

Grail Tutorial

November 1998

This document presents a tutorial introduction to the GRAIL concept modelling language. It is designed to be used in an interactive manner, building up a small demonstrative model using the modelling tools that communicate with a GALEN Terminology Server.

This version of the tutorial is for use with the KnoME modelling environment (version 3.1c or above).

For help or further information, please contact:

galen-help@cs.man.ac.uk

(+44) 161 275 6239/6133

Contents

1. INTRODUCTION 4

1.1 WHAT THIS TUTORIAL IS 4
1.2 WHAT THIS TUTORIAL ISN'T 4
1.3 STRUCTURE OF THIS TUTORIAL 5
1.4 WHERE TO FIND FURTHER INFORMATION 5

2. GALEN, AND MODELLING WITH GRAIL 6

2.1 A BRIEF HISTORY 6
2.2 WHAT IS A "TERMINOLOGICAL MODEL"? 6
2.3 PRE-TUTORIAL - AN INTERACTIVE INTRODUCTION TO THE GRAIL KNOME 7
2.3.1 Accessing the Tools 7
2.3.2 Emptying any existing model from the Terminology Server 8
2.3.3 Creating a new model 9
2.3.4 Viewing the model 9
2.4 SYNTAX 10
2.4.1 Use of Brackets 10
2.4.2 Use of full stops 10
2.4.3 Comments 10
2.5 A WORD ABOUT VOCABULARY 11
2.5.1 Changes in Vocabulary 11

3. AN INTRODUCTORY GRAIL TUTORIAL 12

3.1 AN EXAMPLE APPLICATION AREA 12
3.1.1 What we want to do 12
3.1.2 Doing it in GRAIL 13
3.2 FORMAL SUBSUMPTION: STATEMENTS, DEFINITIONS, AND COMPOSITE CATEGORIES 15
3.2.1 Creating attributes 15
3.2.2 Sanctioning statements 16
3.2.3 Defining Composite Categories 18
3.2.4 Cardinality 18
3.2.5 Criteria 19
3.2.6 Formal Subsumption 19
3.2.7 Value Types 19
3.2.8 Criteria 20
3.2.9 Naming 21
3.3 NECESSARY STATEMENTS AND CANONICAL FORM 22
3.3.1 Necessary statements 22
3.3.2 Canonical Forms 22
3.3.3 The Hierarchy 23
3.4 FURTHER FEATURES 25
3.4.1 Grammar level Queries (Grammar level Composite Categories) 25
3.4.2 Topic and Value necessary statements 26
3.5 SOME COMMON QUESTIONS 26
3.5.1 What is the difference between a criterion in a definition (a which expression) and a criterion in a necessary statement? 26
3.5.2 What is the model? 27

4. USING GRAIL WELL: A TUTORIAL IN BASIC GRAIL STYLE 28

4.1 RE-USE, APPLICATION INDEPENDENCE, AND MODELLING STYLE 28
4.2 THE EXAMPLE MODEL REVISITED 28
4.3 EXTENDING THE MODEL TO A SECOND CLASSIFICATION 28
4.4 REFORMULATING THE MODEL 30
4.4.1 Removing RoadVehicles and RailVehicles (flattening the asserted hierarchy) *31*
4.4.2 Creating the attributes and values *31*
4.4.3 Describing the elementary categories. *31*
4.4.4 Creating the abstractions *32*
4.4.5 Sorting out the order of statements in the source file. *32*
4.4.6 Checking the syntax *32*
4.4.7 Checking the result *32*
4.5 FINAL REMARKS ON THE EXAMPLE 33
4.5.1 Further reformulation of the model? The need for an 'ontology' *33*
4.5.2 Natural kinds and elementary categories *33*
4.5.3 When there must be multiple parents in the asserted hierarchy? *33*
4.6 SUMMARY: PRINCIPLES OF BUILDING RE-USABLE MODELS IN GRAIL 34

5. REFINEMENT AND TRANSITIVITY: WHAT MAKES GRAIL DIFFERENT 35

5.1 TRANSITIVITY 35
5.2 THE SPECIALISEDBY OPERATOR ('REFINEMENT') 37
5.3 TRANSITIVE RELATIONS AND CARDINALITY 38
5.4 WHY ARE SPECIALISEDBY AND TRANSITIVEDOWN IMPORTANT 38
5.5 HOW TO REMEMBER HOW SPECIALISEDBY WORKS 38

6. OTHER FEATURES 39

6.1 EXTRINSIC INFORMATION, 'DEFAULTS', AND INDEXING 39
6.2 CODING SCHEMES 40
6.3 LANGUAGE 40

7. GLOSSARY 41

1. Introduction

GRAIL is a concept modelling language that has been developed and used by the GALEN projects to represent medical terminology. It has also been used to model other domains. GRAIL stands for the GALEN Representation and Integration Language.

1.1 What this tutorial is

This tutorial is an attempt to soften the initial learning curve that has been seen when encountering GRAIL for the first time and to introduce you to what we consider to be good practice in GRAIL modelling. The document adopts a hands-on approach - through the use of examples - to gain experience in the modelling process, and of some of the supporting tools which have been developed.

To make best use of this tutorial, work through it step-by-step using the GRAIL modelling environment. If you do not have access to this environment, contact

`galen@cs.man.ac.uk`

for information on how to obtain and install the software. Note that there are now two different modelling environments. Be sure you get the version of the tutorial that goes with your modelling environment: this version is for use with the GRAIL KnoME (version 3.1c and above) [1].

This tutorial is designed to introduce the GRAIL language itself: it is *not* an introduction to the clinical models that have been developed using that language. A conscious decision was made not to use medicine in the examples, in an attempt to restrict the areas of discussion! Further information is available (see below) to introduce the way GRAIL has been used to represent clinical terminology in the GALEN projects.

This tutorial has been written from a limited number of people's experience and difficulties. We are keen that others - both existing members of the select group who model in GRAIL, and any newcomers - should contribute to the development of this tutorial.

1.2 What this tutorial isn't

This document is *not* an attempt to integrate all existing documentation into a handy pocket-sized edition and this document won't tell you in detail which mouse button to use for any but the basic functions - further documentation is available which describe the modelling environments in detail, and this will be needed if further modelling is to be undertaken.

This document is also not a tutorial in the GALEN ontology and how we use GRAIL, in detail, in GALEN. This is covered separately. It also does not cover the associated facilities for natural language handling, handling coding systems, and the use of GRAIL for indexing knowledge bases.

This tutorial is to get you started. There's lots more, but this needs to be mastered first.

---

1 The two environments have roughly the same functionality from the point of view of this tutorial, but are arranged somewhat differently. Details of which button to click are different and the windows are arranged differently.

1.3 Structure of this tutorial

After this short introduction, Section 2 provides some history of GALEN and GRAIL, and introduces the modelling process, and the tools used to perform and support that process. Section 3 is the tutorial proper, whilst Section 4 provides a discussion of the different ways of modelling in GRAIL: some preferred, some not. Section 5 concentrates on two features that distinguish GRAIL from its peers, and Section 6 provides an overview of some other features of GRAIL. A short glossary concludes the tutorial.

There are some sections in the tutorial specifically aimed at those with experience of other systems - description logics, conceptual graphs, or object oriented design environments. These 'side-bars' are marked optional and distinguished in the text. They are not needed to understand GRAIL, but may be useful to those coming to GRAIL from related environments to help them understand the similarities and differences between GRAIL and systems with which they are familiar.

An important thing to keep in mind is that modelling in and using GRAIL can only really be learnt through a 'hands on' approach. Follow the examples, but experiment on your own at any time.

1.4 Where to find further information

Much has been written on GRAIL, GALEN and the overall modelling process. The following documents may also be relevant:

- Overview of GALEN (Volume B of the GALEN Documentation Set) and a set of Frequently Asked Questions (Volume F);
- The GRAIL KnoME: Overview and User Guide, which offers full details of the modelling environment which is featured in this version of the tutorial;
- The GRAIL Concept Modelling Language for Medical Terminology, a paper which describes the GRAIL language in more detail, and in a different style, than this tutorial (a shortened version is published in the January-February 1997 issue of Artificial Intelligence in Medicine);
- for those wishing to use their knowledge of GRAIL to understand, or extend the clinical model being developed in the GALEN projects, see Deliverable 5.1 of the GALEN-IN-USE project.

Additional (possibly slightly historical) detailed information is available in the GALEN Project Deliverables (in particular 6, 14, and 15).

The GRAIL modelling environments are under continuous development - indeed they can only develop successfully with requirements from existing and new users. So, if there is any aspect of the environments that need changing or extending, or if you think that a totally new approach is needed - let us know!

2. GALEN, and modelling with GRAIL

2.1 A brief history

The GRAIL formalism was developed for use in the GALEN programme for formal representation of medical terminologies. It has since been found useful for a variety of other applications both within and outside medicine where there is a need to describe and classify information - be it the function of enzymes or DNA sequences, the content of pictures, or the nature of diseases and medical procedures.

The motivation for the special features of GRAIL is closely tied to its origins: the support for high quality user interfaces for doctors in the PEN&PAD system. PEN&PAD's aim is to allow the direct entry of detailed clinical data by the practitioner during the consultation, and to display that data in a useful way - to 'shed light' on the medical record.

It was found that building the interface which emerged from our User Centred Design Trials while using existing enumerative coding schemes (e.g. the enumerative Read classification of clinical terms) was simply impossible - a view borne out by the current state of commercial GP systems that attempt to use such schemes. GRAIL's immediate ancestor, SMK {xe "SMK "}(Structured Meta Knowledge), was developed to overcome some of the limitations of enumerative representations. SMK integrated medical terminological knowledge, medical records and the medical dialogue into a single formalism, with the knowledge base of medical terminology providing the semantic control over the patient record.

GRAIL is the development of the terminological, or conceptual, aspects of SMK.

> **Sidebar: Relation to other systems :** skip this section if you are not familiar with Frame Systems, Conceptual Graphs or Description Logics)
>
> In a broader context, GRAIL is a 'Description Logic', i.e. one of a class of formalisms which attempt to make rigorous the notions of informal 'Frame-based systems'. It is related to KL-ONE, Loom, Classic, or Back, although it uses a different notation. By comparison with other description logics, GRAIL has extra features to deal with part-whole relations and simple 'rules', but it lacks other features such as detailed handling of numbers which were found to be less important for medical terminology. (Some of these restrictions can be irksome, and some may be relaxed in future versions.)
>
> Like other description logics, GRAIL is also closely related to Sowa's Conceptual Graphs, and much of GRAIL can be mapped onto the subset of conceptual graphs commonly used for describing terminology.
>
> GRAIL's special features and its unusual syntax make it easy to follow a particular 'style' in developing 'terminological models' or 'ontologies'.. This style has been designed to make models more re-usable and less linked to particular applications. A detailed description of the GALEN ontology and modelling style can be found in Deliverable 5.1 of the GALEN-IN-USE project. This tutorial introduces only the mechanics of the language and the basic principles of modelling style.
>
> GRAIL's definition is now (almost) stable. However, GRAIL is not the ultimate description logic or ultimate system for representing medical terminology. There will be successor systems. But GRAIL is an important step along the way and sufficient for many applications.

2.2 What is a "terminological model"?

Terminological {xe "terminological "}modelling represents the definitions and essential properties of concepts. The terminological definition of a concept provides sufficient information to define, or recognise, that concept. Contrast this with, for example, modelling

facts or assertions about concepts, which we may know to be true in the world, either about particular concepts, or the interaction between them, but which do not add to the definition of those concepts. For example, 'wine' might be defined as a 'drink made of distilled grape juice'. Other information not part of the definition might include that wines are either red or white.

Terminology may be seen as the definitional 'backbone of the concept structure that we wish to describe. It may well be the case that one of the important uses of such a structure is to handle non-definitional assertions about the real world. The features GRAIL provides to handle assertions will be illustrated in the examples.

Terminological models are also called 'ontologies' by many authors, although other authors use the word 'ontologies' to indicate a rather broader kind of knowledge base than that which we consider 'terminological'.

Finally, it is not always clear whether what you wish to say falls into the category of terminological knowledge, or not. Don't worry, but come back to this point later, after you've been through the tutorial. If in doubt, raise the issue personally or by email.

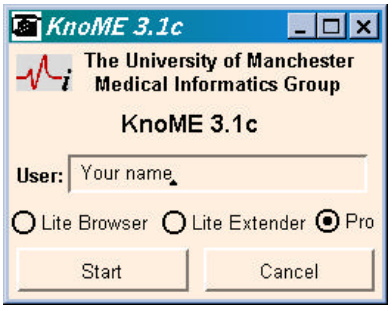
2.3 Pre-Tutorial - An Interactive Introduction to the GRAIL KnoME

One of the GRAIL development environments that is available is known as the 'KnoME' (for Knowledge Modelling Environment), which is a suite of modelling tools for building, browsing and exploring a GRAIL model. This pre-tutorial will provide you wish enough of a starting point to progress with the tutorial. See 'The GRAIL KnoME: Overview and User Guide' for more detailed coverage of the environment.

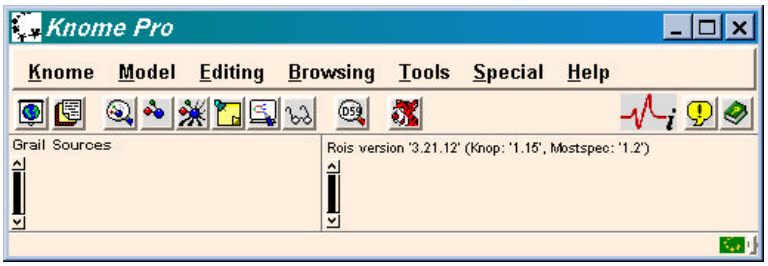
2.3.1 Accessing the Tools

In the environment you will be using, the KnoME communicates with a Smalltalk implementation of a GALEN Terminology Server: most of the time you will only interact with the KnoME itself (via the KnoME Main Window), but occasionally you will need to access functions of the server itself (for example to empty a model) : a user interface for this is also provided (the Terminology Server Console).

When the environment starts (consult your local installation guide or technical support for details of this if you have difficulties), you will see the initial logon screen. Enter your name and ensure that the 'Pro' Radio Button is selected.

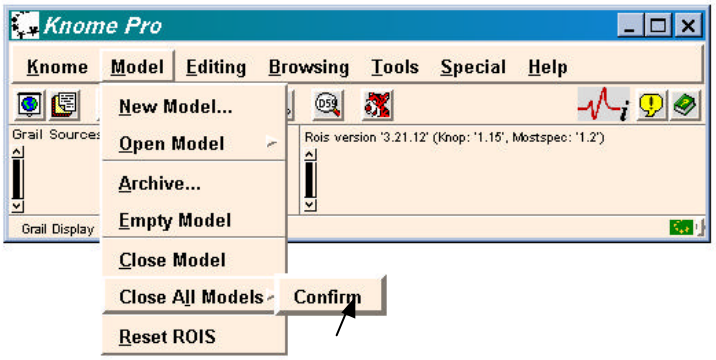


When the 'Start' button is pressed, the KnoME Main Window appears.

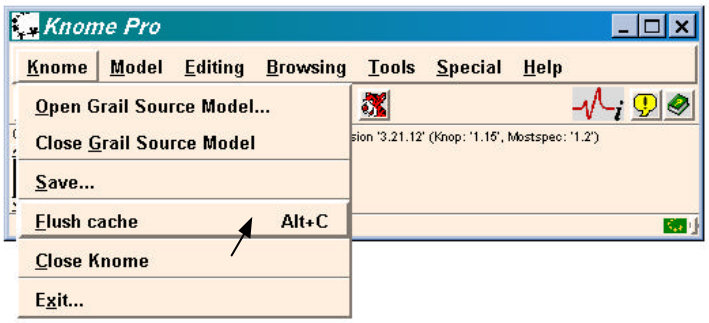


2.3.2 Closing any existing models in the KnoME

Before starting to model, you need to make sure that there are no open models. Close any existing open model from the **Knome Pro → Model → Close all models → Confirm**:



Note that each time you reset the model, you will need to clear any cached results that the KnoME may have, use the menu option: **Knome Pro → Knome → Flush Cache**[2].



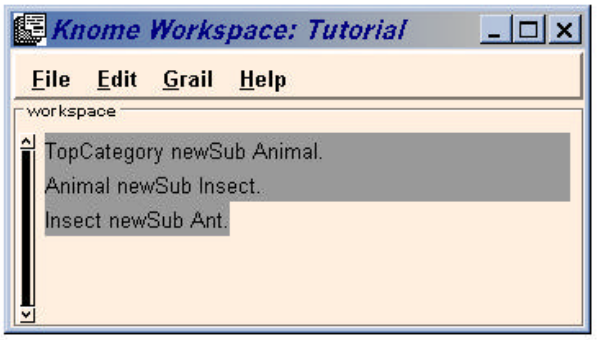
2 It may also be necessary to do this during interactive addition to the model: if some information doesn't appear in the browsers when you suspect it ought, try flushing the cache using this option.

2.3.3 Creating a new model

Before you can proceed, you must first create a new model To do this, select **Knome Pro →Model→New model…** and enter 'Tutorial' as the name of the model in the dialog box that pops up.

Once the model has been created, we are ready to actually model in GRAIL. Usually, the GRAIL is stored as a central set of source files, however in this tutorial we shall be entering GRAIL expressions into a 'workspace' - a scratch area for compiling GRAIL expressions, the contents of which can be saved as a file.

First create a GRAIL workspace (**Knome Pro →Editing→Workspace**) and then save the empty workspace via the **Knome Workspace→File→Save As…** menu. Type the text as shown in the screen-dump below. The GRAIL source then must be 'evaluated', or 'compiled', by using the **Compile** option from the middle mouse-button menu. In other words, the GRAIL expressions you have entered are parsed into individual calls to the Terminology Server; they are then passed to the server and cause the GRAIL engine within the server to extend the model.

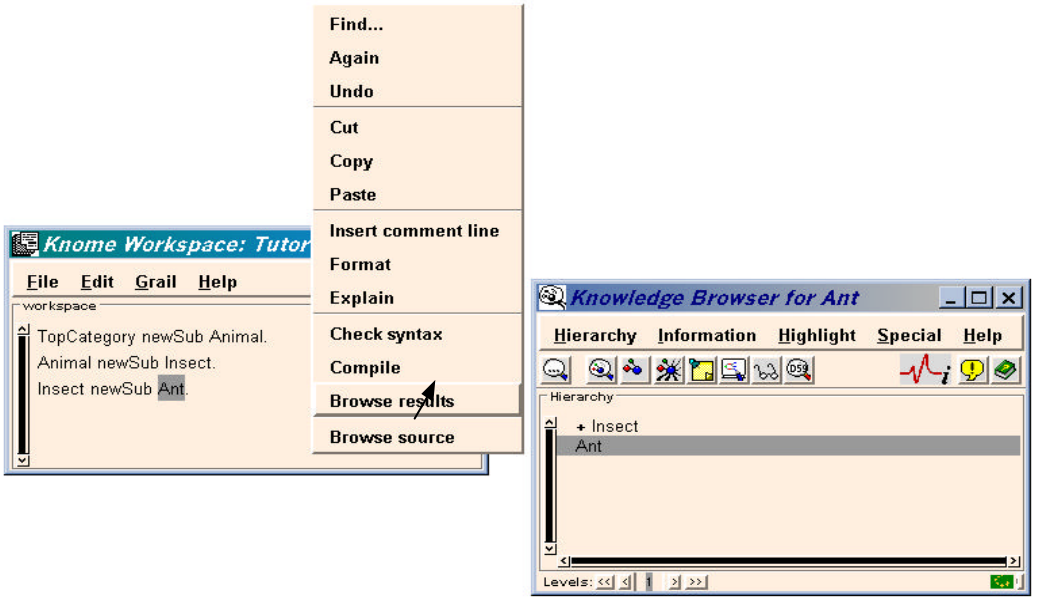


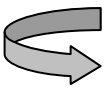
This is a super-simple model - all we have said is that there is a notion of an Ant, which is a kind of Insect, which is, in turn, a kind of Animal. This is just to introduce you to the tools.

2.3.4 Viewing the model

The primary tool for examining a model is called the Concept Browser. The browser always shows the hierarchical structure of the concept of interest (the browser's 'focus'), as well as offering the option of viewing other properties that may apply to the focus.

There are several ways of opening a browser, either using the menu option from **Knome Pro →Browsing→Browse Knowledge**, or you can highlight the concept you are interested in and use the **Browse results** option from the middle mouse-button menu from a GRAIL workspace or file editor:





Notice that the menu options apply to whatever is highlighted in the workspace. The **expand (> and >>)** and **contract (<< and <)** buttons are used to explore levels in the hierarchy (or try double-clicking concepts in the hierarchy).

The indentation of elements within the body of the browser indicates the hierarchical relationships that are in force. You might like to experiment with options on the middle-mouse button menu over the hierarchy, but important facilities will be introduced through the tutorial as you need them. See also 'The Grail KnoME: Overview and User Guide' for full descriptions of the browsers and other tools.

2.4 Syntax

2.4.1 Use of Brackets

One of the confusing things about the existing GRAIL syntax is that it uses three different kinds of brackets for three different purposes.

You will see different types of parentheses used in the compiler;

() Used to enclose a sub definition in long complicated definitions -' ordinary' brackets'. An example not using GRAIL syntax might be:
(people who are tall) who have (hair which is grey).
In this expression
(people who are tall)
defines a kind of *people* and
(hair which is grey)
defines a kind of *hair*. Together the two sub-definitions are used to define "tall people with grey hair", or if you prefer "tall grey-haired people".

[] Used for 'expandable lists' Expandable lists are a kind of short-hand ; these lists are expanded by the compiler before being passed to the underlying system. This will be explained more fully below, but the pattern is that
tell [mary john karen].
means exactly the same as
tell mary.
tell john.
tell karen.

<> Used to group criteria in definitions. Again, as another example not using proper GRAIL syntax,
People who <are tall have grey hair>
would be "people who are tall and have grey hair" or again "tall grey-haired people". Note that the 'and' is understood inside <...>.

2.4.2 Use of full stops

All GRAIL expressions must end with a full stop. If you don't have a full stop at the end of a statement, you may get various odd syntax errors which are not always helpful. If you can't figure out what is wrong with the syntax or why the GRAIL compiler does not recognise something, look for a missing full stop.

2.4.3 Comments

GRAIL treats anything between double quotes "..." as a comment. By convention long comments are put in the form:

```
">>>>>>>>>>>>>>>>>>>>>>
> This is a comment
> extending over two lines
<<<<<<<<<<<<<<<<<<<<<<"
```

2.5 A word about Vocabulary

2.5.1 Changes in Vocabulary

The GRAIL vocabulary has evolved over the years. This tutorial attempts to be in line with current usage, but some older documentation may use slightly different words.

Two changes are perhaps worth noting. The earlier versions of this tutorial referred to 'entities' where this tutorial refers to 'categories'. Earlier versions referred to 'explicit subsumption' where this tutorial refers to 'asserted subsumption'.

Side-bar: GRAIL's vocabulary compared with similar systems

GRAIL's vocabulary was carefully selected to avoid clashes wherever possible with the vocabulary used in object oriented programming languages because GRAIL is itself usually implemented in an object oriented language. However, this sometimes causes confusion to new users. We hope the following will help.

In GRAIL, notions like "Dog", "House" or "Swimming" are known as "categories" and usually written with an initial upper case letter. Categories correspond to 'classes' in object oriented languages, 'types' in Conceptual Graphs, or 'concepts' and 'concept names' in description logics.

GRAIL uses the word 'attribute' for the things which link categories together, corresponding to 'relation' in some object-oriented languages or 'role' in description logics. Conceptual Graphs also use the word 'attribute' in this way.

Two categories linked by an attribute make up a 'statement', or in the older usage, a 'triple'. Some object oriented design formalisms refer to statements as 'relationships' Conceptual Graphs use the word 'statement' in the same way as does GRAIL. Description Logics are formulated in such a way as to avoid the notion.

If you find the vocabulary confusing because of previous experience, please refer to the table in the paper 'The GRAIL Concept Modelling Language for Medical Terminology'. If you find it confusing because you are new to all such systems, please bear with us, and we hope it will become clearer as you work through the tutorial. If you still find it confusing, please get in touch with us.

3. An Introductory GRAIL tutorial

3.1 An example application area

3.1.1 What we want to do

GRAIL was originally designed for medical applications. However, the medicine often gets in the way on first acquaintance. This tutorial uses, as an example, a simple model of vehicles and transport.

Consider that we want to model a terminology about vehicles. It might contain:

Bus, Car, Train, Tram

The first task is to think how these might be divided up into categories. There are any number of possible ways to classify any collection of concepts, but we need to pick one to start with.

Let us say that we choose to classify them by whether they run on rail or road.

Then we might produce a hierarchy that looked like:

RoadVehicle
- Car
- Bus

RailVehicle
- Train
- Tram

We usually like to have a general abstract concept for any part of our terminological model or 'ontology'. We said at the beginning, that these were all kinds of 'vehicles' so we might add to this and rename it to get:

Vehicle
- RoadVehicle
  - Car
  - Bus
- RailVehicle
  - Train
  - Tram

By convention in GRAIL, everything which is about the topic or 'domain' we are considering is placed under DomainCategory, and DomainCategory comes under TopCategory which is built into the system. It is a good idea to separate out branches of an 'ontology' so we shall create an extra TransportCategory in between DomainCategory and Vehicle, So what we want to end up with at the end of the first stage is a hierarchy that looks like:

TopCategory
- DomainCategory
  - TransportCategory
    - Vehicle
      - RoadVehicle
        - Car
        - Bus
      - RailVehicle
        - Train
        - Tram

3.1.2 Doing it in GRAIL

We are going to build an entirely new model, so the first thing to do is to clear any existing model, as described in Section 2.3.2. and then create a new model called Tutorial 2 as described in Section 2.3.3.

Model resetting can also be useful if the model ever gets into a strange state while you're experimenting with the tools and model. Simply reset {xe "reset "}it to throw it all away and start again.

Now create a new file for the statements you will use to create the model by opening a workspace (as described in Section 2.3.3), and giving a new filename when asked.

3.1.2.1 Creating elementary categories

In an empty model, there are a few built-in, primitive categories. The first is TopThing{xe "TopThing"}, the root of the model, or network. Every concept must have at least one parent - except for the built-in starting pint, TopThing. Any other category is meaningless until it is connected to the model. TopCategory is a built-in child of TopThing to act as the starting point for all new categories

Type the following statements

```
TopCategory newSub DomainCategory.

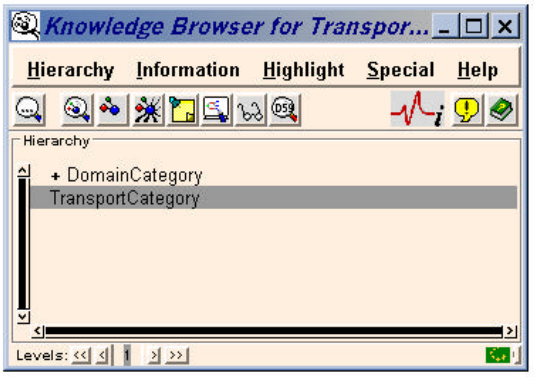
DomainCategory newSub TransportCategory.
```

Do not forget the full stops at the end of each statement. Upper and lower case are significant, for example “Topcategory” is different from “TopCategory” and will not work.

Now select these two statements with the left mouse button and then use the middle mouse button to select **compile** for the selection; this compiles these two expressions.

The operation newSub allows the creation of an “elementary concept”; it is given an identifier, and specifies at the same time where it appears in the concept structure hierarchy. After it has been defined in this way, it can be accessed by its identifier. We've created two new categories in this way - DomainCategory and TransportCategory. TransportCategory will be our starting point for the example. DomainCategory is a standard GRAIL convention which simply used for organisation[3]. By convention, the highest proper category in any GRAIL model is DomainCategory.

Now select TransportCategory in the workspace and **Browse results** from the middle mouse button menu. A browser should open which looks like:



3 ‘Domain’ is used in the AI sense of the field which we are modelling, in this case ‘transport’ or ‘vehicles’. There is a second meaning as in ‘the domain of a relation’ which is entirely distinct and not relevant at this point.

3.1.2.2 Subsumption

Subsumption {xe "Subsumption "}is a posh word for the "is-a" relation. If B is a kind of A, then we say "A subsumes B", and "B is subsumed by A". As an informal example (which we'll see more formally later), Man is a kind of Person, so we can say that the concept of Man is subsumed by Person. It's a good idea to have fixed in your mind which way this relationship goes, as it sometimes seems that intuitively it should be in the other direction[4]. One way to remember it is that if B is a kind of A and sits "below" A in the hierarchy, then we will know more of the properties of B. Thus the properties of A are a *subset* of those of B and A *subsumes* B. There are two different forms of subsumption. One, *asserted* subsumption, we'll meet now; the other, *formal* subsumption, will explain up later.

Add an category that is subsumed by TransportCategory and embodies the concept of Vehicle.

```
TransportCategory newSub {xe "newSub "}Vehicle.
```

This is **asserted subsumption{xe "asserted subsumption"}**.. All we are saying is that there is some concept - Vehicle - that is "a kind of" TransportCategory. Note that the only interpretation of this relationship comes from *our own* linguistic interpretation of what Vehicle is except that it is a kind of TransportCategory. To illustrate this further, let's extend the model to a few specific types of vehicles.

Type in and evaluate the following:

```
Vehicle newSub RoadVehicle.
        RoadVehicle newSub Bus.
        RoadVehicle newSub Car.

Vehicle newSub  RailVehicle.
        RailVehicle newSub Train.
        RailVehicle newSub Tram.
```

There are several points to note about the above expressions. Layout is ignored by the compiler. However, it is important to others trying to read your files. Good layout can help a great deal in clarifying awkward situations. In general, it is easier to read if you indent each level in the newSub statements is indented as shown.

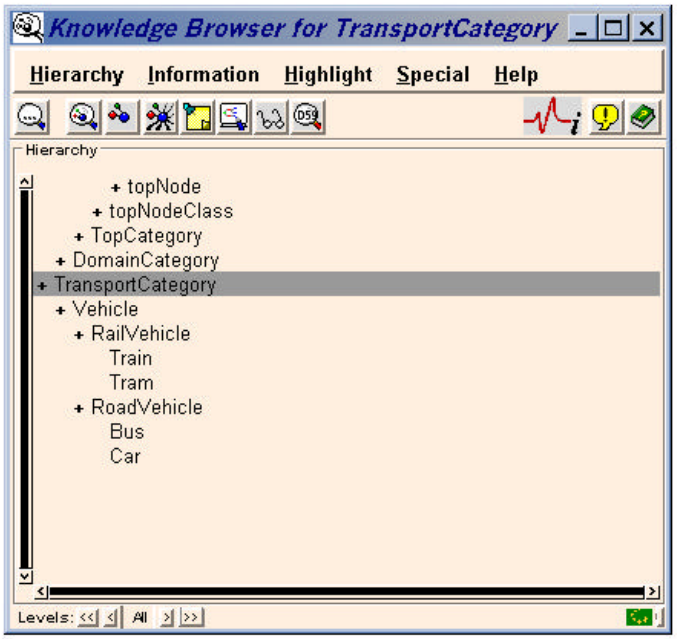
Open a browser on TransportCategory by highlighting it and selecting GRAIL browse from the middle mouse-button menu. Click on Vehicle in the browser and then use the **expand (>)** button on the browser to see what it subsumes. Click on and expand each of the subsumed categories in turn (as a short cut, use the **expand to all levels button (>>)**). You should see a browser as shown below.

This is the 'asserted subsumption hierarchy' or 'asserted taxonomy' for your model so far. It is called the 'asserted subsumption hierarchy'[5] because all of the subsumptions have been created by asserting them with a newSub statement without any reason being given. In the next section we shall see how to create additional 'formal subsumptions'.

---

4 Some object oriented tools use the phrase "is a generalisation of" or "generalises to" or just "generalisation" for "subsumes".

5 In earlier versions of the documentation, this was termed 'explicit subsumption', but this terminology has been found confusing.



3.2 Formal Subsumption: Statements, Definitions, and Composite Categories

To explain the next steps requires adding descriptive information and definitions to our simple model. To do so, we will take a small detour and build a small model of kinds of people before we come back to relate people and vehicles.

3.2.1 Creating attributes

Attributes {xe "**Attributes** "**}**are used to describe relationships between categories, and define new categories. First add a new category to represent the concept of a Person:

```
DomainCategory newSub LivingThing.
  LivingThing newSub Person.
```

Attributes will allow us to talk about, for instance, a Person that has sex male - a Man in other words. For that particular example, we *could* consider simply providing two new subsumptions of Person, Man and Woman. However, the use of attributes provides us with more power. As an illustration of this, consider the concept of people who drive buses. We could provide a new subsumption of Person, CarOwner, which would embody this concept. What if we then want to look at the relationship from the "other end"? We would have to also provide a new subsumption of Car, Car which isOwnedBy Person. One can easily see that this could lead to a proliferation of new categories that have to be added explicitly. In addition, this fails to capture the fact that we may also want to talk about Vehicle drivers in general, as they might all have something in common.

What we want to do is provide a mechanism for *defining* owners, men, women, etc. systematically. This also has the advantage that we know why a man is different from a woman, at least in our model: because men and women have different sex. Likewise we will know what makes owners different from other persons: they own something.

First, some housekeeping. (The syntax in this section is not particularly nice, so you will just have to bear with us.) Type the following into a workspace, select it and evaluate it.

```
Attribute newAttribute DomainAttribute inverseDomainAttribute allAll manyMany.
```

Attribute is the top of the attribute hierarchy and comes built-in in every new model. (Perhaps we should have called it TopAttribute, but we didn't.) The expression that you typed

introduces a new attribute DomainAttribute that will be used in a similar way as DomainCategory, as a top-most attribute to hang all our other attributes from.

Now let's introduce a more useful attribute:

DomainAttribute newAttribute hasOwner isOwnerOf allAll manyOne.

This introduces a new attribute hasOwner. It has an "inverse{xe "inverse"} isOwnerOf that will allow us to talk about the relationship from the "other end". The keyword allAll is obsolete but persists in our compilers. The keyword manyOne says that a vehicle can have only one owner, but one owner can own many vehicles. This is referred to as the **cardinality {xe "cardinality "}**of the attribute. Other cardinalities are manyMany oneMany oneOne. Another way to describe attributes with cardinality manyOne or oneOne is to say that they are 'single-valued'[6]

3.2.2 Sanctioning statements

So far, we've only considered **elementary categories{xe "elementary categories"}** - those categories that we've explicitly defined and placed in the hierarchy. Using attributes, such as hasOwner declared above, we can construct **composite categories{xe "composite categories"}.** However, we can't just make composite categories at will: asserting that a Person is owned by a Bus is obviously nonsensical and should be prevented! Before it can be used in a definition, an attribute has to be **sanctioned{xe "sanctioned"}**. Sanctioning **{**xe "Sanctioning "**}**is the mechanism that describes how categories can be put together with attributes to make definitions, and thus specify what it is sensible and meaningful to say.

In GRAIL, there are two levels of sanctioning statements (plus a third level of 'necessary' statements which we will describe later):

- Grammatical**{**xe "Grammatical"**}**
- Sensible**{**xe "Sensible"**}**

To sanction the hasOwner relation at the grammatical level, we create a new **statement {**xe "**statement** "**}**by evaluating the following statement:

Vehicle grammatically hasOwner LivingThing.

This statement says that in a grammatical sense, a Vehicle can have a owner, which is a LivingThing, associated with it. Roughly speaking, this means that it would not be crazy to ask a question about any sort of living thing owning a vehicle, but it may not make much sense or be the sort of thing we would want to turn up in a list of choices in a user interface. We can ask questions about toads that own cars, but unless we are modelling some sort of cartoon world, it isn't very 'sensible'.

So what have we done? We've expressed the fact that in *general*, it's not crazy to talk about vehicles having owners. Not all vehicles will have owners, and not all living things will be owners of vehicles - the further levels of sanctioning will refine these constraints.

RoadVehicle sensibly hasOwner Person.

This statement tells us that for RoadVehicles, it's sensible to talk about them having people as owners.

In order for a 'sensible' statement to be valid, there must already exist an appropriate grammatical statement. For instance, if we had attempted to evaluate the sensible statement before the grammatical statement, an error would have occurred. Note also that categories inherit the sanctions from their parents. As RoadVehicle inherits from Vehicle, it also inherits

6 Note that in the description logic community, the word 'Role' is used for what GRAIL calls 'attributes' and the word 'Attribute' in the DL community is reserved for single-valued roles., i.e. what GRAIL terms 'single valued attributes'. Confused? Sorry about that, but the vocabularies have grown up independently and confusingly.

the grammatical sanction of the hasDriver relation. Likewise Person inherits the sanction from LivingThing. This is an important point to remember.

Note that there can be more than one grammatical statement for the same attribute. It is also possible for grammatical and sensible statements to occur at the same level. For example, if we say

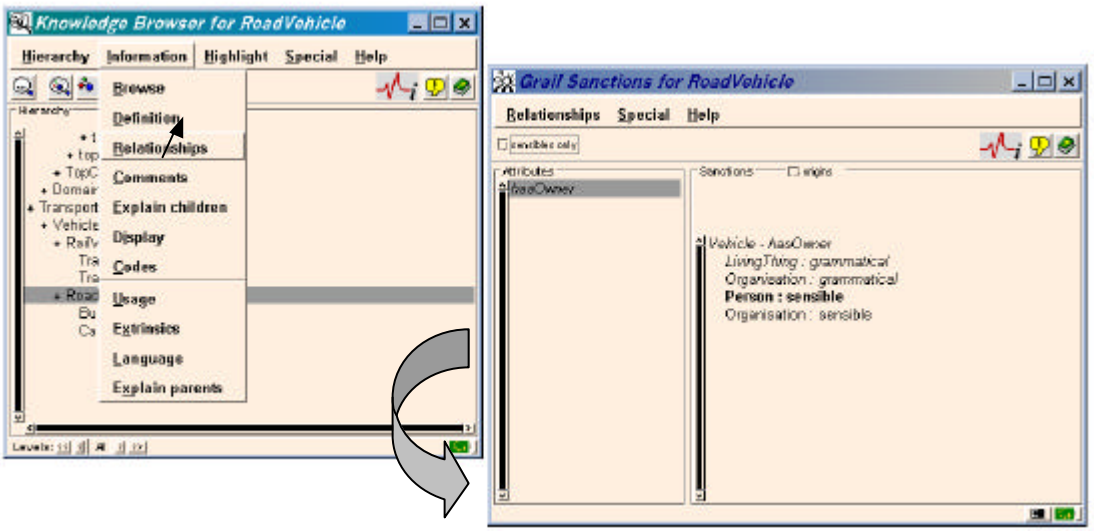
DomainCategory newSub Organisation.

Then we might want to add.

Vehicle grammatically hasOwner Organisation.

Vehicle sensibly hasOwner Organisation.

To view the statements that are in force for a particular concept, use the **Knowledge Browser→Information→Relationships** option from the Concept Browser, as shown below:



The display indicates that there is currently only one attribute sanctioned (hasOwner). Highlight this attribute to display the sanctions; boldface indicates that the sanction has been applied at the browser focus (as opposed to being inherited). When the 'Sensibles only' box is unchecked, the grammatical sanctions are shown in italics. Note that middle-button menu options are available to explore any of the categories displayed as parts of statements.

3.2.2.1 Form of GRAIL statements

All GRAIL statements follow the pattern above:

Topic *qualifierKeyWord* anAttribute Value.

The Topic and Value must be categories and anAttribute must be an attribute. The qualifier keywords grammatically and sensibly have been introduced so far. Other qualifier keywords will be introduced later.

In short, GRAIL statements are triples as in many other languages, plus a qualifier. GRAIL uses the word 'topic' rather than 'object' (as in 'object-attribute-value' triple) in order to avoid confusion with the many other uses of the word 'object'in current circulation.

3.2.3 Defining Composite Categories

Our model now has some elementary categories and an attribute that can be used to construct composite categories. To form composite categories, we use the which {xe "which

"}operator[7]. For example, type the expression below, select it, and browse it by clicking **GRAIL browse** on the middle mouse-button menu.

Car which hasOwner Person.

Note the GRAIL browse option evaluates (compiles) the selection before opening a browser, so there is no need to evaluate it first (though it doesn't matter if you do). The first thing to note here is that this expression, unlike all the others we've met so far, does not *add* any knowledge to the model. Instead, it is a *query* that asks the terminology engine to return the category corresponding to the expression.

In order for the expression to be valid, and the evaluation to succeed, the relation must have been sanctioned at the sensible level. Recall that this sanction could come from a parent - in fact in this case it does, being inherited from the RoadVehicle category. To illustrate this you could try asking for, say, a Train that has an owner.

Notice that the object we asked for has been put in the appropriate place within the hierarchy. It's a child of Car, which is itself a child of RoadVehicle (the hierarchy can be seen by using the **Expand** feature of the browser).

Although the attribute hasDriver has an 'inverse' isDriverOf, it is important to remember that the categories they allow us to construct are different. If we ask for the category:

Person which isOwnerOf Car.

this is very different from a Car which hasOwner Person. The first expression provides us with a kind of Car, the second provides us with a kind of Person. People and cars are very different kinds of things: people are not kinds of cars nor are cars kinds of people.

3.2.4 Cardinality

In section 3.2.1, we briefly mentioned cardinality {xe "cardinality "}- the manyOne argument to the operation that created the hasDriver attribute. Now that we have introduced sanctioning and prototypes, we can describe cardinality properly.

The cardinality of an attribute specifies *how many* categories can be connected by a given attribute. For example, we have now sanctioned the hasOwner attribute to connect Vehicles and Persons[8]. The cardinality we used was manyOne - meaning that any Vehicle has only one Owner, but that a Person could own many Vehicles. This mechanism allows us greater control on which composite categories are allowed; for example, evaluating the following expression will produce an error[9]:

Car which <hasOwner Man hasOwner Woman>

because we are asking for a kind of Car that has two owners. We have disallowed this with our choice of cardinality for the hasOwner attribute.

The other possible values for cardinality are (with their effect):

oneMany — this would mean that Vehicles could have many kinds of Persons as drivers, but a Person could only drive one sort of Vehicle;

oneOne — this would mean that a Vehicle could only have one kind of Person as a driver, and that a Person can only drive one kind of Vehicle;

---

7 In the past, 'composite entities' have been called 'prototypes' and 'particularisations'. Both phrases caused confusion and are being dropped. However, some documentation still uses the old vocabulary.

8 At present, cardinality is specified when creating an attribute; in the future, we may extend GRAIL to allow different cardinalities for particular sanctions.

9 We haven't yet defined Man and Woman - try the evaluation of this expression after evaluating the definitions that appear later.

manyMany this would mean that Vehicles could have many kinds of Persons as drivers, and that Persons could be drivers of many kinds of Vehicles.

3.2.5 Criteria

A **criterion {**xe "**criterion** "**}**is the name given to a combination of an attribute and an category. For instance,

hasOwner Person

Criteria are the building blocks that are used to construct composite categories from elementary ones.

3.2.6 Formal Subsumption

We've already met asserted subsumption, which is where the subsumption is asserted using newSub. For instance, a Car is a kind of RoadVehicle. The other type of subsumption is what is known as **formal subsumption{xe "formal subsumption"}**.. This is where the relationship arises due to the structure of categories and is derived by the internal semantics of GRAIL.

In the earlier example, the category (Person which isDriverOf Car) is formally subsumed by (remember the direction!) Person.

One rule about formal subsumptions is that: any category X always subsumes any category of the form X which hasSomeAttribute Y.

There are other rules that define formal subsumption. We'll not go into detail here, but will return to this topic once we have introduced a few more attributes.

3.2.7 Value Types

So far, we've been dealing with fairly simple basic categories and combinations of those. Next, we'll go on to look at value types. These are again categories that sit in the hierarchy, but we separate them out for convenience in a special section of the model subsumed under a special built-in category, SymbolicValueType.

Introduce a new Value Type**{**xe "Value Type"**}** to represent gender, along with a couple of subsumptions[10]:

```
SymbolicValueType newSub SexValueType.
  SexValueType newSub male.
  SexValueType newSub female.
```

The categories, male and female have been given lowercase initial letters - this is simply a convention that helps to distinguish between value types and other categories.

Now, a new attribute:

```
DomainAttribute newAttribute hasSex isSexOf
        allAll manyOne.
```

The sanctioning expression which describes the way this attribute can be combined with categories:

---

10 Note that we could have abbreviated this to:

```
(SymbolicValueType newSub SexValueType)
        newSub [male female].
```

The parentheses () are used in a standard manner to group expressions. Note that the result of the expression (SymbolicValueType newSub SexValueType) is the category SexValueType to which we can use as the topic of the next newSub operation.

```
Person grammaticallyAndSensibly hasSex SexValueType.
```

We've introduced the statement at both the grammatical and sensible levels with one operation. This is simply a time saving measure, and has the same effect as if the statements were introduced individually, i.e. this is exactly the same as:

```
Person grammatically hasSex SexValueType.
Person sensibly hasSex SexValueType.
```

We can now evaluate the expression

```
Person which hasSex male.
```

Try it, and browse the category to see where it fits in the hierarchy. In a similar way, we'll introduce the concept of age:

```
SymbolicValueType newSub AgeValueType.
  AgeValueType newSub young.
  AgeValueType newSub old.

DomainAttribute newAttribute hasAge isAgeOf    allAll manyOne.

Person grammaticallyAndSensibly hasAge AgeValueType.
```

We can combine expressions

```
(Person which hasSex male) which hasAge young.
```

We can write this expression more simply using the <...> brackets to put both criteria in a criteria set.

```
Person which < hasSex male  hasAge young >.
```

We might translate this roughly into English as “Person which is male *and* young” which means the same thing as “Person which is male which is young”.

Similarly, we can form the composite category:

```
(Person which hasSex female) which hasAge old.
```

However, the cardinality of the hasSex attribute will prevent us doing anything silly. The expression:

```
(Person which hasSex male) which hasSex female.
```

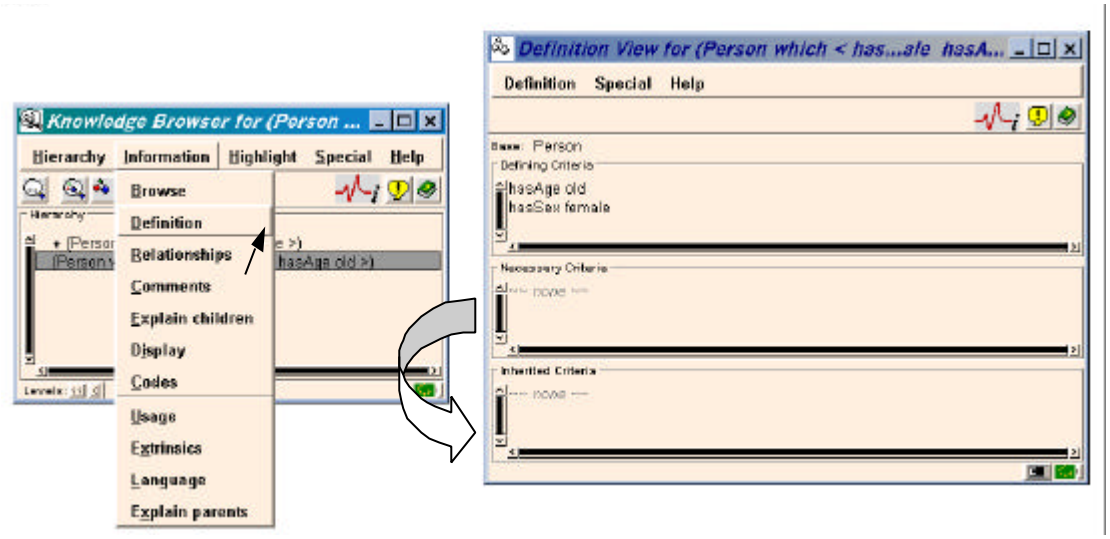
will produce an error if we try and evaluate it, due to the manyOne cardinality of hasSex.

3.2.8 Criteria

To examine the criteria of a category, browse again on

```
(Person which hasSex female) which hasAge old.
```

Select **Knowledge browser→Information→Definition**, to open a display showing the structure of the category:



The criteria that form the definition of the concept are also known as its 'defining criteria'. The top line of the **Definition** pane shows the elementary category from which the category is formed (the 'base'), whilst the remaining entries show each of its defining criteria. (We will explain the lower two panes of this window later).

3.2.9 Naming

As we can already see in our little example, categories can become quite complex. For instance, we can now evaluate the expression:

```
Car which hasOwner
  ((Person which hasAge old)
        which hasSex male).
```

The concept of a car that is driven by an old man. In a more complex model, these expressions could quite easily become unwieldy and difficult to interpret. To help with this, GRAIL allows the naming or 'aliasing' of categories. For example:

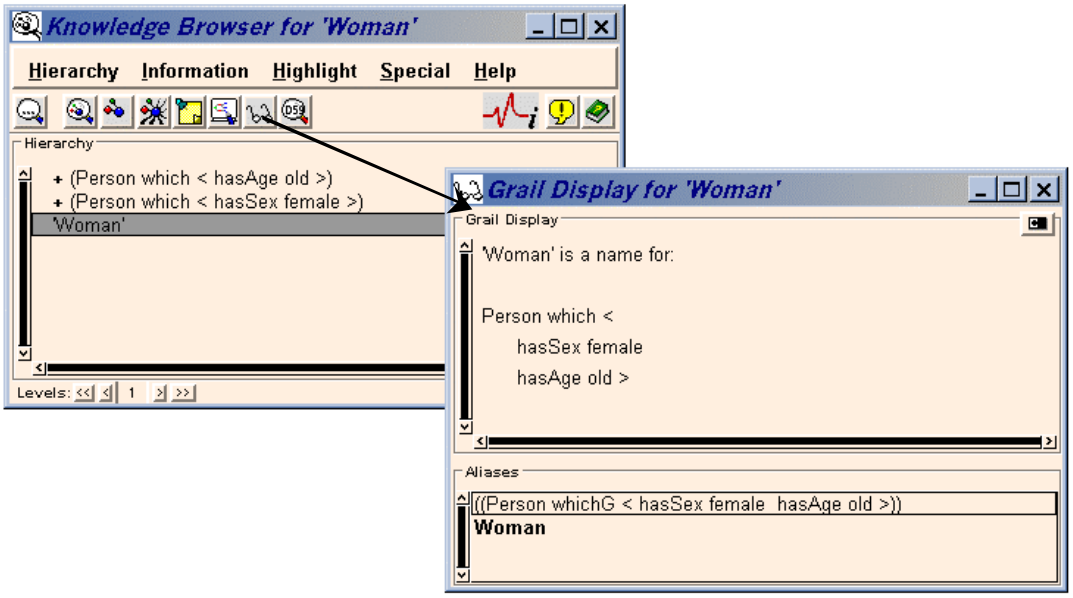
```
((Person which hasSex male) which hasAge young)
        name Boy.

((Person which hasSex female) which hasAge old)
        name Woman.
```

Now, we can simply use the names Boy, Woman, to represent, and to access the complex categories to which they refer.

To practice, add names for the concepts Man and Girl.

Now browse the category Woman and click again on the *criteria* button. Note that nothing has changed. Now click on the **Display** (hint: it's icon on the toolbar is like a pair of specs) button to obtain another view of the structure of the browser's focus category:



Note that in the Concept Browser hierarchy display, Woman appears inside quotes: this is simply a display convention to indicate that it is a name for a complex category.

3.3 Necessary statements and Canonical form

3.3.1 Necessary statements

So far, we've seen the two kinds of sanctioning statement - grammatical and sensible. We now meet the third kind of statement, the **necessary {**xe "**necessary** "**}**statement. This is used for situations where a criterion is mandatory but not part of the definition. For instance, we may expect that all owners of road vehicles must be old (for the purposes of our example, 'old' is considered to be 'old enough to drive' - this does not necessarily reflect the actual views of the management!), We can incorporate this into the GRAIL model as follows:

(Person which isOwnerOf RoadVehicle) necessarily hasAge old.

Note that the category on the left is a complex category - previously all our statements involved basic categories. A statement can have any valid category expression on the left or right hand side. Now try browsing the expression:

(Person which isOwnerOf RoadVehicle).

Click on the **criteria** button. This time, in the **Definition** pane you'll notice the criterion isOwnerOf RoadVehicle. However, in the **necessary criteria** pane, the criterion hasAge old will also be present, telling us the extra information that we've just added to the model. The lowest pane (**inherited criteria**) shows any necessary criteria that the browser's focus may have inherited from its parents, illustrated in the next section.

If we now attempt to evaluate:

Girl which isOwnerOf Car.

The evaluation will fail as the criteria hasAge old and hasAge young would both be present in the result of the expression and are conflicting. Recall that hasAge has cardinality manyOne.

3.3.2 Canonical Forms

Consider the expression:

Man which hasSex male.

This is a well formed[11] and sensible expression: a Man is a (kind of) Person, and can thus be related to a SexValueType via hasSex. In addition, there is no cardinality problem as we're not asserting that a Person has two different sexes. However, we can see that the criterion hasSex male is redundant as we already know that about a Man because hasSex male is part of the definition of the category Man.

The terminology engine is able to reason in this way as well, and reduces categories to a **canonical form{xe "canonical form"}**. This consists of a basic category and a collection of criteria with any 'unnecessary' criteria like those above removed.

Try browsing the expression:

Man which hasSex male.

You will see that the browser opens simply on the category, Man.

3.3.2.1 Necessary statements and canonical forms.

Consider the naming:

(Person which isOwnerOf Car) name CarOwner.

If we now look at the categories:

CarOwner.

CarOwner which hasAge old.

They will be the same. CarOwner already has the criterion hasAge old due to the necessary statement introduced earlier, so the system has reduced them to the same category.

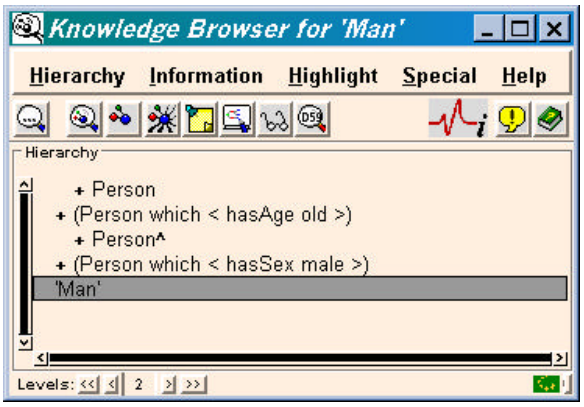
Click on the **criteria** button. The criterion hasAge old appears in the **inherited criteria** pane because it is inherited from one of its ancestors in the subsumption hierarchy: Person which isOwnerOf RoadVehicle.

To summarise, a canonical form is a starting elementary category known as the 'base' along with a collection of criteria in which all redundant information has been removed. If the GRAIL browser is invoked on a which expression, the compiler will reduce it to canonical form before opening the browser. To see the detailed criteria click on the **criteria** button. The criteria are divided into three panes: the defining criteria; the necessary criteria which result from necessary statements about this specific category; and the inherited criteria which result from necessary statements, or definitions of categories, further up the hierarchy.

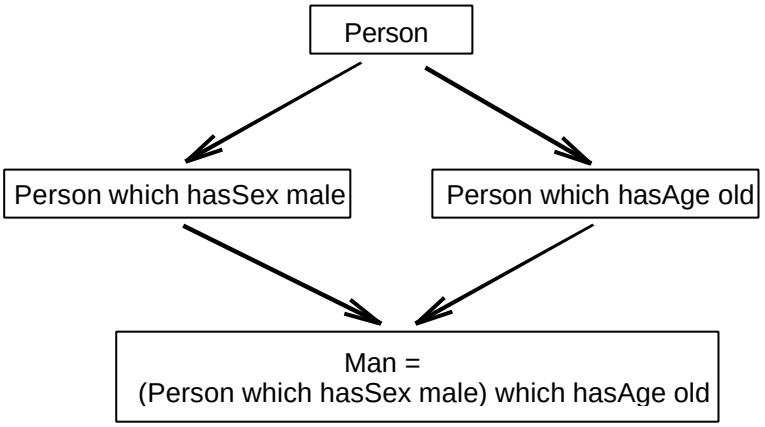
3.3.3 The Hierarchy

At this point, we'll take another look at the hierarchy **{**xe "hierarchy "**}**we've constructed. We'll only show parts of it, as the network is now quite large.

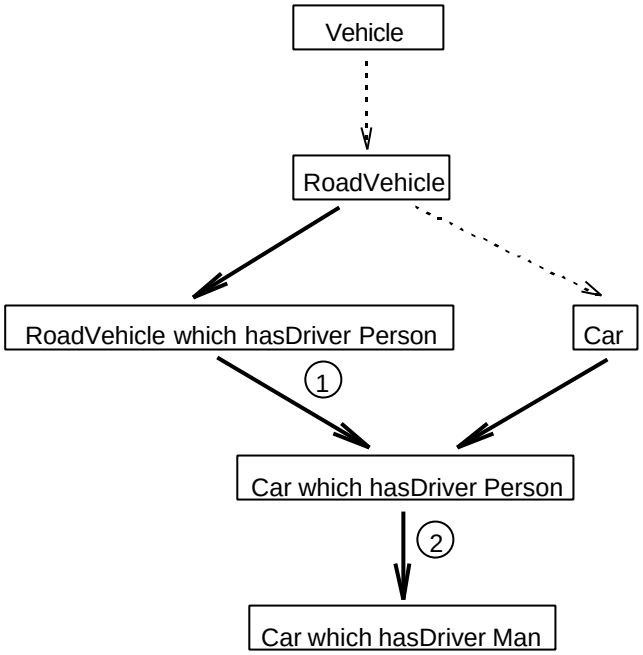
[11] By well-formed, we mean that the expression is syntactically valid, and will be parsed by the compiler.



The browser above shows us some of the position of Man in the hierarchy. Note that all the subsumptions shown above are formal: they are derived from the definition of the Man category. The browser is displaying the following subsumption relationships:



Formal subsumption for composite concepts is essentially a recursive notion using subsumptions of the constituent categories. In the following diagram, illustrating a more complex area of the hierarchy, Subsumption 1 illustrates a subsumption where the 'base' categories (RoadVehicle and Car) are related, while 2 illustrates the situation where the categories in the criteria (Person and Man) are related. The dotted arrows indicate asserted subsumptions, whereas the solid arrows indicate formal subsumption relationships that have been derived by the GRAIL engine.



3.4 Further features

3.4.1 Grammar level Queries (Grammar level Composite Categories)[12]

The which operator allows us to build up composite categories which have been sanctioned at the sensible level. Often we want to ask queries which are more general than are allowed by the sensible level sanctions. For example, we only have sanctions to say that a RoadVehicle can sensibly be owned by a Person. This says nothing about a Vehicle in general. (We might, for example, want to build a model in which RailVehicles could only sensibly be owned by companies.) However, it is perfectly reasonable to want to find all of the vehicles which are owned by men.

However, there is a grammatical sanction allowing us to talk about a Vehicle owned by any LivingThing, or conversely a LivingThing which owns a vehicle. To ask a general query allowed by this grammatical sanction, we use the whichG operator **{**xe "whichG "**}** (G for 'grammatical' categories). **{**xe "*grammatical prototypes*"**}** **{**xe "grammatical query"**}{**xe "grammatical composite category"**}**

We can browse:

Person whichG isOwnerOf Vehicle.

This represents the concept of a Person who owns a Vehicle (remember that this may not be sensible if the Vehicle is, say, a Train or a StealthBomber).

Note that Person which isOwnerOf RoadVehicle is classified automatically under Person whichG isOwnerOf Vehicle. Note the three dots following Person whichG isOwnerOf Vehicle (indicating that it has children). Select it and expand it, and underneath will be 'CarOwner' which will have been classified automatically and put in the correct place in the hierarchy according to the rules for formal subsumption. (Remember, the quote marks indicate that this is a 'named' concept. Click on the **display** button to see the definition or the **criteria** button (hint: it looks like a molecule made of 3 atoms) to see the full list of criteria).

12 For short these are sometimes called 'Grammatical Categories' or for historical reasons 'Grammatical prototypes'.

Grammar level composite categories exist primarily to be used as a query mechanism as shown above. They can also be used for adding general knowledge, but should be used with caution. As an example of using grammar level composite categories for simple rules, consider the idea of insurance ratings for owners of Vehicles:

```
DomainAttribute newAttribute hasInsuranceRating isRatingOf
        allAll manyOne.

(Person whichG isOwnerOf Vehicle) grammaticallyAndSensibly
        hasInsuranceRating IntegerValueType.
```

We don't want just to have a rating for owners of vehicles on the roads, but for all vehicle owners - even owners of kinds of vehicles we have not yet thought of. This statement guarantees that when we do think of new kinds of vehicles and extend the model to include them, then the sanctioning for having an insurance rating will also apply to owners of these as-yet-un-thought-of kinds of vehicles.

3.4.2 Topic and Value necessary statements

We introduced the necessary qualifier earlier to assert necessary, or essential, facts about categories. For example, in the assertion:

```
(Person which isOwnerOf RoadVehicle) necessarily hasAge old.
```

we are saying that anyone who drives a road vehicle must be old. If we look at such an category with the GRAIL browser, the criteria hasAge old will appear in the necessary pane.

What is not obvious, is that the 'inverse' attribute will also have been qualified in this way. If we browse the category old, the criteria isAgeOf (Person which is isOwnerOf RoadVehicle) will be present in the necessary criteria pane. We may not always wish this to happen - some extra classification information has been added to the right hand category of the statement. GRAIL provided two special operations topicNecessary **{**xe "topicNecessary "**}**and valueNecessary **{**xe "valueNecessary "**}**which allow us to specify 'one-way' necessary qualification. Thus we can assert[13]:

```
(Person which isOwnerOf RoadVehicle) topicNecessarily hasAge old.
```

This would place the necessary qualifier on the left hand category but not the right hand. A similar effect could be achieved by asserting:

```
old valueNecessarily isAgeOf (Person which isOwnerOf RoadVehicle).
```

As with the necessary qualifier, the relation must be sanctioned to the sensible level in order for the statement to be valid

The keywords topicNecessary and valueNecessary come from the names for the different parts of a GRAIL statement:

```
Topic qualifierKeyword Attribute Value
```

3.5 Some common questions

3.5.1 What is the difference between a criterion in a definition (a which expression) and a criterion in a necessary statement?

The difference is subtle but important. A which expression defines a new category. A necessary statement adds new information to an existing category. Put another way, the criteria in a which expression are sufficient to recognise a category, and the category cannot be recognised unless all of the criteria in the which expression are found. By contrast, a

13 Note that as we have already asserted the necessary statement for this entity, the statement will have no effect. To see the effect of the one-way necessary try recompiling the model to this point then using topicNecessarily rather than necessary.

category can be recognised even though the information in a necessary statement is unknown, but it is added as soon as the category is recognised.

To take our example, we have said that CarOwners are necessarily old. We can recognise a CarOwner just because they own a car, but as soon as we recognise them, we know that they must be old (or so says this model). If we find something which claims to be both a CarOwner and young, the system will attempt to add that the thing is old because of the necessary statement. This will cause an inconsistency and the system will report an error.

In summary, we can recognise something even if we are ignorant of the information in the necessary statements, but if we have information which contradicts the information in the necessary statements we have an inconsistency.

> **Side-bar for the logically minded** The conjunction of all of the criteria in the definition is true if and only if the category defined is true, *i.e.* the definition is necessary and sufficient for the category defined. Each criterion in a necessary statement is true if the category is true, but there is no implication in the other direction, *i.e.*. each criterion in a necessary statement is individually necessary but not sufficient for the category.

Still confused? Again, bear with us until the end of the tutorial.

3.5.2 What is the model?

There are two things which are often referred to as 'the model{xe "the model"}'. These are:

a) The textual GRAIL statements containing the information presented by the modeller;

b) All the concepts and facts implied by that information and the GRAIL rules compiled into a form that they can be manipulated by the GRAIL engine.

In general it is more convenient to speak of a), the text files, as the 'source files' for the model and b) as the 'compiled model'. Exactly how the compiled model is represented, loaded, and saved depends on the implementation[14].

You look at the source files using a file editor, file list or workspace. (The source files are simply text files, and you can, if you wish, edit them with any text editor.) You look at the compiled model using the browser and other GRAIL tools. Information only gets from the source files to the compiled model when you evaluate an expression or expressions in the source files. (There are also tools to let you evaluate an entire source file or an entire set of source files with a single statement, but we will come to those later.)

The GRAIL engine only makes composite categories when it is asked to, since there are a nearly infinite number of possible composite categories which it could make. If you just look at the model using the browser, you will only find those composite categories which have previously been mentioned or requested by a which expression. In the example, we have never mentioned or requested the category (Car which hasOwner Woman). As a result, we will not find it if we look for it by starting with Car in the browser and then expanding to look at all of the kinds of car in the model. However, if we were to evaluate the expression (Car which hasOwner Woman), the GRAIL engine would immediately make the category and classify it in the correct place in the hierarchy.

14 With the version of the software you are using, the compiled model is simply stored as an internal structure of Smalltalk objects, whereas in other implementations it may be stored externally in a database.

4. Using GRAIL well: A tutorial in basic GRAIL style

4.1 Re-use, application independence, and modelling style

Chapter 3 should have given you understanding of the basics of building GRAIL models. It has introduced all of the basic ideas except for the way in which GRAIL handles transitive relationships (this is covered in section 5). This chapter explains in more detail how GRAIL is intended to be used. GRAIL has been designed to make a certain kind of modelling possible. To get the most out of GRAIL, you need to follow a few simple rules to produce the sort of models intended.

Our goal is to produce terminological models which are 'application independent' and 're-usable'. To do this, we want to make as much information explicit in the model as possible so that applications can use it, and we want to make as few decisions as possible on the basis of individual applications. We want to avoid 'building in' a bias towards one particular way of looking at the model.

Of course this can never be perfect, and no model will ever be re-usable over the entire range of human experience. However, our goal is to produce models which cover a range of applications in a given area rather than models which are tuned just to just one, very specific application.

The main strategy for producing application independent models is to break our hierarchies of elementary categories into independent sub-hierarchies which do not overlap and then to fit them together using definitions and necessary statements. The result of the classification and formal subsumption may be a very complex multiple hierarchy, but we know why each category is classified the way it is because the information is explicit in the definition and necessary statements.

We have to have some asserted subsumption: we have to start somewhere. But over-use of asserted subsumption is the enemy of re-use, because the reason for asserted subsumption is 'tacit' and invisible to the computer. Personally, we may or may not know or remember why we asserted that a given elementary category should be classified in a certain way, but it is certain that the information is *not* available to the system or to applications. We want to build the minimum of tacit assumptions into the asserted hierarchies. We want to make as much as possible explicit through the use of definitions and necessary statements.

4.2 The example model revisited

In the example we chose to classify vehicles into road vehicles and rail vehicles. But this is only one of many different ways of classifying vehicles. For example we might have been interested in what fuel they use, whether they have wheels, or whether they are used for transporting people or freight.

The choice of what criteria to use for the asserted taxonomy is entirely arbitrary, but it should be consistent. Mixing different principles in the asserted hierarchy means that it is impossible to determine why something is classified in a given place.

4.3 Extending the model to a second classification

Let us consider that we also want to classify vehicles by the type of fuel they use.

Then the best way to do this is in three steps:

i) Create a hierarchy of kinds of fuel and an attribute linking that hierarchy to vehicles;

ii) Use necessary statements to describe which kinds of vehicle use which kinds of fuel;

iii) Create the general categories such as 'petrol driven vehicle' as composite concepts using which or whichG expressions and giving them convenient names.

To illustrate, create a hierarchy of kinds of fuel by typing the following into the file editor and evaluating it. (Anything between "..." is a comment and is ignored by the compiler.)

"Step i)"

```
DomainCategory newSub Fuel.
  Fuel newSub Petrol.
  Fuel newSub Diesel.
  Fuel newSub Electricity.
```

"Step 2)"

```
Attribute newAttribute hasFuel isFuelOf allAll manyOne.

Vehicle grammaticallyAndSensibly hasFuel Fuel.

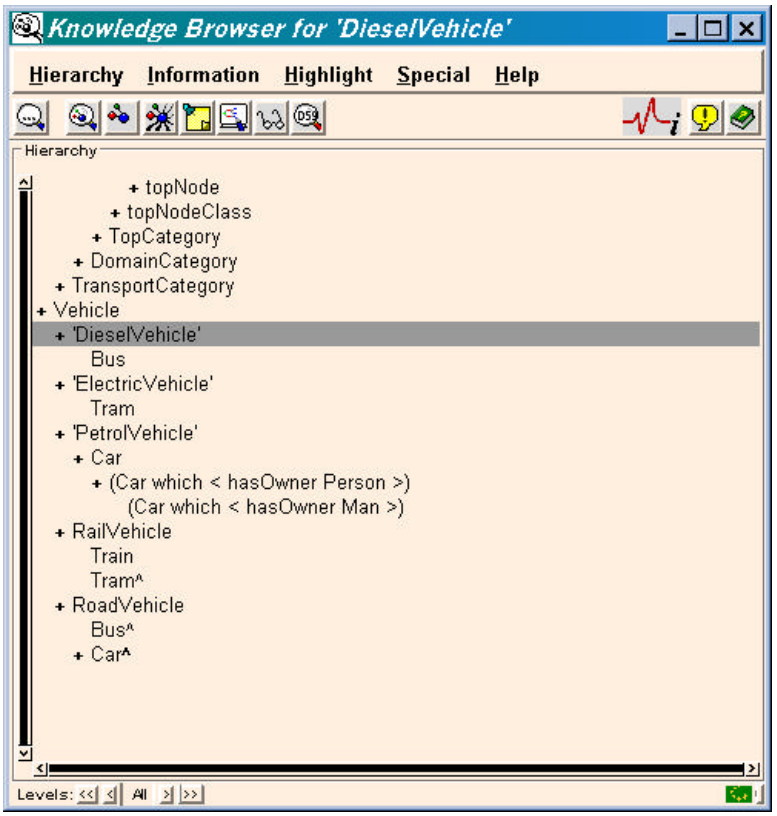
Car topicNecessarily hasFuel Petrol.
Bus topicNecessarily hasFuel Diesel.
Tram topicNecessarily hasFuel Electricity.
```

"Step 3)"

```
(Vehicle which hasFuel Petrol) name PetrolVehicle.
(Vehicle which hasFuel Diesel) name DieselVehicle.
(Vehicle which hasFuel Electricity) name ElectricVehicle.
```

Now browse Car. Note that Car is now classified under both PetrolVehicle and RoadVehicle. We often say that it has two 'parents'.

Also browse DieselVehicle. Finally, browse Vehicle and use the middle mouse-button to **show all levels,** as illustrated below:



(The carets '^' beside categories mean that they have already appeared further up the hierarchy and are visible on this screen.).

Notice that only Train has not been multiply classified because we have not said anything further about trains, because trains can be fuelled by either Diesel or Electricity or Petrol. (We have simplified the world by saying that all buses use diesel and all cars petrol, but that is convenient for the illustration.)

Note therefore that what we have found is not all of the vehicles which *might* use a particular fuel - that list could be very long indeed, but all of the vehicles which anybody has thought it worth modelling which use a particular fuel.

If we decide it is worth modelling diesel and electric trains, we can add the statements.

(Train which hasFuel Diesel) name DieselTrain.
(Train which hasFuel Electricity) name ElectricTrain.

Then DieselTrain and ElectricTrain will be multiply classified as required.

4.4 Reformulating the model

Browse Car again and look at the criteria by pressing the **criteria** button. Note that the criteria tell you that cars run on petrol but not that they are road vehicles or anything about what it means to be a road vehicle. That information is implicit in the hierarchy.

It would probably be better to rethink the model to make both pieces of information explicit. (Don't be afraid to rethink models and rebuild them from scratch. Most models have been rebuilt from scratch several, sometimes dozens of, times. You won't get it right the first time).

Let's reformulate the model so that both the fuel and whether it runs on rail or road are of equal status.

What we want to do:

i) Remove RoadVehicle and RailVehicle from the asserted hierarchy.

ii) Define attributes and values so that we can describe a vehicle running on road or rail.

iii) Describe each vehicle using the attribute as running either on road or rail using necessary statements.

iv) Define the idea of RoadVehicle and RailVehicle in terms of those attributes and values using which expressions.

First reset the model so you get rid of all the existing definitions as described in section 2.3.2.

Now make a new copy of the file you are using. (Sometimes it is easier to start from scratch, but in this case we can probably manage to keep most of what we have already done.) First, if you have not done so already, save the file in which you have been working by selecting the **save** option on the middle-mouse button. Then use the **save-as** option to save it to a new file with a different name.

4.4.1 Removing RoadVehicles and RailVehicles (flattening the asserted hierarchy)

First, since are going to use formal subsumption instead of asserted subsumption to classify the different kinds of vehicles, then we need to remove the elementary categories RoadVehicle and RailVehicle from the asserted hierarchy.

Edit the initial commands in the file until they look as shown below. (The cut and paste keys can make this a quick operation.)

```
TransportCategory newSub Vehicle.
        Vehicle newSub Bus.
        Vehicle newSub Car.
        Vehicle newSub Train.
        Vehicle newSub Tram.
```

4.4.2 Creating the attributes and values

Then let us create the attributes and values needed. We need something to call the attribute which describes whether a vehicle runs on rail or road. It is often not easy to come up with a name. We shall use travelsOn for the attribute and TravelMediumType for the rail, road etc. (We might extend this to air at the same time with little effort.)

Add to the file immediately after the asserted hierarchy a new attribute definition, a new value type, and the appropriate sanctions:

```
DomainAttribute newAttribute travelsOn isTraveledOnBy allAll manyOne.

SymbolicValueType newSub TravelMediumType
  TravelMediumType newSub air.
  TravelMediumType newSub rail.
  TravelMediumType newSub road.

Vehicle grammaticallyAndSensibly travelsOn TravelMediumType.
```

4.4.3 Describing the elementary categories.

Next we need to is add the descriptions of the different travel mediums. Immediately after the above add:

```
Car     topicNecessarily    travelsOn road.
Bus     topicNecessarily    travelsOn road.
Train   topicNecessarily    travelsOn rail.
Tram    topicNecessarily    travelsOn rail.
```

4.4.4 Creating the abstractions

Now all we have to do is define RoadVehicle and RailVehicle, but this time they are formal definitions so the model contains the explicit information on what they mean. Immediately after the above add:

```
(Vehicle which travelsOn Road)      name RoadVehicle.
(Vehicle which travelsOn Rail)      name RailVehicle.
```

4.4.5 Sorting out the order of statements in the source file.

Everything in GRAIL must be defined before it is used. This is one of its strengths and one of its weaknesses, but it often makes modifying files a bit painful.

Find the first point in the file where you have first used RoadVehicle or RailVehicle (after their introduction). It should be the statement

```
RoadVehicle sensibly hasOwner Person.
```

The new definitions must go before this point.

Since the new definitions are replacing some of the things which were in the original asserted hierarchy, it is good practice to place them immediately after the asserted hierarchy. This is what we have done in the example.

However, this still leaves a problem, because the definition of DomainAttribute is further down the file. (You will find this out if you do a **check syntax** on the file because you will get an error message).

Move the statement defining DomainAttribute up until it comes after the asserted hierarchy and before the new section you have just added.

(Good style organises source files in a sensible order so this happens as rarely as possible. More about this in later volumes. But for the tutorial, this is sufficient.)

4.4.6 Checking the syntax

Select the entire file (either by clicking three times in the upper left hand corner, or by extending a selection by clicking with the shift-key held down). Choose **check syntax** from the middle-mouse button menu. If you get errors it will be because either you have mis-typed something or that some statements are out of order so that some category or attribute is not defined before it is used.

It may take you several tries to get the syntax correct. Don't worry about it. GRAIL is an unforgiving language. That means that errors get detected early. It also means that it can take considerable effort before you have a clean file to compile.

(The syntax checker does not guarantee that a file compiles without errors. There can still be contradictions and other 'semantic' errors which can only be recognised when the expression is fully compiled. But it saves much time and often gives more helpful error messages.)

4.4.7 Checking the result

Once the syntax is checked, again select the entire file and **compile it**.

Browse Car and Vehicle again to be sure that you have the same overall result as previously. Check the criteria windows to see that you now have explicit criteria representing whether a vehicle travels on rails or road. Note that RoadVehicle and RailVehicle now appear in single quotes indicating that they are named composite categories rather than elementary categories.

4.5 Final Remarks on the Example

4.5.1 Further reformulation of the model? The need for an 'ontology'

Is this the ultimate way to formulate a terminological model of vehicles? Almost certainly not. Most obviously, the concepts labelled 'air', 'road', 'rail', etc. have been used in a very peculiar way. Almost anything else we might want to say about roads, rails, or air would not be sensibly said about a ValueType as used here.

To go further, we need a larger framework or 'ontology' into which to fit the various notions we come across. ('Ontology' is a posh word misappropriated from philosophy. In philosophy it means the study of meaning. In computing, it means the high level structure of a terminological model.)

There is a separate tutorial on the GALEN ontology for medical terminology which explains how it is organised. There is also a detailed manual and set of papers available as GALEN-IN-USE Deliverable 5.1. However, this is sufficient for now.

4.5.2 Natural kinds and elementary categories

Which things should be represented by elementary categories introduced using newSub and which things should be defined using a name statement? There is no hard and fast rule. Philosophers argue about whether or not there is any such thing as an elementary concept. However, any computer system must start somewhere from a hierarchy of elementary things.

The answer depends partly on the application and partly on the nature of the thing being represented. Philosophers and psychologists speak of many things which are 'natural kinds' - we know what they are when we see one but would be hard put to give a definition. Notions such as 'dog', 'child', 'tree', 'red' are usually thought of as natural kinds. It might be possible to define them in some sense for some purposes, but that definition might not correspond very well with common usage, and we certainly don't recognise them by checking their definition. On the other hand, ideas such as or 'parent' - a person who has children - or 'car-owner' - a person who owns a car - are relatively straightforward to define, at least for the purposes of a convenient model to cover 95% plus of all usage.

Roughly speaking, it is a waste of time and usually counter productive to try to define a representation for natural kind in GRAIL. Natural kinds can be described by necessary statements, but they will almost always be elementary categories. The reverse is not true. There may be many things in a model which could be defined, but whose definitions are irrelevant to any likely application. Alternatively we may just not yet have refined the model that far. A common strategy is to put things in first roughly as elementary categories and then come back and define them more carefully later.

4.5.3 When there must be multiple parents in the asserted hierarchy?

Ideally, no elementary category would have more than one elementary parent - i.e. no elementary category would be subsumed by more than one other elementary category. Put another way, the asserted hierarchy ought to be a pure tree.

Unfortunately, GRAIL is not sufficiently powerful to allow us to make this restriction absolute. In fact the GRAIL compiler will allow you to add many elementary parents to an elementary category, but it is in general a very bad idea to do so because it throws away most of the power of GRAIL.

There are normally only two places where there should be multiple parents in the asserted hierarchy: at the very top and at the very bottom.

At the very top, there is sometimes a need for very general categories which cut across other categories. An example from the medical experience is the notion of a 'disorder' which can be anything from a tumour to a temperature. It cuts across so many other categories that it is easier to create a second supplementary structure than to try to reshape the rest to fit.

At the very bottom, there are sometimes values which are not mutually exclusive. For example in some versions of the medical hierarchy 'both-sided' is made a child of both 'left-sided' and 'right sided'.

The keywords addSuper and addSub can be used to add an additional asserted parent or child to an existing elementary category. Note that they may only be used between elementary categories. For example:

```
SymbolicValueType newSub SidedValueType.
  SidedValueType newSub leftSided.
  SidedValueType newSub rightSided.

  rightSided newSub bothSided.
  bothSided addSuper leftSided.
```

It makes no difference in this case which parent is added with newSub and which with addSuper.

Again we emphasise: you are unlikely to produce models that work well and are easy to maintain if they contain more than a very few statements using addSub or addSuper.

4.6 Summary: Principles of building Re-usable Models in GRAIL

Here then are two key principles for producing re-usable models in GRAIL:

- Keep the asserted hierarchy of elementary categories simple and clean. Always make sure there is just one principle involved in the asserted hierarchy - e.g. whether vehicles are road or rail vehicles. Don't mix different principles in the same hierarchy;
- Make as much of the information as possible explicit by using necessary statements and definitions. Check the categories when you are finished to make sure that all of the information which applications may need is explicitly represented in the criteria.

A common problem in many representations which violates these principles, is that modellers confuse the 'part-of' and 'kind-of' relations. Often this is inevitable, because many systems do not provide adequate support for 'part-of' relations. This is one of GRAIL's important features which will be explained in the next chapter.

5. Refinement and Transitivity: What makes GRAIL different

GRAIL differs from most other description logics and frame systems in providing extra facilities to cope with transitive relations and the interaction of transitive relations and subsumption. These are amongst the most powerful features of GRAIL. They are what allows GRAIL to deal cleanly with part-whole and causal relations. However, they can also be confusing, particularly the operator for co-ordinating transitive relations and subsumption which is known as refinedAlong or specialisedBy. (The fact that there are two different variants of the syntax reflects the difficulty we have had explaining it to others.)

5.1 Transitivity

Transitive relations are relatively straightforward to understand. The basic pattern is that *"The part of the part is a kind of part of the whole"*. The pattern carries on indefinitely.

So if we say that seat covers are parts of seats, and seats are part of vehicles, then seat covers are also part of vehicles. Likewise if tread is part of tyres, and tyres are part of vehicles, then the tread is part of the vehicle.

(Philosophers and linguists worry about whether indefinitely long chains of "part-of" links should really be considered transitive, but for purposes of classification, we shall maintain here unequivocally that they should.)

To illustrate transitivity, let us extend the model a bit further as follows:

```
">>>>>>>>>>>>>>>>>>>>>>
> New categories
<<<<<<<<<<<<<<<<<<<<<<"
DomainCategory newSub VehiclePart.
   VehiclePart newSub Seat.
   VehiclePart newSub Wheel.
   VehiclePart newSub Nut.
   VehiclePart newSub SeatCover.
   VehiclePart newSub Tyre.
   VehiclePart newSub Tread.

DomainCategory newSub Damage.
   Damage newSub Puncture.
   Damage newSub Wear.
   Damage newSub Tear.


">>>>>>>>>>>>>>>>>>>>>>>>>>
>New attribute and its transitive declaration.
<<<<<<<<<<<<<<<<<<<<<<<<<<"
DomainAttribute newAttribute hasPart isPartOf allAll oneMany.
hasPart transitiveDown.


">>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>Sanctions
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<"
```

```
VehiclePart grammaticallyAndSensibly isPartOf Vehicle.
VehiclePart grammaticallyAndSensibly isPartOf VehiclePart.


">>>>>>>>>>>>>>>>>>>>>>
> Facts
<<<<<<<<<<<<<<<<<<<<<<"
Vehicle necessarily hasPart Seat.
    Seat necessarily hasPart SeatCover.
Vehicle necessarily hasPart Wheel.
    Wheel necessarily hasPart Tyre.
        Tyre necessarily hasPart Tread.

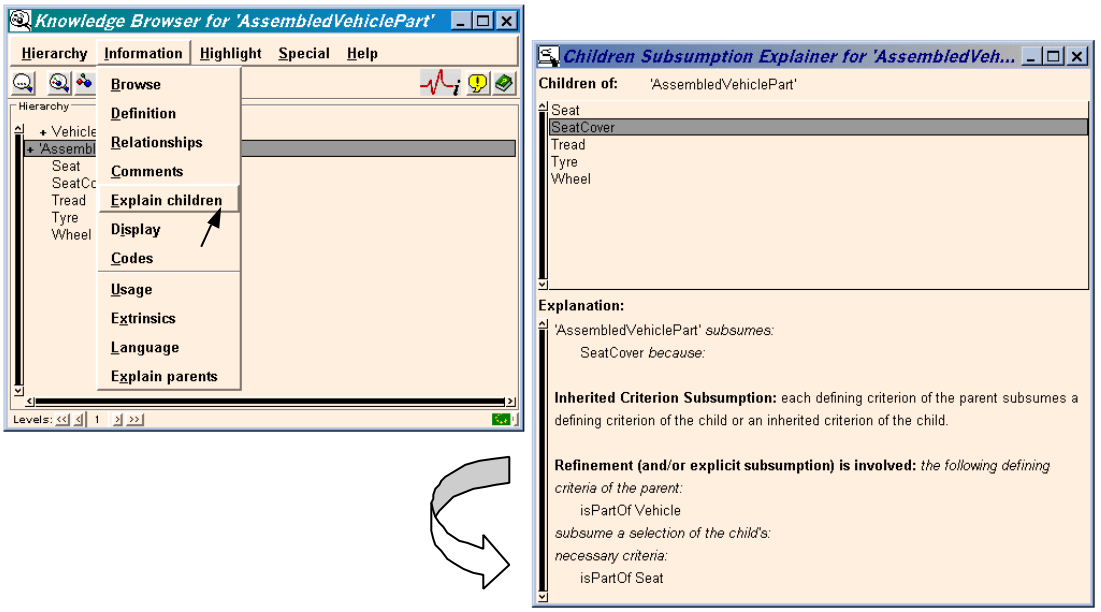
">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
> The definitions for demonstrating transitive relations
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<"
(VehiclePart which isPartOf Vehicle) name AssembledVehiclePart
```

Note the declaration:

```
hasPart transitiveDown
```

immediately following the newAttribute statement for hasPart. This declaration states that the attribute hasPart is transitive. (The downwards direction marker is now obsolete but can be useful to the compiler for optimisations.)

Evaluate everything and then browse AssembledVehiclePart. You should get an result as shown below. The list of subsumed categories includes Tread and SeatCover. Without the transitiveDown declaration neither Tread nor SeatCover would have been included. (Note that the order of the list of subsumed items is irrelevant - in fact it is alphabetical.)



The option **explain children**, as shown, offers a brief explanation of why each of these subsumptions exists.

5.2 The specialisedBy operator ('refinement')

The specialisedBy operator (and its converse form refinedAlong) are slightly more complicated.

The pattern is that *"Damage to a part is a kind of damage to the whole"*. For example, damage to a tyre's tread is a kind of damage to a vehicle. Conversely, a vehicle with a damaged part is itself damaged, *e.g.* a vehicle with a damaged tyre tread is a kind of damaged vehicle.

To illustrate the point, add the following to the model:

```
">>>>>>>>>>>>>>>>>>>>>>>
>New attribute and specialisedBy declaration
<<<<<<<<<<<<<<<<<<<<<<<"

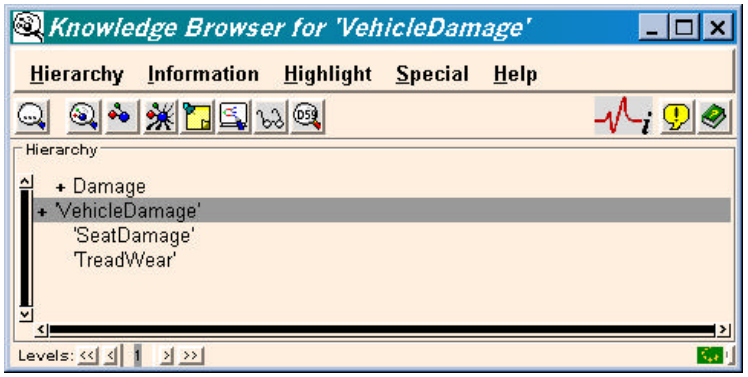
DomainAttribute newAttribute hasLocation isLocationOf allAll manyOne.
hasLocation specialisedBy isPartOf.

">>>>>>>>>>>>>>>>>>>>>>>
Sanctions for hasLocation
<<<<<<<<<<<<<<<<<<<<<<<"

Damage grammaticallyAndSensibly hasLocation VehiclePart.
Damage grammaticallyAndSensibly hasLocation Vehicle.

">>>>>>>>>>>>>>>>>>>>>>>>>>>>
> The definitions for demonstrating specialisedBy
<<<<<<<<<<<<<<<<<<<<<<<<<<<<"
(Wear which hasLocation Tread) name TreadWear.
(Damage which hasLocation Seat) name SeatDamage.
(Damage which hasLocation Vehicle) name VehicleDamage.
```

Evaluate the above and then browse VehicleDamage. You should find a pattern as shown in the screen dump below. Both SeatDamage and TreadWear are subsumed by VehicleDamage because of the specialisedBy declaration.



To illustrate the effect of the converse pattern, enter the following further definitions, evaluate them and browse DamagedVehicle.

```
(Tread which isLocationOf Wear) name WornTread.
(Seat which isLocationOf Damage) name DamagedSeat.

(Car which hasPart (Tyre which hasPart WornTread)) name CarWithWornTread.
(Bus which hasPart DamagedSeat) name BusWithDamagedSeat.
(Vehicle which isLocationOf Damage) name DamagedVehicle.
```

DamagedVehicle should now subsume both CarWithWornTread and BusWithDamagedSeat.

(Note also in passing the use of two levels of parentheses in the definition of CarWithWornTread. GRAIL syntax allows definitions to be nested to any level required, but each category must be enclosed in brackets as shown in the example.)

5.3 Transitive relations and cardinality

Strictly speaking, it makes no sense to talk about a transitive relation with a cardinality of manyOne or oneOne *i.e.* for the attribute to be both transitive and 'single valued'. Obviously, if the relation is transitive, then there can be many values. However, GRAIL interprets single valued for transitive relations to mean that there can be only one value for each step which is equivalent to saying that the relation must form a strict tree. An analogous meaning is used for a single-valued relation which is specialisedBy another. Strict trees are very common and very useful, so having a value type for them is helpful.

Finally, GRAIL assumes that transitive relations do not form cycles, *i.e.* that there are never loops in the transitive relation, for example, it can never be the case that something is a part of itself if the hasPart attribute is declared as transitive. There is no check for this in the system, but cycles in transitive relations may cause the system to loop endlessly.

5.4 Why are specialisedBy and transitiveDown important

It is very common in knowledge representation systems to confuse part-whole relations with the kind-of (or subsumption) relation. For any given application, it is usually possible to work around this limitation and to deliberately conflate part-whole and kind-of. However, as soon as you try to re-use the model, the work-arounds for one application get in the way of the next.

In later tutorials, many different applications of the transitiveDown and specialisedBy declarations will be described. The combination allows GRAIL models to be much simpler and more general than models in languages which lack these features.

5.5 How to remember how specialisedBy works

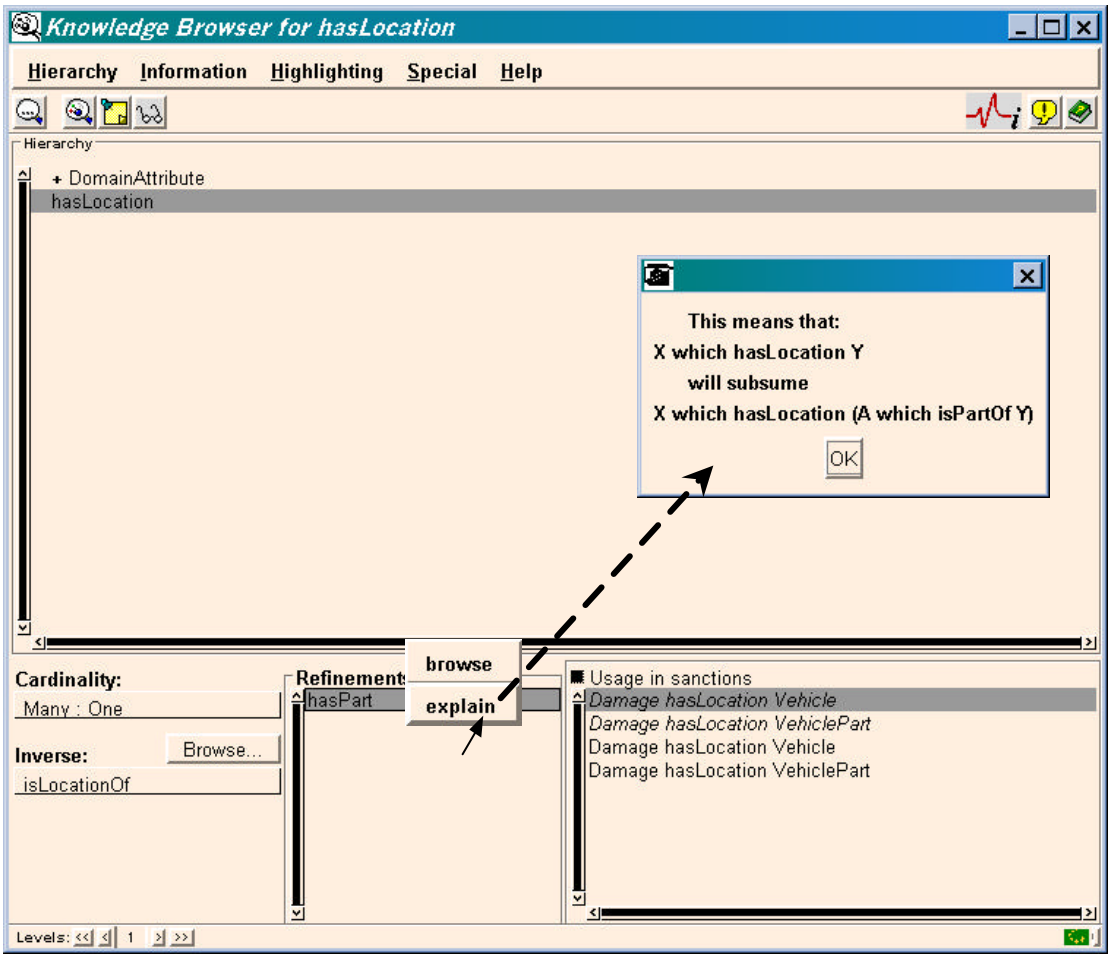
Everybody has trouble remembering the exactly how specialisedBy works when using it in a new situation. The best advice is to write down the basic pattern:

hasLocation specialisedBy isPartOf.

Therefore

Damage which hasLocation Thing
subsumes
Damage which hasLocation (Bit which isPartOf Thing)

Then match up the categories and attributes for your new problem precisely to this pattern. An additional prompt is available when browsing the refinements that have been applied to a particular attribute, as shown below:



6. Other Features

6.1 Extrinsic Information, 'Defaults', and Indexing

All information in definitions (which expressions) and necessary statements is inherited, without exception, by all concepts subsumed by the given concept. GRAIL does not support 'default reasoning' directly. If we make hasLocomotion flight part of the definition of birds, or if we add it to birds as a necessary statement, then hasLocomotion flight will be true of all birds in the model, including penguins, ostriches, and kiwis.

GRAIL does not support defaults directly. This is a deliberate choice. Default reasoning has been found to be extremely troublesome. One interpretation of the trouble is that there is just not enough information in a system like GRAIL to resolve the ambiguities which arise from defaults. GRAIL's approach is therefore not to try to resolve such ambiguities.

Instead, GRAIL makes it easy to hang 'extrinsic' information on the basic GRAIL model and use that information to identify the set of 'most specific' relevant information of a given sort. For example, we might define an attribute canFly with a value true or false and say:

Bird extrinsically canFly true.

Ostrich extrinsically canFly false

There is an operation which collects all the most specific extrinsic values, so in this case since Ostrich is more specific than Bird, it would collect just false. This gives many of the effects of default reasoning, but it cannot guarantee that there will be just one answer or that the set of answers will be consistent. It is up to the application or the user to decide how to chose the right answer if more than one is found, or indeed to accept all of the answers. The correct strategy depends on the application, and it often requires additional information beyond that held in the GRAIL hierarchy.

Extrinsic links do not affect classification. If we form the category Animal which canFly true we won't find Bird. This is a serious disadvantage, but it is the price to pay for a clean, simple system. We argue that ideas such as whether or not birds fly are not properly 'terminological', they are not a part of the naming of the thing, otherwise there would be, by definition, no exceptions.

However, we also argue, and have much experience to show, that having a clean explicit terminological structure minimises the clashes and ambiguities which otherwise occur with default reasoning in multiple hierarchies.

One way to think about this is that the GRAIL terminological structure provides a high quality index for looking up other information including information usually managed by default reasoning. Having a good index does not eliminate the need for reasoning, but it makes the reasoning easier and quicker.

6.2 Coding Schemes

GALEN includes services which allow annotation of concepts with codes from "external" coding schemes. The intention is to allow linking of the GALEN CORE with existing medical coding schemes. Volume J provides detailed descriptions of the services available to support this activity.

It is rather difficult to illustrate the code conversion capabilities of GALEN with the tutorial's transport example. However, the coding services can be seen simply as a more sophisticated version of the extrinsics mechanism described above.

6.3 Language

In the tutorial the only names given to expressions are their GRAIL names, otherwise known as 'knowledge names'. Every knowledge name must identify exactly one category or attribute. No synonyms are allowed, and the names must conform to GRAIL syntax - they must contain no spaces and begin with a letter.

One of the main functions of GALEN is to build terminologies with convenient, user friendly names. In fact, a major function is to build multilingual terminologies in which any term can have names in each language.

The GALEN terminology servers contain special modules for generating natural looking text from GRAIL expressions and for 'annotating' GRAIL categories and attributes to provide multilingual lexicons. This is beyond the scope of this tutorial.

7. Glossary

This section provides a brief recap of the GRAIL features we've introduced in this tutorial. It is not exhaustive - please let us know what's missing.

attribute

or 'relationship' - if you consider the model to be made up of nodes (concepts) and arcs relating them, attributes are the arcs…

category

…and categories are the nodes. The objects in GRAIL representing concepts - things - in the world. Sometimes 'concept'. Categories may be **elementary** (ie introduced using the newSub operator) or **complex** - made up by combining together categories with relationships using the which operator.

canonical form

the unique form of a complex GRAIL concept - ie with any redundancies or tautologies removed. See GALEN Deliverable 6 for a much more detailed treatment of this.

concept

see **category**

criterion

an attribute-value pair. Eg hasOwner Person. There are several varieties, indicating where they are used. **Defining criteria** are criteria which make up the definition of a complex category; for example, the defining criteria of

```
Person which hasSex male
```

is the single (in this case) criterion.

```
hasSex male
```

necessary criteria (sometimes 'essential', or 'conventional' criteria) are those properties which have been applied to a category using necessary statements. Properties are inherited along the subsumption network maintained by the GRAIL engine; **inherited criteria** are those necessary criteria that hold true of a category by inheritance along with any defining criteria of 'explicit' parents. The union of the defining, necessary and inherited criteria are known as the **complete criteria**.

GALEN

is the project itself - Generalised Architecture for Languages, Encyclopaedias and Nomenclatures in Medicine.

GRAIL

(or, more properly, 'the GRAIL Kernel') is the concept modelling language (or formalism) being developed in GALEN. It is the GALEN Representation and Integration Language.

model

the set of information presented as source files to the system - *or* - the result of the system (implementing the GRAIL formalism) having interpreted those sources. Also sometimes the set of implied categories that the 'model' sanctions. The word 'model' is over-used, and often for different things. Usually context helps.

naming

the process of attaching a useful 'nick-name' (sometimes 'knowledge name') to a complex concept.

necessary statement

a necessary statement applies an indefeasible criterion (or property) to a category. For example

(Person which isOwnerOf RoadVehicle) necessarily hasAge old

The necessary operator is bi-directional - this statement has said something about people that drive road vehicles, *and* the concept of 'old'. Two unidirectional variants exist - topicNecessarily and valueNecessarily; to say things only about the first concept, (as this example should be written):

(Person which isDriverOf RoadVehicle) topicNecessarily hasAge old

prototype

a complex GRAIL category - sometimes also known as a 'particularisation'.

refinement

the notion of making one attribute transitive across another.

sanctioning

Before we can combine categories and relationships to form complex categories, we need to say that we can. This mechanism, of telling the system which kinds of things can be combined (using particular attributes) to other kinds of things, is called *sanctioning*. We take the position that nothing is allowed until it is sanctioned by means of a 'sanctioning statement' (often just 'statement'; also previously known as a 'triple'). There are multiple levels of sanctioning: *grammatical* and *sensible*. Before you can create a complex category, or attach a necessary statement, it must be sanctioned to the sensible level. To sanction to a sensible level, it must be already sanctioned at the grammatical level. Sanctions are inherited.

SMK

Structured Meta-Knowledge - a precursor to GRAIL.

subsumption

the notion of 'is kind of'. If Man is 'a kind of' Person, then we say that Person subsumes Man, or that Man is subsumed by Person. Person is the subsumer, Man the subsumee. **Asserted (or explicit) subsumption** is the subsumption that comes about because of explicit facts said by the modeller (using the newSub or addSub operators); **formal subsumption** is the subsumption that comes about due to the rules of the GRAIL formalism; for example that Person subsumes Person which hasSex male.

Value type

Value types are concepts which have different subsumption rules applied to them. They typically appear only as values in criteria. Some kinds of value types are built into the system (eg IntegerValueType); additional types can be introduced as required (eg AgeValueType in this tutorial).