Spec>`), the set (possibly empty comma-separated list surrounded by curly brackets) of grammatical function/argument SynRep pairs (`<Args>`) and the sequence (possibly empty list) of adjunct SynReps (`<Adjs>`). The head subterm is a lexeme specification which is simply a feature-value structure (`<FVM>`) with the concrete syntax of the Eisele-Dörre Prolog encoding.

A specifier specification is either the empty list, indicating that there is no specifier, or a SynRep specification (`<SynRep>`).

A grammatical function/SynRep pair has two subterms, the grammatical function itself (`<Fun>`) and the actual argument specification (`<SynRep>`).

The list of adjunct SynReps requires no comment.

7.3.10 Example

```
John saw the moon tonight.

SynRep(
 head([root=see]),
 spec([]),
 args(
  {arg(fun(subj),
      SynRep(
       head([root=John]),
       spec([]),
       args({}),
       adjs([]))),
   arg(fun(obj),
      SynRep(
       head([root=moon]),
       spec(
        SynRep([root=the]),
         spec([]),
         args({}),
         adjs([]))),
       args({}),
       adjs([]))}),
 adjs(
  [SynRep(
    head([root=tonight]),
    spec([]),
    args({}),
    adjs([]))])).
```


Example using Objects and Arrows

The syntactic representation of the first proposition of the example in Chapter 5 ("blow your nose") can be represented in RAGS object-and-arrow format as figure 7.1. Here we can see

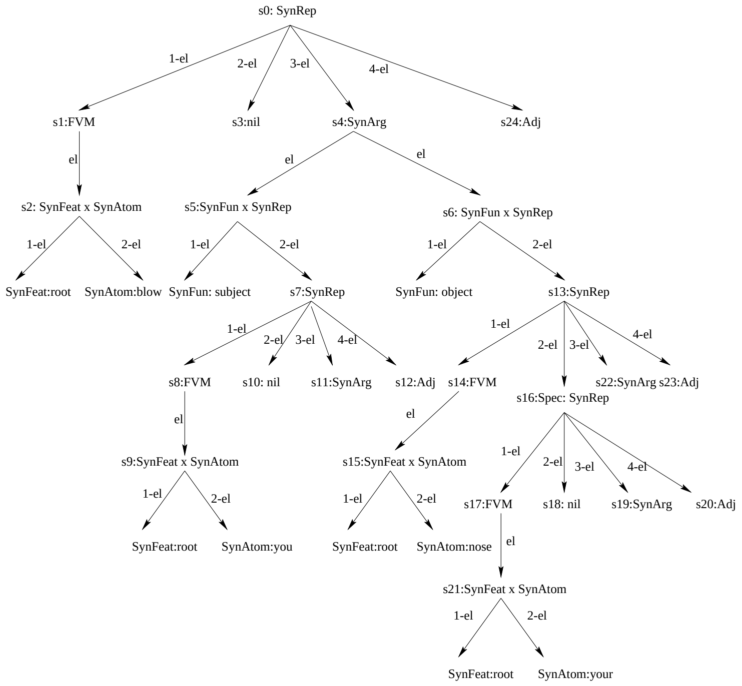


Figure 7.1: SynRep for "blow your nose"

that a SynRep is a tuple whose first element is a Feature Value Matrix (FVM), which here defines the root to be *blow*. The second element is a Specifier, here nil, and the fourth, Adjuncts, also nil. The third element is the sequence of SynReps for the arguments of the head. In this case, these consist of the SynReps for "you" and "your nose". The second of these has a specifier, itself a SynRep. This SynRep simply has the head's root defined as "your". Thus, "your" is the specifier for "nose".

Chapter 8

Quote Representation Datatype Specification

8.1 Executive Summary

The *Quote* level of representation allows for the referencing of fixed pieces of output (speech, text, graphics) which are being considered for inclusion (without change) into the output of the NLG system. It is important to realise that Quote only refers to canned material ready for the "output" stage of generation. Other kinds of canned material (e.g. canned syntactic or semantic representations) can be represented by the other levels without any changes (in general, "canned" indicates that a complex representation has been retrieved as a single item rather than constructed from first-principles – this may have implications for how it is to be used. There is no inherent reason why "canned" representations should however look any different in shape to "non-canned" ones.)

Where the fixed output can be construed as a phrasal lexical item of some kind, an alternative possibility would be to have a syntactic representation which is unspecified apart from the head lexical item. Such a representation would, however, tend naturally to produce output that varies according to context (i.e. inflected forms). Quote is intended for material that can be guaranteed never to change.

RAGS does not take a position on what sorts of material might be 'Quote'd. This is really just a placeholder allowing such material to be mixed with other representations.

8.2 Abstract Type Definitions

Since RAGS has nothing to say about the internal structure of a quote (and the NLG system will not in general consider it to have any), *Quote* is a Primitive type:

$$Quote \quad \in \quad Primitives$$

8.3 Example Instantiations and Surface Syntaxes

For textual material, the subtype of a Quote could provide the text directly. In many programming languages, this would be represented simply by a string. Alternatively, the subtypes

could correspond to names of files, where the extension of the file indicates the type of material involved.

8.4 Justification and Motivation

It could be argued that when an NLG system uses a piece of fixed output there is no need to represent this apart from actually producing this output. However, when output is assembled from a mixture of fixed and generated material, it is useful for a system to be able to represent where the fixed material is to appear. A template, for instance, can be viewed as a mixed representation with some fixed parts and some other parts that will be filled in by "first principles". See section 9.6.3 for an example.

Chapter 9

Data Representation

The abstract type definitions given in the previous chapters only give a high-level view of what RAGS representations should be like. There are a number of deeper assumptions about the nature of data that can only be spelled out at a level closer to actual implementation. We present here a "reference implementation" of RAGS representations, called the "objects and arrows" model. This is used primarily to make precise notions of partiality and mixed representation which are important for practical use of the RAGS representations. This model could be used quite directly as the basis of an actual implementation, but there are many other ways in which the ideas could be faithfully implemented. The intention here is not to suggest a particular implementation strategy but to provide a reference point with which any proposed implementation can be compared in order to assess whether it is faithful to the RAGS ideas. In particular, the mapping from RAGS representations into XML (Appendix B) has to allow the right distinctions to be captured.

It is important to emphasise that RAGS only has something to say about the types of data that are actually communicated between modules. How a RAGS module organises its private data is its own business.

9.1 The "Objects and Arrows" Model

The "objects and arrows" model implements complex data via a directed graph structure (which can have multiple roots and cycles) of nodes and arrows. The graph has:

- *objects* (nodes) which represent (possibly primitive) structures and substructures. Each object is labelled with a type (one of the RAGS types defined in the previous chapters). In addition, objects with primitive types can be labelled with theory-specific subtyping information.

- *arrows* between the objects. Each arrow has a unique source and target object, as well as an (atomic) label. Arrows are typed, in that for each arrow label the possible types of the source and target are restricted.

In the objects and arrows model, a complex structure is modelled by an object which has (labelled) *component* arrows leading to other nodes which represent its parts. Arrows labelled *el* go from objects with set types to objects with the appropriate type for the set elements; arrows labelled *1-el*, *2-el*, etc go from objects with tuple or sequence types to objects with the

appropriate type for the 1st, 2nd, etc. element of the tuple or sequence. This is illustrated in more detail below. With the exception of any *closed* arrows (discussed below), the set of objects and arrows representing a complex structure and its components is a directed acyclic graph (cycles are not allowed but reentrancy is). Because the objects are typed, it is possible to check that only structures compatible with the RAGS definitions are built.

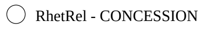
This basic notion will be embellished slightly in the following sections. First of all, we summarise how RAGS representations of different types are "implemented" by objects and arrows.

9.1.1 Primitive Types

This covers types that are defined in the following way:

$$RhetRel \in Primitives$$

An instance of this type is represented by a single object, labelled with a combination of the type and subtype information about it. So the following represents an instance of the rhetorical relation CONCESSION:

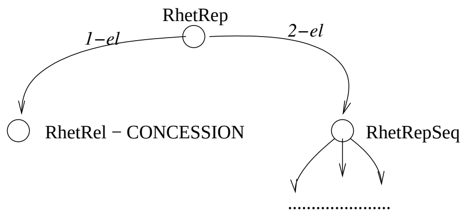


9.1.2 Tuple/Product Types

This is for types defined in terms of a fixed number of components, for instance as in:

$$RhetRep = RhetRel \times RhetRepSeq$$

An instance of such a type gives rise to a single object, whose components are specified by arrows labelled *1-el*, *2-el*, etc:

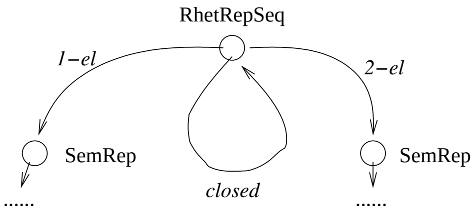


9.1.3 Sequence Types

This is for types where the order of components matters but the number is not known in advance, for instance:

$$RhetRepSeq = (RhetLeaf \cup RhetRep)^+$$

(similarly for types defined by X^*). The implementation of components is as for tuple types. The only difference is that it is necessary to be able to distinguish whether *all* the components are indicated by the current arrows or whether only some of them are indicated (see section 9.2 below). A *closed* arrow from the object to itself indicates that all the components are present:



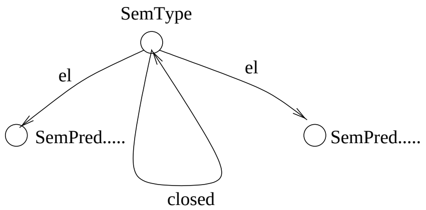
The *closed* arrow would be omitted if one wished to represent a sequence with at least 2 elements (but possibly more), say.

9.1.4 Set Types

This is for types where the order of components does not matter and the number is not known in advance, for instance:

$$SemType = 2^{SemPred} - \phi$$

(similarly for types defined by 2^X). These are represented in the same way as sequence types (including the possible use of a *closed* arrow), except that all components are indicated by *el* arrows:



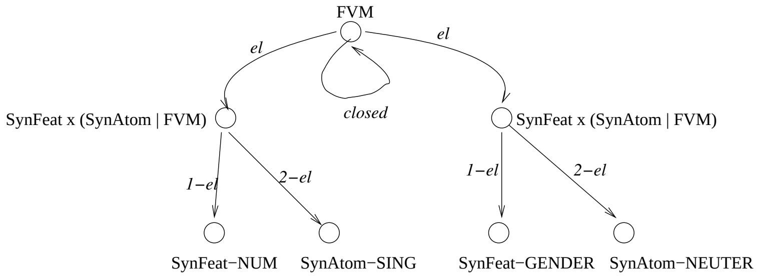
Note that it makes no sense for there to be two *el* arrows from an object to the same other object.

9.1.5 Functional Types

This is for types defined as mappings from one type to another, for instance:

$$FVM = SynFeat \rightarrow (SynAtom \cup FVM)$$

An instance of such a type (a function from X to Y) is basically represented as a set of pairs in the product $X \times Y$:



This represents what in some programming language might be expressed by a notation like `(NUMBER=SING, GENDER=NEUTER)`. As with all sets, a *closed* arrow can be provided from the object to itself. An implementation would ideally check that no more than one element of the X set with a given subtype occurs as the first element of a pair. Standard names for the types of the extra $X \times Y$ objects are given in Chapter 10 and it is suggested that these are used.

Note that there is no way to represent disjunctions – this is consistent with the idea that the objects and arrows model is for representing actual data (even if that may sometimes be incomplete).

9.2 Partiality

NLG systems should be able to exchange data representations that are only partially complete. This section describes what is allowed in terms of objects and arrows. Basically, a representation can be partial in that it has a RAGS type but not all of the arrows have yet been provided. This possibility was referred to above in connection with sequences and sets.

The following is an abstract characterisation of the set of (possibly partial) objects and arrows structures that are allowed. The idea is that any allowed representation should be reachable from an empty graph by applying the following operations (in any order and number):

1. **Adding a new object -** Adding one object (not connected in any way) with its RAGS type. This should be an explicitly defined or primitive type (not, for instance, one of the product types used in the implementation of function argument-value pairs (see 7 below)).

2. **Instantiating a primitive -** Adding subtyping information to an object with a primitive type. Note that the RAGS model regards this as a unitary operation as it has no knowledge of any internal structure to the subtyping information.

3. **Adding an element to a set -** Adding an *el* arrow from a non-*closed* object with a set type to an existing object with an appropriate type where there is not already an *el* arrow between these objects. This must not introduce a cycle of local arrows in the graph.

4. **Adding an element to a sequence or tuple -** Adding an *n-el* arrow from a non-*closed* object with a sequence or tuple type to an appropriate existing object. There must not

already be any *n-el* arrows with the same n from the first object. This must not introduce a cycle of local arrows in the graph.

5. **Closing a set -** Adding a *closed* arrow (which is not already there) from an object of type set to itself, as long as this set cardinality is legal by the RAGS definitions.

6. **Closing a sequence -** Adding a *closed* arrow (which is not already there) from an object of type sequence to itself, as long as this sequence cardinality is legal by the RAGS definitions and has a complete set of *n-el* arrows from 1 up to some number (and no others).

7. **Adding an element to a function -** Adding an *el* arrow from an object with a function type (but with no *closed* arrow) to a new object (with the appropriate product type), at the same time adding *1-el* and *2-el* arrows from that object. The first of these goes to an existing object with the appropriate primitive type and with specified subtype; the second goes to an appropriate existing object. There must not already be a similar structure in place for a primitive object with the same subtype. Also this must not introduce a cycle of local arrows in the graph.

8. **Closing a function -** Adding a *closed* arrow (which is not already there) to an object of functional type.

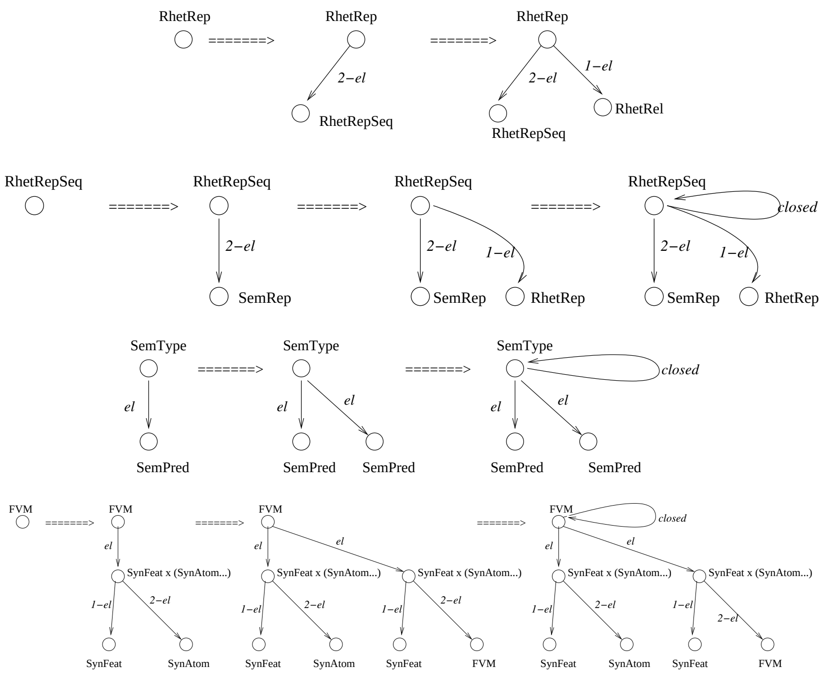
It is important to make three points about this definition:

1. In the above, "appropriate" means "legal according to the RAGS type definitions". That is, component arrows are only allowed between certain types and in general this has to be checked in an implementation. In general a complete implementation would also check for lack of cycles with local arrows, though the expense of this would not always be justified.

2. There is no suggestion that an implementation need only ever deal with the above states or need create them in this way. The above is purely a formal way of describing the states that can be communicated between RAGS modules.

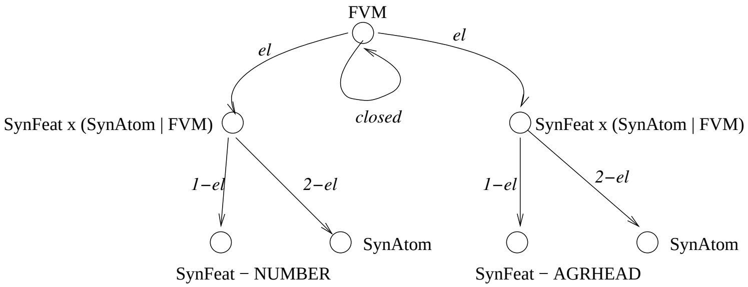
3. Adding an *el* or *n-el* arrow from an object that already has a *closed* arrow is non-monotonic, and so this operation would not be supported in modules that wish to respect decisions that have been made earlier.

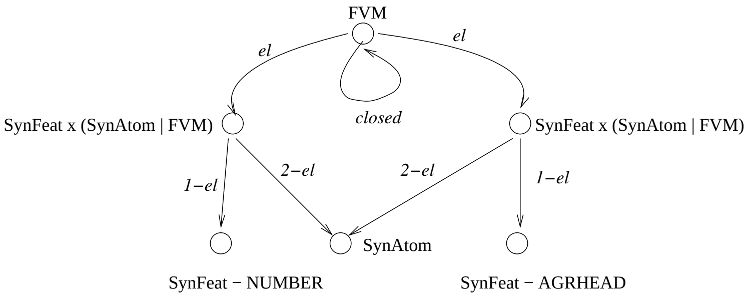
The following figure shows typical stages in the "lifetime" of objects of various types. These show the content of an object being built up slowly from completely unspecified to fully specified (at the top level). In practice, many objects in RAGS representations will be completely formed when they are first communicated between modules. Also the content of an object's components can build up progressively or all at once, independently of that for the object itself.



9.3 Re-entrancy

The above definition of the (partial) representations that are allowed allows for graphs to be constructed which are not trees. The following figure, for instance, shows two versions of a RAGS representation which differ not in the the types of the objects or the arrows that are present, but according to whether two arrows end up at the same object or not (whether there is *reentrancy* in the graph or not).





The RAGS position is that the distinction illustrated in the figure is a potentially important one and that implementations of the RAGS representations must allow it to be expressed and detected. In this case, the first graph represents an FVM where there are values for the features NUMBER and AGRHEAD but those values are not yet known. The second represents this plus the information that these values are the same.

It is currently unclear what meaning is to be read into the distinction between there being two separate (identical in structure) objects and one single object if the objects concerned are fully specified. For now, RAGS assumes that this is a valuable distinction and allows it to be captured.

9.4 Mixed Representations

At an intermediate state in an NLG system, it is likely that partial representations at a number of levels have been built and these different representations correspond to one another in systematic ways. It should be possible for researchers using the RAGS architecture to exchange datasets that involve more than one level of representation at a time.

The first measure for allowing mixed representations is the fact that the data that one RAGS module communicates to another can correspond to any objects and arrows graph even if that graph is not completely connected[1]. The second measure is the allowing of *non-local* arrows between representations of the same and different kinds.

[1]This of course allows other things, such as multiple representations at the same level, to be communicated.

Whereas component arrows represent standard notions of constituency of structures, non-local arrows are intended to be used to indicate relations between structures, in particularly relations that are relevant to the evolution of data as the generation process proceeds. In particular, note that non-local arrows are able to introduce cycles (whereas non-closed local arrows cannot do this on their own). The set of non-local arrow types is open-ended in order to allow for distinctions relevant to particular processing strategies. A RAGS system chooses the arrow types to be used in the same way as it for instance chooses how to instantiate the sets involved in semantic representations. However, the set of non-local arrows must include the following types (which does not mean that every system has to use them):

realised_by - Indicates that the destination object (and all its components) is of a type arising "later" in the generation process but corresponds to one way that the source object (and all its components) could be realised. In particular, the set of documents (or parts of documents) that the destination could eventually give rise to is a (not necessarily proper) subset of the set of documents (or parts) that the source could give rise to. For this purpose, the following ordering of representation levels is used:

KBId < SemRep
KBId < DR
KBId < ScopedSemRep
KBId < SemConstant
RhetRep < DocRep
RhetRep < SynRep
SemRep < SynRep
DR < SynRep
ScopedSemRep < SynRep
SemConstant < SynRep
SynRep < Quote

There is currently some discussion about whether this < relation should be considered to be transitive.

revised_to - Indicates that the destination object is of the same type as the source object and that it is believed that the former (with all its components) is a preferred representation to the latter (and all its components).

refers_to - Indicates that a leaf at one level of representation corresponds to a representation at another level. This is intended to be used only for the following situations:

RhetLeaf → SemRep or ScopedSemRep or DR or SemConstant
DocLeaf → SynRep or Quote

The existence of non-local arrows means that another case needs to be added to the set of operations characterising legal objects and arrows structures:

9. Adding a non-local arrow - Adding a local arrow between two existing appropriate objects.

The use of non-local arrows is best illustrated by the examples in section 9.6.

9.5 Generic Sequences and Atoms

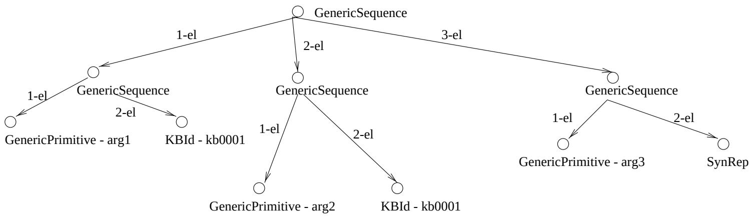
There is one further way that mixed RAGS representations can be built up, and this is intended simply as a way of creating complex inputs or outputs to be passed between modules. As things stand, it is easy for one module to send another an unstructured set of RAGS representations – the default assumption embodied by the objects and arrows model is that one has a simple set of representations. Sometimes, however, one module needs to send another module something with more structure to it, for instance an ordered list of possible SemReps associated with numerical confidence scores or the combination of two KBIds and a SynRep. In such a case, it is useful to have some kind of wrapper to put around the RAGS representations to indicate clearly what the role of each one is in the interface.

For the construction of such wrappers, the following extra type definitions apply:

$$\begin{aligned} \mathit{GenericPrimitive} &\in \mathit{Primitives} \\ \mathit{GenericSequence} &= (\mathit{GenericPrimitive} \cup \mathit{GenericSequence} \cup \mathit{RhetRep} \cup \ldots)^* \end{aligned}$$

where the union is over all RAGS types. The type GenericPrimitive is intended to be used for simple ways of labelling RAGS representations, for instance numbers or argument role names. This is a primitive type and can have subtypes as one wishes. GenericSequence is a simple universal way of combining together multiple structures (as, perhaps, with lists in LISP).

For instance, here is an example of how one might wrap up two KBIds (with different roles in the interface) and a SynRep into a single unambiguous structure to be passed from one module to another:



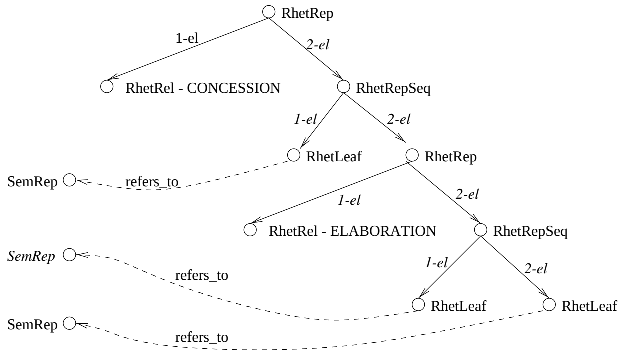
Here, each RAGS representation has been bundled together with a GenericPrimitive indicating its role in the interface (arg1, arg2 and arg3). These are then grouped into a single GenericSequence holding the whole structure together. In fact, in this example, the roles of the three RAGS representations are also indicated by their position in the overall GenericSequence, so the explicit labelling as "arg1", "arg2" etc is not really needed (though might be useful if these names had some mnemonic significance).

Note that in this subsection we are talking about mechanisms that are only used to bundle together RAGS representations into larger structures for passing around unambiguously. These could be considered outside the main definitions of RAGS and entail no changes to what SynReps, Quotes etc can be. No non-local arrows are allowed to connect GenericSequences or GenericPrimitives.

9.6 Examples

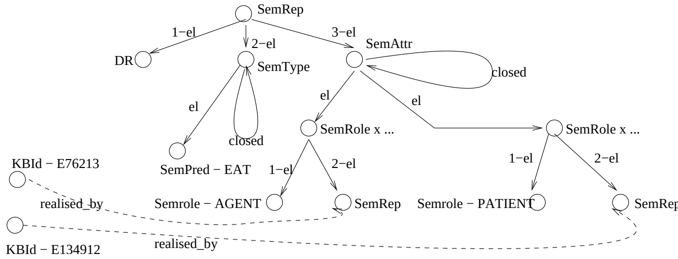
9.6.1 Mixing RhetReps and SemReps

Associating SemReps with RhetLeafs by `refers_to` arrows is intended to be a standard way of making a mixed representation. (Similarly for associating SynReps with DocLeafs). The following shows what this might look like:



9.6.2 Abstract Semantics

During the development of semantic representation, it may be useful to communicate a stage where the "top level" structure of the semantics is established but the lower level detail is not yet (e.g. because referring expression generation has not yet taken place). For bookkeeping purposes, the association between the "holes" in the evolving semantic representations and the conceptual entities that they will realise needs to be recorded. This is how this idea could be captured via a mixed representation:

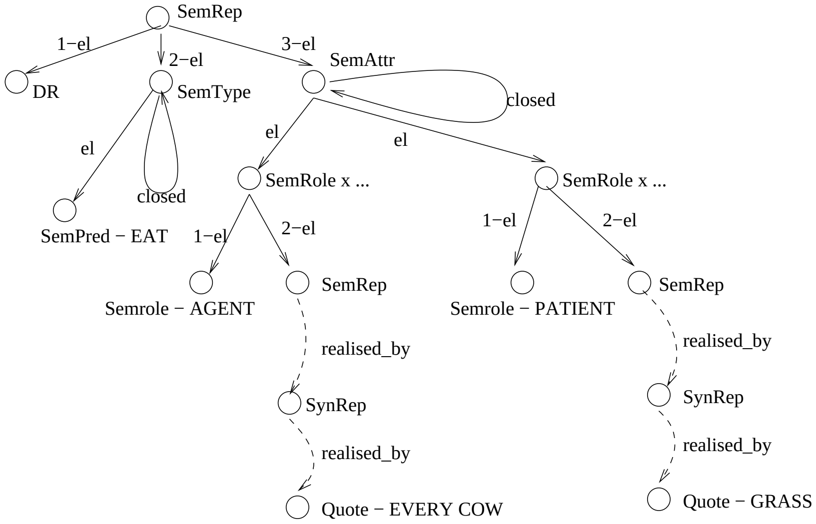


Here **realised_by** arrows are used to connect the KBIds to the (so far unspecified) semantic representations that correspond to them.

In the past, RAGS specified a distinct level of representation for semantic predicates with conceptual role fillers ("abstract semantics"). There is now no need for that extra level, as it can be captured by a mixed conceptual-semantic representation.

9.6.3 Introducing Canned Material

The following is an example of how canned material can be related to other representations by *realised_by* arrows.

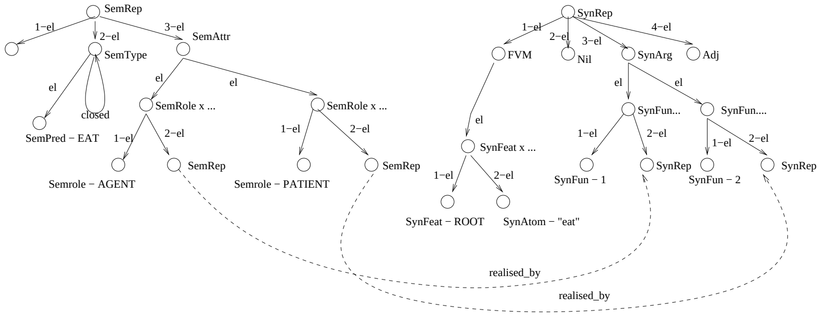


It shows a semantic representation where two role fillers (whose detail is not shown) are associated with canned material to realise them. In this example, the content of the role fillers is probably irrelevant (and not worth further determining if it is partial), as the existence of the *realised_by* arrows represents a strong suggestion to use fixed phrases instead of proceding by a more first-principles approach through syntax.

Similarly, canned material could be related to at least rhetorical and abstract syntactic representations (or their parts).

9.6.4 Simple Lexical Entry

The following shows a way of representing lexical knowledge that would yield a more principled realisation of the semantic representation of the last example. It relates a skeletal semantic representation with a skeletal syntactic one, with *realised_by* arrows linking the corresponding subparts.



Here the semantic roles AGENT and PATIENT have been associated with the syntactic argument roles 1 and 2. This mixed semantic-syntactic structure is close to what is called a *sign* in a number of Linguistic theories.

Chapter 10

XML Encoding

Although RAGS representations will often be implemented by programming language specific mechanisms, when they are communicated between different programming languages or machines it is useful for there to be a standard interchange format. This chapter describes such a format using XML. It does this mainly by presenting the DTD for RAGS documents (which specifies the set of allowed XML forms). This has parts for all the RAGS data representations, these echoing in a consistent way the abstract type definitions and being consistent with the "objects and arrows" model of Chapter 9.

The form of the DTD reflects the following principles:

Use of XML elements. XML elements usually correspond one-to-one with objects in the "objects and arrows" model and the element names are the RAGS type names. For functional types, the "objects and arrows" model specifies the existence of objects standing for "function and value" pairs. The types of these are assigned XML element names as follows:

Type name	XML name
$DocFeat \times DocAtom$	`DocFeatAtom`
$SemRole \times (SemRep \cup ScopedSemRep \cup DR \cup SemConstant)$	`SemRoleRep`
$SynFeat \times SynAtom$	`SynFeatAtom`
$SynFun \times SynRep$	`SynFunRep`

Primitive types. Primitive types whose subtypes depend on the details of the theory being used (e.g. semantic roles, predicates) are here represented by XML EMPTY elements with the attribute `name` used to provide the subtype information. If some theory requires instances of such a type to have internal structure then the corresponding parts of the DTD should be amended.

Reentrancy. Elements that correspond to objects that can be sources of reentrancy can have an attribute `ident` which gives them a unique identifier. This allows us to represent structures with reentrancy. The XML ID/IDREF mechanism is used and so XML-validation will detect dangling pointers etc. These identifiers must be legal XML names. Where there is reentrancy, exactly one element representing an object is represented using the standard arrangement for its type and with the attribute `ident` set to the objects's identifier (an ID attribute for these element types). All other occurrences of this object are represented by XML empty elements of type `XREF`, with an `idref`

attribute (an IDREF atttribute for `XREF`) holding the identifier. `XREF` is defined as follows:

```
<!ELEMENT XREF EMPTY>
<!ATTLIST XREF idref IDREF #REQUIRED>
```

Wherever an object type can appear in an XML element, an `XREF` is able to appear as well.

Placeholders. Where in a tuple or sequence the objects and arrows model does not specify an arrow for some particular component, in the XML the corresponding position is filled by a placeholder element `T`, which is defined as follows.

```
<!ELEMENT T EMPTY>
```

Partial representations. Partial structures need to be allowed for at all levels. I.e. an XML structure should not be illegal because it is partial. This entails the DTD being more generous than one would expect in some places.

Type explicitness. In a simple encoding, one can't always infer the arrow types for nested structures without having access to the type definitions. This could be awkward for procedures that translate from XML into programming language-specific notations. The solution to this is to have more information about the types explicit in the XML. Thus for each non-primitive type, the corresponding XML element must have an attribute `type` defined whose value is one of `set`, `sequence`, `tuple` and `functional`. For `tuple` types (used for tuples of fixed lengths) the additional `length` type specifies the length. The DTD forces the values to be the correct ones by specifying enumerations which consist of single values. The attribute is made `#REQUIRED` to force it to be explicitly stated.

External references. In order to allow at some point for references to objects defined externally to the document, we introduce an element `EXTREF`. These must be free-standing elements (not nested inside other ones) and can be referenced by `XREF`s in the standard way (and arrows, see below). `EXTREF`s will have compulsory `url` attributes, but exactly what we will use there remains to be decided later. `EXTREF` is defined as follows:

```
<!ELEMENT EXTREF EMPTY>
<!ATTLIST EXTREF ident ID #REQUIRED
                 url CDATA #REQUIRED>
```

The `closed` arrow. Presence of a `closed` arrow from an object to itself is represented by the value `yes` to the attribute `closed`. This attribute can be given no other value – leaving the attribute unspecified amounts to saying that the set is not closed.

Other arrows. There are two ways that an arrow in the "objects and arrows" model can be expressed – "inline" where the destination of the arrow is embedded inside the source element and "out of line" with seperate XML elements. Arrows indicated "out of line" are empty elements which refer to the `ident`s of the source and target via `IDREF` attributes:

```
<!ELEMENT arrow EMPTY>
<!ATTLIST arrow name CDATA #REQUIRED
                source IDREF #REQUIRED
                target IDREF #REQUIRED>
```

Non-local arrows *must* be specified by separate arrow elements. The DTD for the RAGS representations do not allow these to be embedded. On the other hand, each local arrow can be expressed either inline or (if the name can be expressed as the value of the `name` attribute) by a separate `arrow` element. Note that each arrow must be specified only once. Also, `closed` arrows should never be indicated "out of line".

Generic Sequences and Primitives. These are defined as follows:

```
<!ELEMENT GenericPrimitive EMPTY>
<!ATTLIST GenericPrimitive name CDATA #IMPLIED
                           ident ID #IMPLIED>

<!ELEMENT GenericSequence (GenericSequence | GenericPrimitive | XREF |
                        | ...... SynRep )*>
<!ATTLIST GenericSequence ident ID #IMPLIED
             closed (yes) #IMPLIED
             type (sequence) #REQUIRED>
```

Note that GenericPrimitive and GenericSequence are treated essentially in the same way as the normal primitive types and sequence types described below.

10.1 XML Translations of the Types

Each of the RAGS abstract type definitions is in one of the following forms and so all that is necessary is a way of handling each of these cases:

$X \subset Primitives$
$X = A \times B \times C \ldots$ or $X = A^1$
$X = 2^A$ or $X = 2^A - \phi$
$X = R \rightarrow A$
$X = A^*$
$X = A^+$
$X = A^{++}$

where A, B and C are *simple* types, that is types that are explicitly defined in one of these ways or which are disjunctions of such types.

Here's how the type definitions give rise to descriptions to appear in the XML DTD.

10.1.1 Tuple Types

This covers types defined as:

$$X = A \times B \times C$$
$$X = A^1$$

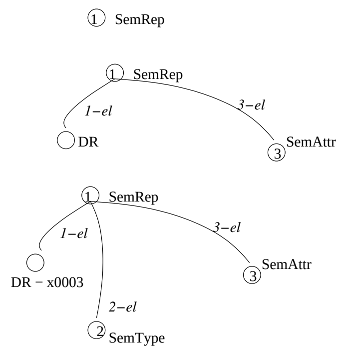


Figure 10.1: Partial Structures

These translate into:

```
<!ELEMENT X ((A | XREF | T), (B | XREF | T), (C | XREF | T))?>
<!ATTLIST X ident ID #IMPLIED
            type (tuple) #REQUIRED
            length (3) #REQUIRED>
```

The value of the `length` attribute specifies the number of components for this type (3 in this example). Entries for the positions of A, B and C can't be optional (for partial structures) since then we may not be able to tell which are which. A placeholder `T` is used to indicate a position that is not filled For example, Figure 10.1 shows three partial `SemRep` structures, which are represented as:

```
<SemRep ident="I1" type="tuple" length="3"/>

<SemRep ident="I1" type="tuple" length="3">
   <DR/>
   <T/>
   <SemAttr ident="I3" type="functional"/>
</SemRep>

<SemRep ident="I1" type="tuple" length="3">
   <DR name="x0003"/>
   <SemType ident="I2"/>
   <SemAttr ident="I3" type="functional"/>
</SemRep>
```

The use of an empty XML element is an alternative way of specifying an element with a full set of `T` components. Both of these specify an object with no arrows indicated inline. The `ident` is optional, as it is only really needed if an `XREF` needs to point to this element.

The above XML definition is also used for types defined by $X = A^1$, these being treated as tuples with just one component (of type A).

10.1.2 Set Types

This covers types defined as:

$$X = 2^A$$
$$X = 2^A - \phi$$

These translate into:

```
<!ELEMENT X (A | XREF)*>
<!ATTLIST X ident ID #IMPLIED
            closed (yes) #IMPLIED
            type (set) #REQUIRED>
```

Even for a non-empty set, this has to be Kleene *, to allow partiality and out of line arrows.

The `closed` attribute being set to `yes` indicates that the only relevant components are those specified in the current document (the set must have exactly this size).

10.1.3 Sequence Types

This covers types defined by one of:

$$X = A^*$$
$$X = A^+$$
$$X = A^{++}$$

These translate into:

```
<!ELEMENT X (A | XREF | T)*>
<!ATTLIST X ident ID #IMPLIED
            closed (yes) #IMPLIED
            type (sequence) #REQUIRED>
```

The use of Kleene * and the `closed` attribute are as for the set types, though the `type` attribute has a different value. Note that the order of the embedded elements corresponds to the order in the sequence.

10.1.4 Functional Types

This covers types defined as:

$$X = R \to A$$

Here it is assumed that R is always primitive (not even a disjunction of primitive types). As in the objects and arrows model, we assume that this is represented in the same way as $X = 2^{R \times A}$ would be. Thus this translates into:


```
<!ELEMENT X (XXX)*>
<!ATTLIST X ident ID #IMPLIED
            closed (yes) #IMPLIED
            type (functional) #REQUIRED>
```

where `XXX` is an element type defined by:

```
<!ELEMENT XXX (R, (A|XREF))>
```

Such elements have no `ident` attributes (they cannot be sources of reentrancy) or `type` attributes (this can be inferred from the containing elements). Note that this means that the component arrows to and from these have to be expressed inline.

For instance, imagine we have the type X defined as $Roles \to Values$ and a complete structure with values v_i for roles r_i. Then this would come out as:

```
<X ident="I27" type="functional" closed="yes">
   <XXX>
      <Roles name="r1"/>
      <Values...value v1...>
   </XXX>
   <XXX>
      <Roles name="r2"/>
      <Values...value v2...>
   </XXX>
   ...
</X>
```

10.1.5 Primitive types

For most primitive types, the following is needed:

```
<!ELEMENT X EMPTY>
<!ATTLIST X name CDATA #REQUIRED
            ident ID #IMPLIED>
```

Where a primitive type R is only used in types of the form $R \to S$, the definition is the same except that no cross-referencing is possible and the name is required:

```
<!ELEMENT X EMPTY>
<!ATTLIST X name CDATA #REQUIRED>
```

10.2 The RAGS DTD

The RAGS overall DTD includes:

- The introduction of the RAGS document type and root element which can contain arbitrary objects and arrows, in any order (this includes primitives, XREFs and EXTREFs),
- The definition of XREF, EXTREF, T, GenericPrimitive, GenericSequence and `arrow`,

- Definitions of the different data representations, generated as above.

Here it is:

```
<!DOCTYPE RAGS [

<!ELEMENT RAGS ( GenericPrimitive | GenericSequence | arrow | XREF | EXTREF |
                 Adj | DR | DocAtom | DocAttr | DocFeat | DocLeaf | DocRep |
                 DocRepSeq | FVM | KBId | Nil | Quote | RhetLeaf | RhetRel |
                 RhetRep | RhetRepSeq | ScopeConstr | ScopeRel | ScopedSemRep
                 | Scoping | SemAttr | SemConstant | SemPred | SemRep |
                 SemRole | SemType | SynArg | SynAtom | SynFeat | SynFun |
                 SynRep )*>

   <!-- **** GENERAL **** -->

<!ELEMENT T EMPTY>

<!ELEMENT XREF EMPTY>
<!ATTLIST XREF idref IDREF #REQUIRED>

<!ELEMENT EXTREF EMPTY>
<!ATTLIST EXTREF ident ID #REQUIRED
                 url CDATA #REQUIRED>

<!ELEMENT arrow EMPTY>
<!ATTLIST arrow name CDATA #REQUIRED
                source IDREF #REQUIRED
                target IDREF #REQUIRED>

<!ELEMENT GenericPrimitive EMPTY>
<!ATTLIST GenericPrimitive name CDATA #IMPLIED
                           ident ID #IMPLIED>

<!ELEMENT GenericSequence (GenericSequence | GenericPrimitive | XREF | EXTREF
                           | Adj | DR | DocAtom | DocAttr | DocFeat | DocLeaf
                           | DocRep | DocRepSeq | FVM | KBId | Nil | Quote |
                           RhetLeaf | RhetRel | RhetRep | RhetRepSeq |
                           ScopeConstr | ScopeRel | ScopedSemRep | Scoping |
                           SemAttr | SemConstant | SemPred | SemRep | SemRole
                           | SemType | SynArg | SynAtom | SynFeat | SynFun |
                           SynRep )*>
<!ATTLIST GenericSequence ident ID #IMPLIED
   closed (yes) #IMPLIED
   type (sequence) #REQUIRED>

   <!-- **** CONCEPTUAL **** -->

<!ELEMENT KBId EMPTY>
<!ATTLIST KBId name CDATA #IMPLIED
          ident ID #IMPLIED>

   <!-- **** RHETORICAL **** -->
```



```
<!ELEMENT RhetRel EMPTY>
<!ATTLIST RhetRel name CDATA #IMPLIED
          ident ID #IMPLIED>

<!ELEMENT RhetRep ( (RhetRel | XREF | T),(RhetRepSeq | XREF | T))?>
<!ATTLIST RhetRep ident ID #IMPLIED
   type (tuple) #REQUIRED
   length (2) #REQUIRED>

<!ELEMENT RhetRepSeq ( ((RhetLeaf|RhetRep) | XREF | T))*>
<!ATTLIST RhetRepSeq ident ID #IMPLIED
   closed (yes) #IMPLIED
   type (sequence) #REQUIRED>

<!ELEMENT RhetLeaf EMPTY>
<!ATTLIST RhetLeaf name CDATA #IMPLIED
          ident ID #IMPLIED>

   <!-- **** DOCUMENT **** -->

<!ELEMENT DocRep ( (DocAttr | XREF | T),(DocRepSeq | XREF | T))?>
<!ATTLIST DocRep ident ID #IMPLIED
   type (tuple) #REQUIRED
   length (2) #REQUIRED>

<!ELEMENT DocRepSeq ( ((DocLeaf|DocRep) | XREF | T))*>
<!ATTLIST DocRepSeq ident ID #IMPLIED
   closed (yes) #IMPLIED
   type (sequence) #REQUIRED>

<!ELEMENT DocAttr (DocFeatAtom)*>
<!ATTLIST DocAttr ident ID #IMPLIED
   closed (yes) #IMPLIED
   type (functional) #REQUIRED>

<!ELEMENT DocFeatAtom (DocFeat,(XREF|DocAtom))>

<!ELEMENT DocLeaf ( (DocAttr | XREF | T))?>
<!ATTLIST DocLeaf ident ID #IMPLIED
   type (tuple) #REQUIRED
   length (1) #REQUIRED>

<!ELEMENT DocFeat EMPTY>
<!ATTLIST DocFeat name CDATA #REQUIRED>

<!ELEMENT DocAtom EMPTY>
<!ATTLIST DocAtom name CDATA #IMPLIED
          ident ID #IMPLIED>

   <!-- **** SEMANTIC **** -->

<!ELEMENT SemRep ( (DR | XREF | T),(SemType | XREF | T),(SemAttr | XREF | T))?>
<!ATTLIST SemRep ident ID #IMPLIED
```

```
   type (tuple) #REQUIRED
   length (3) #REQUIRED>

<!ELEMENT ScopedSemRep ( (DR | XREF | T),(SemType | XREF | T),
                         (SemAttr | XREF | T),(Scoping | XREF | T))?>
<!ATTLIST ScopedSemRep ident ID #IMPLIED
   type (tuple) #REQUIRED
   length (4) #REQUIRED>

<!ELEMENT SemType ( SemPred | XREF)*>
<!ATTLIST SemType ident ID #IMPLIED
   closed (yes) #IMPLIED
   type (set) #REQUIRED>

<!ELEMENT SemAttr (SemRoleRep)*>
<!ATTLIST SemAttr ident ID #IMPLIED
   closed (yes) #IMPLIED
   type (functional) #REQUIRED>

<!ELEMENT SemRoleRep (SemRole,(XREF|SemRep|ScopedSemRep|DR|SemConstant))>

<!ELEMENT Scoping ( (ScopeConstr | XREF | T))*>
<!ATTLIST Scoping ident ID #IMPLIED
   closed (yes) #IMPLIED
   type (sequence) #REQUIRED>

<!ELEMENT ScopeConstr ( (ScopeRel | XREF | T),(DR | XREF | T),(DR | XREF | T))?>
<!ATTLIST ScopeConstr ident ID #IMPLIED
   type (tuple) #REQUIRED
   length (3) #REQUIRED>

<!ELEMENT DR EMPTY>
<!ATTLIST DR name CDATA #IMPLIED
          ident ID #IMPLIED>

<!ELEMENT SemConstant EMPTY>
<!ATTLIST SemConstant name CDATA #IMPLIED
          ident ID #IMPLIED>

<!ELEMENT SemPred EMPTY>
<!ATTLIST SemPred name CDATA #IMPLIED
          ident ID #IMPLIED>

<!ELEMENT SemRole EMPTY>
<!ATTLIST SemRole name CDATA #REQUIRED>

<!ELEMENT ScopeRel EMPTY>
<!ATTLIST ScopeRel name CDATA #IMPLIED
          ident ID #IMPLIED>

   <!-- **** SYNTACTIC **** -->

<!ELEMENT SynRep ( (FVM | XREF | T),((SynRep|Nil) | XREF | T),
```



```
                   (SynArg | XREF | T),(Adj | XREF | T))?>
<!ATTLIST SynRep ident ID #IMPLIED
   type (tuple) #REQUIRED
   length (4) #REQUIRED>

<!ELEMENT FVM (SynFeatAtom)*>
<!ATTLIST FVM ident ID #IMPLIED
   closed (yes) #IMPLIED
   type (functional) #REQUIRED>

<!ELEMENT SynFeatAtom (SynFeat,(XREF|SynAtom|FVM))>

<!ELEMENT SynArg (SynFunRep)*>
<!ATTLIST SynArg ident ID #IMPLIED
   closed (yes) #IMPLIED
   type (functional) #REQUIRED>

<!ELEMENT SynFunRep (SynFun,(XREF|SynRep))>

<!ELEMENT Adj ( (SynRep | XREF | T))*>
<!ATTLIST Adj ident ID #IMPLIED
   closed (yes) #IMPLIED
   type (sequence) #REQUIRED>

<!ELEMENT Nil EMPTY>
<!ATTLIST Nil name CDATA #IMPLIED
          ident ID #IMPLIED>

<!ELEMENT SynFun EMPTY>
<!ATTLIST SynFun name CDATA #REQUIRED>

<!ELEMENT SynFeat EMPTY>
<!ATTLIST SynFeat name CDATA #REQUIRED>

<!ELEMENT SynAtom EMPTY>
<!ATTLIST SynAtom name CDATA #IMPLIED
          ident ID #IMPLIED>

   <!-- **** QUOTE **** -->

<!ELEMENT Quote EMPTY>
<!ATTLIST Quote name CDATA #IMPLIED
          ident ID #IMPLIED>


]>
```

10.3 Examples

To be legal XML, each of the following examples should appear in a file with the following format:

```
<?xml version="1.0"?>
.....the above DTD.....
<RAGS>
.....the example.....
</RAGS>
```

Instead of including the DTD directly, one can include at the same position in the file a pointer to where the DTD can be found, as follows:

```
<!DOCTYPE RAGS SYSTEM
 "xxxxxx">
```

Here xxxxxx can be the name of a local file or a URL for a file obtainable elsewhere. The DTD can be found in an appropriate form at `http://www.dai.ed.ac.uk/homes/chrism/rags.dtd`. Given the nature of the DTD, more than one example can appear between the `<RAGS>` ... `</RAGS>` delimiters.

10.3.1 Rhetorical Representation

The following is the example Rhetorical Representation of section 4.3, including the semantic elements:

```
<RhetRep ident="ID1" type="tuple" length="2">
   <RhetRel name="motivation"/>
   <RhetRepSeq ident="ID2" type="sequence">
       <RhetLeaf ident="ID3"/>
       <RhetLeaf ident="ID4"/>
   </RhetRepSeq>
</RhetRep>

<arrow name="refers_to" source="ID3" target="ID5"/>
<arrow name="refers_to" source="ID4" target="ID6"/>

<SemRep ident="ID5" type="tuple" length="3">
   <DR name="r5"/>
   <SemType type="set">
      <SemPred name="blow"/>
   </SemType>
   <SemAttr type="functional">
      <SemRoleRep>
         <SemRole name="actor"/>
         <SemConstant ident="c1" name="patient"/>
      </SemRoleRep>
      <SemRoleRep>
         <SemRole name="actee"/>
         <SemRep ident="s2" type="tuple" length="3">
            <DR name="r7"/>
            <SemType type="set">
               <SemPred name="nose"/>
            </SemType>
            <SemAttr type="functional">
               <SemRoleRep>
                  <SemRole name="owner"/>
```



```
                  <XREF idref="c1"/>
               </SemRoleRep>
            </SemAttr>
         </SemRep>
      </SemRoleRep>
   </SemAttr>
</SemRep>

<SemRep ident="ID6" type="tuple" length="3">
   <DR name="r6"/>
   <SemType type="set">
      <SemPred name="clear"/>
   </SemType>
   <SemAttr type="functional">
      <SemRoleRep>
         <SemRole name="subject"/>
         <XREF idref="s2"/>
      </SemRoleRep>
   </SemAttr>
</SemRep>
```

10.3.2 Document Representation

Here is the Document Representation of section 5.4.

```
<DocRep type="tuple" length="2">
   <DocAttr type="functional">
      <DocFeatAtom>
         <DocFeat name="TEXT-LEVEL"/>
         <DocAtom name="paragraph"/>
      </DocFeatAtom>
      <DocFeatAtom>
         <DocFeat name="LAYOUT"/>
         <DocAtom name="wrapped-text"/>
      </DocFeatAtom>
      <DocFeatAtom>
         <DocFeat name="POSITION"/>
         <DocAtom ident="pos1"/>
      </DocFeatAtom>
      <DocFeatAtom>
         <DocFeat name="PICTURE"/>
         <DocAtom name="nil"/>
      </DocFeatAtom>
   </DocAttr>
   <DocRepSeq type="sequence">
      <DocLeaf ident="l1" type="tuple" length="1">
         <DocAttr type="functional">
            <DocFeatAtom>
               <DocFeat name="TEXT-LEVEL"/>
               <DocAtom name="text-clause"/>
            </DocFeatAtom>
            <DocFeatAtom>
               <DocFeat name="LAYOUT"/>
```

```
               <DocAtom name="wrapped-text"/>
            </DocFeatAtom>
            <DocFeatAtom>
               <DocFeat name="POSITION"/>
               <DocAtom name="1"/>
            </DocFeatAtom>
            <DocFeatAtom>
               <DocFeat name="PICTURE"/>
               <DocAtom name="figs/noseblow.eps"/>
            </DocFeatAtom>
         </DocAttr>
      </DocLeaf>
      <DocLeaf ident="l2" type="tuple" length="1">
         <DocAttr type="functional">
            <DocFeatAtom>
               <DocFeat name="TEXT-LEVEL"/>
               <DocAtom name="text-clause"/>
            </DocFeatAtom>
            <DocFeatAtom>
               <DocFeat name="LAYOUT"/>
               <DocAtom name="wrapped-text"/>
            </DocFeatAtom>
            <DocFeatAtom>
               <DocFeat name="POSITION"/>
               <DocAtom name="2"/>
            </DocFeatAtom>
            <DocFeatAtom>
               <DocFeat name="PICTURE"/>
               <DocAtom name="nil"/>
            </DocFeatAtom>
         </DocAttr>
      </DocLeaf>
   </DocRepSeq>
</DocRep>

<arrow name="refers_to" source="l1" target="s4"/>
<arrow name="refers_to" source="l2" target="s16"/>

<SynRep type="tuple" length="4" ident="s4"/>
<SynRep type="tuple" length="4" ident="s16"/>
```

10.3.3 Semantic Representation

The following is the SemRep from section 6.3.1:

```
<SemRep ident="I301" type="tuple" length="3">
   <DR name="lc1"/>
   <SemType ident="I302" type="set">
      <SemPred name="Logical-Condition"/>
   </SemType>
   <SemAttr ident="I303" type="functional" closed="yes">
      <SemRoleRep>
        <SemRole name="domain"/>
```



```
        <XREF idref="I324"/>      <!-- to ni-number-on-letter -->
      </SemRoleRep>
      <SemRoleRep>
         <SemRole name="range"/>
         <XREF idref="I304"/>      <!-- to gets-letter -->
      </SemRoleRep>
      <SemRoleRep>
         <SemRole name="theme"/>
         <XREF idref="I304"/>      <!-- to gets-letter -->
      </SemRoleRep>
   </SemAttr>
</SemRep>

<SemRep ident="I304" type="tuple" length="3">
   <DR name="gets-letter"/>
   <SemType ident="I305" type="set">
      <SemPred name="get"/>
   </SemType>
   <SemAttr ident="I306" type="functional" closed="yes">
      <SemRoleRep>
         <SemRole name="actor"/>
         <SemRep ident="I307" type="tuple" length="3">
            <DR name="h"/>
            <SemType ident="I308" type="set">
               <SemPred name="hearer"/>
            </SemType>
            <SemAttr ident="I399" type="functional" closed="yes">
               <SemRoleRep>
                  <SemRole name="recoverability"/>
                  <SemConstant name="recoverable"/>
               </SemRoleRep>
            </SemAttr>
         </SemRep>
      </SemRoleRep>
      <SemRoleRep>
         <SemRole name="actee"/>
         <SemRep ident="I309" type="tuple" length="3">
            <DR name="letter1"/>
            <SemType ident="I310" type="set">
               <SemPred name="letter"/>
            </SemType>
            <SemAttr ident="I311" type="functional" closed="yes">
               <SemRoleRep>
                  <SemRole name="identifiability"/>
                  <SemConstant name="nonidentifiable"/>
               </SemRoleRep>
               <SemRoleRep>
                  <SemRole name="actor"/>
                  <SemRep ident="I312" type="tuple" length="3">
                     <DR name="a1"/>
                     <SemType ident="I313" type="set">
                        <SemPred name="arbitrary"/>
                     </SemType>
```

```
               <SemAttr ident="I314" type="functional" closed="yes"/>
            </SemRep>
         </SemRoleRep>
         <SemRoleRep>
            <SemRole name="modification"/>
            <SemRep ident="I315" type="tuple" length="3">
               <DR name="address1"/>
               <SemType ident="I316" type="set">
                  <SemPred name="address-action"/>
               </SemType>
               <SemAttr ident="I317" type="functional" closed="yes">
                  <SemRoleRep>
                     <SemRole name="time"/>
                     <SemConstant name="e&lt;r"/> <!-- Note char is escaped -->
                  </SemRoleRep>
                  <SemRoleRep>
                     <SemRole name="actee"/>
                     <DR name="letter1"/>
                  </SemRoleRep>
                  <SemRoleRep>
                     <SemRole name="modification-form"/>
                     <SemConstant name="clause"/>
                  </SemRoleRep>
                  <SemRoleRep>
                     <SemRole name="destination"/>
                     <SemRep ident="I318" type="tuple" length="3">
                        <DR name="h"/>
                        <SemType ident="I319" type="set">
                           <SemPred name="hearer"/>
                        </SemType>
                        <SemAttr ident="I320" type="functional" closed="yes">
                           <SemRoleRep>
                              <SemRole name="recoverability"/>
                              <SemConstant name="recoverable"/>
                           </SemRoleRep>
                        </SemAttr>
                     </SemRep>
                  </SemRoleRep>
               </SemAttr>
            </SemRep>
         </SemRoleRep>
      </SemAttr>
   </SemRep>
</SemRoleRep>
<SemRoleRep>
   <SemRole name="time"/>
   <SemConstant name="e=r&lt;s"/>        <!-- Note char is escaped -->
</SemRoleRep>
<SemRoleRep>
   <SemRole name="inclusive"/>
   <SemRep ident="I321" type="tuple" length="3">
      <DR name="form1"/>
      <SemType ident="I322" type="set">
```



```
               <SemPred name="form"/>
            </SemType>
            <SemAttr ident="I323" type="functional" closed="yes">
               <SemRoleRep>
                  <SemRole name="deixis"/>
                  <SemConstant name="environmental"/>
               </SemRoleRep>
            </SemAttr>
         </SemRep>
      </SemRoleRep>
   </SemAttr>
</SemRep>

<SemRep ident="I324" type="tuple" length="3"/> <!-- ni-number-on-letter omitted -->
```

10.4 XML Version of KB Interface

In chapter 3 the abstract interface to KBIds is specified. In many systems, KBIds will arise from some knowledge base or reasoning component distinct from the main NLG system, and it will be that component that is in charge of supporting the API. In this case, it may be useful to have a concrete syntax for expressing queries to the knowledge base and the responses to them. The following is a suggested DTD for this. It is available at `http://www.dai.ed.ac.uk/homes/chrism/ragskb.dtd`

```
<?xml version="1.0"?>

<!DOCTYPE KB [

<!ELEMENT KB (ISSTRING | ISNUMBER | EQUIVALENT | SUBSUMED-BY | ROLE-VALUES |
   TYPES | ROLES-WITH-VALUES | BOOL | KBId* |
   SemPred* | SemRole*)>

<!-- QUERIES ******************** -->

<!ELEMENT ISSTRING (KBId)>

<!ELEMENT ISNUMBER (KBId)>

<!ELEMENT EQUIVALENT (KBId, KBId)>

<!ELEMENT SUBSUMED-BY (KBId, SemPred)>

<!ELEMENT ROLE-VALUES (SemRole, KBId)>

<!ELEMENT TYPES (KBId)>

<!ELEMENT ROLES-WITH-VALUES (KBId)>
```

```
<!-- RESPONSES ******************** -->

<!ELEMENT BOOL EMPTY>
<!ATTLIST BOOL name (true|false|unknown) CDATA #REQUIRED>

<!ELEMENT KBId EMPTY>
<!ATTLIST KBId name CDATA #REQUIRED>

<!ELEMENT SemPred EMPTY>
<!ATTLIST SemPred name CDATA #REQUIRED>

<!ELEMENT SemRole EMPTY>
<!ATTLIST SemRole name CDATA #REQUIRED>

]>
```

This DTD allows a document to consist of the specification of one call to one of the supported functions or to be the specification of the result of such a call. Thus there is one element defined for each of the possible functions and that element must contain the specification of the actual arguments of that function. For instance, the following document (whose header information has been removed) represents a call to *kb_subsumed_by(kb998,animate)*:

```
<KB>

<SUBSUMED-BY>
   <KBId name="kb998"/>
   <SemPred name="animate"/>
</SUBSUMED-BY>

</KB>
```

Similarly the DTD allows a document to consist of a possible result from such a function call. Here Booleans are represented by empty elements in the same way as RAGS primitives. The `name` attribute specifies the value (`true`, `false` or `unknown`) – note that for convenience `unknown` is treated as a Boolean. So a possible response to the above query would be:

```
<KB>

<BOOL name="true"/>

</KB>
```

For calls to other functions, the result might be a set of KBIds (represented simply by a sequence of XML elements), a set of SemPreds etc. The DTD allows all the appropriate possibilities but does not provide any check that a given function call is given the right kind of result.

Chapter 11

Acknowledgements

This project is supported by the UK Engineering and Physical Sciences Research Council through grants GR/L77041 and GR/L77102. The following people have kindly offered to act as consultants to the project and we are grateful to them for their interest and feedback: John Bateman, Kalina Boncheva, Stefan Busemann, Sandra Carberrry, John Carroll, Alison Cawsey, Robert Dale, Keed van Deemter, Nancy Green, Helmut Horacek, Dick Kittredge, James Lester, David McDonald, Kathy McKeown, Johanna Moore, Cecile Paris, Ehud Reiter, Oliviero Stock, Lyn Walker, Ingrid Zukerman.

We would also like to especially thank David McDonald for his incredibly detailed comments on two earlier versions of this document.

Bibliography

[1] J.A. Bateman. Automated discourse generation. In A. Kent, editor, *Encyclopedia of Library and Information Science*, volume 62. Marcel Dekker Inc., New York, 1998.

[2] John A. Bateman. Enabling technology for multilingual natural language generation: the KPML development environment. *Natural Language Engineering*, 3(1):15–55, 1997.

[3] J. (ed.) Bresnan. *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge, Mass., 1982.

[4] Lynne Cahill, Roger Evans, Chris Mellis, Daniel Paiva, Mike Reape, and Donia Scott. Introduction to the RAGS architecture . Available at `http://www.itri.brighton.ac.uk/projects/rags`, 2001.

[5] N. Chomsky. *Lectures on Government and Binding*. Foris, Dordrecht, 1981.

[6] World Wide Web Consortium. Extensible markup language (XML). Available at http://www.w3.org/TR/PR-xml.html, 1997.

[7] Annette Frank and Uwe Reyle. Principle-based semantics for HPSG. In *Proceedings of EACL95*, 1995.

[8] Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar*. Harvard University Press, 1985.

[9] Barbara J. Grosz and Candace L. Sidner. Attention, intentions and the structure of discourse. *Computational Linguistics*, 12(3):175–204, July-September 1986.

[10] J. R. Hobbs. On the coherence and structure of discourse. Technical Report CSLI-85-37, CSLI, Palo Alto, 1985.

[11] Hans Kamp and Uwe Reyle. *From Discourse to Logic: Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Kluwer, 1993.

[12] Robert T. Kasper. A flexible interface for linking applications to PENMAN's sentence generator. In *Proceedings of the DARPA Workshop on Speech and Natural Language*, Philadelphia, 1989. Available from USC/Information Sciences Institute, Marina del Rey, CA.

[13] William C. Mann and Sandra A. Thompson. Rhetorical structure theory: Toward a functional theory of text organization. *Text*, 8(3):243–281, 1988. Also available as USC/Information Sciences Institute Research Report RR-87-190.

[14] Christopher D. Manning and Ivan A. Sag. Dissociations between ARG-ST and grammatical relations. In Gert Webelhuth, Jean-Pierre Koenig, and Andreas Kathol, editors, *Lexical and Constructional Aspects of Linguistic Explanation*, pages 63–78. CSLI, Stanford, 1999.

[15] Chris Mellish, Mick O'Donnell, Jon Oberlander, and Alistair Knott. An architecture for opportunistic text generation. In *Proceedings of the 9th International Workshop on Natural Language Generation*, pages 28–37, Niagra-on-the-Lake, 1998.

[16] Igor A. Mel'čuk. *Dependency Syntax: Theory and Practice*. State University of New York Press, Albany, 1988.

[17] Marie Meteer. *The Generation Gap: The Problem of Expressibility in Text Planning*. PhD thesis, University of Massachusetts, 1990.

[18] Marie Meteer. *Expressibility and the Problem of Efficient Text Planning*. Pinter, 1992.

[19] Geoffrey Nunberg. *The Linguistics of Punctuation*. CSLI Lecture Notes, No. 18. Center for the Study of Language and Information, Stanford, 1990.

[20] Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics. Vol 1: Fundamentals*. CSLI, Stanford, 1987.

[21] Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, Chicago, 1994.

[22] D. Radev, N. Kambhatla, Y. Ye, C. Wolf, and Z. Wlodek. DSML: A proposal for XML standards for messaging between components of a natural language dialogue system. In *Proceedings of the AISB'99 Workshop on Reference Architectures and Data Standards*, University of Edinburgh, April 1999.

[23] Ehud Reiter and Roma Robertson. The architecture of the stop system. In Proceedings of the Workshop on Reference Architectures for Natural Language Generation. Edinburgh, Scotland. Available at `http://www.itri.bton.ac.uk/projects/rags`, 1999.

[24] Uwe Reyle. Reasoning with ambiguities. In *Proceedings of EACL95*, 1995.

[25] Stuart Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343, 1985.

[26] H. Uszkoreit. Categorial unification grammars. In *Proceedings of the 11th International Conference on Computational Linguistics (COLING-86)*, Bonn, 1986. Also appeared as Report No. CSLI-86-66, Stanford: CSLI.

[27] K. Vander Linden. GIST - Specification of the Extended Sentence Planning Language. Technical Report LRE Project 062-09 Deliverable TST-0, Information Technology Research Institute (ITRI), University of Brighton, 1994.

[28] Henk Zeevat. *Aspects of Discourse Semantics and Unification Grammar*. PhD thesis, University of Amsterdam, 1991.

[29] Henk Zeevat, Ewan Klein, and Jo Calder. Unification Categorial Grammar. In Nicholas J. Haddock, Ewan Klein, and Glyn Morrill, editors, *Categorial Grammar, Unification Grammar and Parsing*, volume 1 of *Edinburgh Working Papers in Cognitive Science*. Centre for Cognitive Science, University of Edinburgh, 1987.

Appendix A

Notation for Type Definitions

Our specifications will be defined in terms of *sets* and *tuples*. The name of a set will always indicate what kind of element it is a set of. For example AbsSynRep is a set of abstract syntactic representations.

A tuple is an *ordered* multi-set.

It is useful to define some operations on sets. Let A, B, C, D, $\ldots$, A_1, A_2, $\ldots$ be sets. Furthermore, let the notation $\langle a_1, \ldots, a_n \rangle$ be a tuple of elements $a_1, \ldots, a_n$ *in that order.*

Then the first thing we need is "set constructor notation". $\{x \mid P(x)\}$ denotes the set of elements that make the proposition P predicated of x true. For example, $\{x \mid x = 3{\times}n \text{ for } n \geq 0\}$ is the set of all numbers which are 3 times a natural number. Then we can define the following set operations in terms of set constructor notation.

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\} \tag{A.1}$$

$$2^A = \{X \mid X \subseteq A\} \tag{A.2}$$

$$A - B = \{x \mid x \in A \text{ and } x \notin B\} \tag{A.3}$$

$$\emptyset = \{\} \tag{A.4}$$

$$A \times B = \{\langle a, b\rangle \mid a \in A \text{ and } b \in B\} \tag{A.5}$$

$$A_1 \times \ldots \times A_n = \{\langle a_1, \ldots, a_n\rangle \mid a_1 \in A_1 \text{ and } \ldots \text{ and } a_n \in A_n\} \tag{A.6}$$

Then (A.1) says that $A \cup B$ is the union of A and B, (A.2) says that 2^A is the set of all subsets of A, (A.3) says that $A - B$ is the set of all elements of A not in B, (A.4) says that $\emptyset$ is the set with no elements and (A.5) says that $A \times B$ is the set of all 2-tuples whose first element is an element of A and whose second element is an element of B. Similarly, (A.6) says that $A_1 \times \ldots \times A_n$ is the set of all n-tuples whose first element is an element of A_1, whose second element is an element of A_2, $\ldots$ and whose n-th element is an element of A_n. (Note that (A.5) is a subcase of (A.6).)

Then we can make the following tuple notation definitions.

$$A^0 = \langle\rangle \tag{A.7}$$

$$A^1 = \{\langle a\rangle \mid a \in A\} \tag{A.8}$$

$$A^2 = \{\langle a_1, a_2\rangle \mid a_1, a_2 \in A\} \tag{A.9}$$

$$A^3 = \{\langle a_1, a_2, a_3\rangle \mid a_1, a_2, a_3 \in A\} \tag{A.10}$$

$$\begin{aligned} \ldots &= \ldots && \text{(A.11)}\\ A^n &= \{\langle a_1, \ldots, a_n\rangle \mid a_1, \ldots, a_n \in A\} && \text{(A.12)} \end{aligned}$$

Then the above definitions say that A^n is the set of all tuples of length n whose elements are elements of A.

Finally, we define "Kleene plus" and versions of "Kleene star" over sets.

$$A^+ = \bigcup_{n \geq 1} A^n \tag{A.13}$$

$$A^{++} = \bigcup_{n \geq 2} A^n \tag{A.14}$$

$$A^* = \bigcup_{n \geq 0} A^n \tag{A.15}$$

That is, A^+ is the set of all tuples of elements of A of length 1 or greater and $A^* = A^+ \cup \{\langle\rangle\}$.

This takes care of all the notation used here to "specify" datatypes. We have yet to indicate how the input and output arguments of operations are specified.

If f is an operation and A and B are sets then $f : A \to B$ means that "f is a function from A to B". I.e., the *domain* of f is a subset of A and the *range* of f is a subset of B. $f :\to B$ means that f is a *nullary* operation, i.e., it takes no input argument. When an operation takes more than one input or output argument, the definition looks like the following:

$$f : A \times B \to C \times D \times E \tag{A.16}$$

Basically, this means that if an operator has two input arguments and three "value" arguments they are "packaged up" into a 2-tuple and a 3-tuple respectively. This means that in some cases you have to "wrap" a tuple argument in another level of tupling to get the intended meaning but this doesn't arise in any of what follows.

NB:

$$\langle a, b\rangle \neq \langle a, \langle b\rangle\rangle \neq \langle\langle a\rangle, b\rangle \neq \langle\langle a\rangle, \langle b\rangle\rangle \tag{A.17}$$

just like in Lisp or Prolog.

Finally, the notation $A \to B$ denotes the set of *all* functions from A to B, i.e., whose *domain* is a subset of A and whose *range* is a subset of B.

Technically, the equations we use are *domain equations*. The main thing to watch for is equations like $X = A \cup (A \times X)$ where X and A are sets. Literally, what the equations says is that the set X is the union of the set A and the set $\{\langle a, x\rangle \mid a \in A, x \in X\}$. That is, X is defined in terms of *itself*. Assume that $A = \{a\}$. Then

$$X = \{a, \langle a, a\rangle, \langle a, \langle a, a\rangle\rangle, \langle a, \langle a, \langle a, a\rangle\rangle\rangle \ldots\} \tag{A.18}$$

Equations like this are conventionally understood as meaning "the smallest set X such that ...".

So to take an example which may be familiar, consider the following definition of the set of feature structures *without* reentrancy. Let FS be the set of feature structures, V the set of atomic values and F the set of features. Then

$$FS = V \cup (F \to FS) \tag{A.19}$$

That is, the set of feature structures is the union of the set V and the set of all functions from the set F to the set FS. (This is a very well-known equation.)

Appendix B

Glossary

Abstract (of a representation) – occurring at a relatively early point in an NLG system and hence probably being rather tentative and unfinished. See Sections 5.1, 6.1, 7.1 and 8.1.

Concrete (of a representation) – occurring at a relatively late point in an NLG system and hence probably being rather well-considered and finished. See Sections 5.1, 6.1, 7.1 and 8.1.

KBId – "Knowledge base ID". Another term for Conceptual Representations. See Chapter 3.

Primitive type – Type of a terminal in a RAGS representation. Must be further specified in a theory-dependent way. See Chapter 2.

Representation – The form of specific types of (linguistic) information. Fundamentally, this is a declarative notion.

Data-type – see Abstract Type.

Abstract Type – Formal definition of the syntactic and semantic form of a level of representation. See Chapter 1.

Operation – Any means of changing one representation into another. May change a representation of one type into another representation of the same type, or may change a representation of one type into a representation of another type. See Chapter 1.

Applied (of an NLG system) – used for an externally specified task. See Chapter 1.

Complete (of an NLG system) – performs the entire generation process from (usually) non-linguistic input to text. See Chapter 1.