Abstract

Since 1985 the terminological representation system BACK (Berlin advanced computational knowledge representation system) has been developed at the Technical University Berlin. Its origin lies in the KL-ONE-based knowledge representation paradigm, semantic networks, and frame-based representation languages. The current system version has gone through several changes not only on the implementational level, but also on the conceptual level. For some time now the syntax has stabilized, and larger applications may be approached in the future.

Until now, introductions to the BACK system as well as its representation language were distributed throughout several publications, making it difficult for users to learn to handle the system. Hence, we found it worthwhile to write a tutorial guide through the BACK system. With BACK V5 the representation language has changed. On the one hand it became much more uniformer than in previous versions, on the other hand it was extended by some useful constructs which allow an easier customization of the language. Thus, we felt also the need for a user manual documenting the revised language.

The first part of this report is a tutorial introduction into the BACK system, which describes by example the modeling process we find most appropriate for terminological modeling with BACK. The second part is written in the form of a user manual, describing the modified and newly introduced language constructs.

BACK V5
Tutorial & Manual

Thomas Hoppe
Carsten Kindermann
J. Joachim Quantz
Albrecht Schmiedel
Martin Fischer

Technische Universität Berlin
Institut für Software und theoretische Informatik
Projekt KIT-BACK
Sekr. FR 5-12, Franklinstraße 28/29
W-1000 Berlin 10, Germany

March 1993

Contents

1 Introduction **1**
- 1.1 Guideline . . . 2
  - 1.1.1 How to Read This Document . . . 2
- 1.2 Quick Installation Guide . . . 3
- 1.3 Disclaimer . . . 3

2 BACK Tutorial **4**
- 2.1 A Tutorial Example . . . 6
- 2.2 Modeling a Terminology . . . 8
  - 2.2.1 Built-in Concepts . . . 8
  - 2.2.2 Primitive Concepts . . . 9
  - 2.2.3 Primitive Roles . . . 10
  - 2.2.4 Defined Concepts . . . 11
  - 2.2.5 Revision of Concepts . . . 11
  - 2.2.6 Disjoint Concepts . . . 12
  - 2.2.7 Defined Roles . . . 12
  - 2.2.8 Closed Attribute Domains . . . 13
  - 2.2.9 Open Attribute Domains . . . 13
  - 2.2.10 Attribute Sets . . . 14
  - 2.2.11 Number Ranges . . . 14
  - 2.2.12 Extended Role Definitions . . . 15
  - 2.2.13 Revising Roles . . . 15
  - 2.2.14 Value Restrictions . . . 15
  - 2.2.15 Number Restrictions . . . 16
- 2.3 Non-Definitional Information . . . 18
- 2.4 Representing a World . . . 18
  - 2.4.1 Creating Named Objects . . . 19
  - 2.4.2 Retracting Partial Descriptions . . . 20
  - 2.4.3 Revising and Retracting Objects . . . 20
  - 2.4.4 Creating Unnamed Objects and Filling Roles . . . 21
  - 2.4.5 Indirectly Referencing Objects . . . 22
  - 2.4.6 Asserting Unnamed Objects in Nested Descriptions . . . 22
  - 2.4.7 Closing Roles . . . 22
  - 2.4.8 Filling a Role with a Set of Objects . . . 23
  - 2.4.9 Defining Concepts by a Set of Objects . . . 24
- 2.5 Querying the System . . . 24

2.5.1 Retrieving Entities . . . 24
2.5.2 Describing Entities . . . 25
2.5.3 Applying Output Functions to Multiple Entities . . . 26
2.5.4 Disambiguating Entities . . . 27
2.5.5 Full Description of Entities . . . 28
2.5.6 Retrieving User-Given Descriptions . . . 30
2.5.7 Retrieving Combined Information . . . 31
2.5.8 Testing Subsumption . . . 32
2.5.9 Testing Equivalence . . . 32
2.5.10 Testing Incoherence and Disjointness . . . 33
2.5.11 Testing Concept Membership . . . 33
2.5.12 Retrieving the Difference between Entities . . . 33
2.5.13 Language Constructs Restricted to Queries . . . 34

3 Back Manual **35**
:</2 . . . 36
:=/2 . . . 37
=>/2 . . . 38
::/2 . . . 39
?</2 . . . 40
?:/2 . . . 41
***=/2** . . . 42
:/2 . . . 43
[. . .] . . . 44
../2 . . . 44
Attribute Set Term . . . 45
Number Term . . . 46
all . . . 47
allknown . . . 48
Concept Term . . . 49
Role Term . . . 50
Filler Expression . . . 51
anything . . . 52
aset . . . 53
atleast . . . 53
Concept Term . . . 54
Macro . . . 55
atmost . . . 55
Concept Term . . . 56
Macro . . . 57
attribute_domain . . . 58
backask . . . 59
backdump . . . 60
backinit . . . 61
backload . . . 62
backmacro . . . 63
backread . . . 64

backretrieve . . . 65
backstate . . . 67
backtell . . . 69
backwrite . . . 70
close . . . 71
comp . . . 72
defined_as . . . 73
describe . . . 74
describe_fully . . . 75
difference . . . 76
disjoint . . . 77
domain . . . 78
equivalent . . . 79
exactly . . . 80
for . . . 81
forget . . . 82
ge, gt . . . 83
getall . . . 84
incoherent . . . 85
intersection . . . 85
Attribute Set Term . . . 86
Number Term . . . 87
introduced_as . . . 88
inv . . . 89
le, lt . . . 90
msc . . . 91
name . . . 92
not . . . 92
no . . . 93
not . . . 93
Concept Term . . . 94
Role Term . . . 95
nothing . . . 96
nr . . . 97
number . . . 98
oneof . . . 99
or . . . 99
Concept Term . . . 100
Filler Expression . . . 101
range . . . 102
redescribe . . . 103
rf . . . 104
rvm_some . . . 105
rvm_no . . . 106
self . . . 107
some . . . 108
someknown . . . 109

string . . . 110
subsumes . . . 111
the . . . 112
theknown . . . 113
trans . . . 114
type . . . 115
uc(*i*) . . . 116
union . . . 117
vr . . . 118
without . . . 119

A Installation of BACK **122**

B Syntax Overview **124**

C Formal Semantics Overview **130**

D Programming Interface **133**

Chapter 1

Introduction

Since the end of the 1970s work on knowledge representation has addressed the development of representation languages with well-founded semantics, nowadays called 'Description Logics'. Description Logics (DL), which where previously called terminological logics (TL), term subsumption, etc., can be seen as a formal elaboration of the ideas underlying *Semantic Networks* (e.g., [Quillian, 1968]) and *Frames* (e.g., [Minsky, 1975]). Both representation formalisms share the idea of a hierarchically organized knowledge structure in which information is inherited from general concepts or frames to more specific ones. They also provide means for an internal structuring of concepts or frames which leads to horizontal connections: frames contain slots whose fillers are known to be instances of other frames; concepts contain properties that are modeled by links leading to other concepts.

Though both representation formats are very similar, there are also important differences: semantic networks adequately convey the general *structure* of the represented information, and especially interconnections and dependencies; a frame representation, on the other hand, focuses not so much on the overall structure but on the basic units and the information locally associated with each frame.

Both semantic networks and frames are ancestors of description logics and all three approaches to knowledge representation have much in common. There are, however, essential characteristics of DLs which distinguish them from their ancestors. The basic difference concerns the attitude towards theoretical foundations and towards the question of what is constitutive for a representation formalism. According to DL philosophy, a representation formalism should have a formal syntax, a formal semantics, a proof theory, and efficient inference algorithms.

In the second half of the 1970's representation languages from the area of semantic networks, frames, or scripts were seriously criticized in a number of papers for their apparent lack of formal rigor (e.g., [Woods, 1975] and [Hayes, 1977]). The key issue was the relationship between knowledge representation and formal logic. Brachman endorsed the logic-oriented view on knowledge representation in his early papers on semantic networks. In [Brachman, 1977] and [Brachman, 1979] he examined in detail, what the constructs used in semantic networks were supposed to represent. As a result he presented a collection of so-called *epistemological primitives*, which were supposed to be application-independent and became the basic language constructs of KL-ONE.

An overview over the basic features of the KL-ONE formalism circulated in the

beginning of the 1980's and was finally published in [Brachman and Schmolze, 1985]. In the following years, several terminological representation systems (TRS) have been developed incorporating different dialects but similar with respect to the underlying representation philosophy [Rich, 1991]. In addition to these practice-oriented implementations, thorough theoretical investigations yielded numerous results concerning decidability, tractability, and proof theory (see e.g., [Donini *et al.*, 1991a], [Donini *et al.*, 1991b], and [Royer and Quantz, 1992]).

In the course of this development an initial prototype of a terminological representation system, known as the BACK (Berlin Advanced Computational Knowledge representation) system, was implemented in the mid-80's at the Technical University Berlin. Over the years, the BACK system has evolved over several different implementations to a more efficient and effective representation system. Experience gained in this development process not only has led to major improvements in its architecture, but has led also to extensions of its inferential services and to a more uniform representation language.

Since users intending to approach larger applications with BACK have been missing a tutorial introduction to the BACK system, we decided to give the first part of this report the form of a tutorial introduction, offering users an easy access to the exploration of BACK's capabilities. Further, the representation language of BACK was revised with BACK V5. It not only has become much more uniform than in previous releases; also additional constructs were integrated (such as, defined roles, objects in definitions etc.), yielding an extended expressivity. Consequently, we gave the second part of this report the form of a manual, describing in more detail the modified and newly introduced language constructs.

1.1 Guideline

Chapter 2 of this report consists of a tutorial introduction to terminological modeling. We start with a brief overview over the terms used in terminological representation systems, introduce an example domain, which we use throughout Chapter 2, and discuss the language constructs in the course of modeling the example domain. The last part of Chapter 2 is divided into four sections. In these sections, we explain by example how the representation language can be used to model a terminology, how objects of a domain can be represented, how represented information can be retrieved, and what effect non-terminological inferences have.

The manual part in Chapter 3 is devoted to the current representation language of BACK V5. Here we describe for every language construct its syntax and semantics, explain the language construct by example, point out its idiosyncrasies and describe its differences in comparison to BACK V4. Some appendices in the end summarize in a compact manner the installation of BACK under Quintus Prolog, describe the syntax of the representation language and its semantics more formally.

1.1.1 How to Read This Document

This document is definitely not intended to serve as an introduction to description logics; readers without prior knowledge about description logics should first consult

some introductory literature (like e.g., [Brachman and Schmolze, 1985], [Nebel, 1990] or [Nebel and Peltason, 1990]) to get a feeling for such notions as “concept”, “role”, “classification” or “terminology”. Having acquired this basic knowledge, readers may either explore terminological modeling through the tutorial in Chapter 2, or “learn by doing”. In the latter case you should get a copy of BACK V5, install it as described in section 1.2 resp. the appendix, and try out the examples of Chapter 2.

Readers with prior knowledge about description logics can explore BACK’s capabilities through the tutorial examples in Chapter 2 or through the manual in Chapter 3, which describes in more detail the language constructs of version 5. A more compact description of BACK’s syntax and semantics can be found in the appendices.

Readers with prior knowledge, who wish to use BACK V5 in an application, should read first the appendix describing the installation of BACK. Chapter 2 may then be used as some kind of informal reference manual to the language of BACK. Detailed information about the BACK V5 representation language as well as its semantics can then be looked up in Chapter 3 and the appendices.

1.2 Quick Installation Guide

A quick installation of the BACK V5 system may be realized through the following sequence. A more detailed description of the installation procedure can be found in the appendix.

- Ensure that Quintus Prolog is installed on your local site.
- Uncompress the file ‘BACK.tar.Z’, and untar afterwards the file ‘BACK.tar’ with the command ‘tar -xf BACK.tar’. You should do this in a separate directory.
- The file ‘Readme.Back51’ contains further informations how to install BACK V5 for Quintus.

1.3 Disclaimer

BACK is still strictly an experimental prototype. It’s main purpose is to demonstrate the functionality of a fairly comprehensive implementation of description logics. There are bound to be situations where it will be too inefficient for practical applications. Also, there are still bugs, and in certain cases there will be a mismatch between the behaviour of the system and what is described in this manual. In such cases refer to the documentation that comes with the distribution, or send email to back@cs.tu-berlin.de.

Chapter 2

BACK Tutorial

In description logics (DL) one typically distinguishes between *terms* and *objects* as basic entities from which three kinds of formulae can be formed: *definitions*, *descriptions*, and *rules*. A definition has the form $\mathsf{t}_n := \mathsf{t}$ and expresses that the name t_n is used as an abbreviation for the term t. A list of such definitions is often called *terminology*. All DLs provide two types of terms, namely *concepts* (unary predicates) and *roles* (binary predicates), but they differ with respect to the term-forming operators they support. Common concept-forming operators are: conjunction (c_1 **and** c_2), disjunction (c_1 **or** c_2), and negation (**not**(c)), as well as quantified restrictions such as value restrictions (**all**(r,c)), which stipulate that all fillers for a role r must be of type c, or number restrictions (**atleast**(n,r,c) or **atmost**(n,r,c), stipulating that there are at least or at most n role-fillers of type c for r. Role-forming operators, besides conjunction, disjunction, and negation, are role composition (r_1 **comp** r_2), transitive closure (**trans**(r)), inverse roles (**inv**(r)) and domain or range restrictions (**domain**(c) or **range**(c)). In a description, an object is described as being an instance of a concept (o :: c), or as being related to another object by a role (o_1 :: r:o_2). Rules have the form $\mathsf{c}_1 => \mathsf{c}_2$ and stipulate that each instance of the concept c_1 is also an instance of the concept c_2. These basic notions can be summarized briefly by the following definitions, which are useful for understanding the example we present in the following.

Definition 1 (Concept) *A* concept *represents a set of instances, either intensional or extensional. An intensional definition of a concept specifies the characteristic properties of its instances. An extensional definition consists of an enumeration of all instances.*

Definition 2 (Role) *A* role *represents a binary relation between concept instances.*

Definition 3 (Term) *A* term *is a concept or a role.*

Definition 4 (Object) *An* object *is an instance of a concept. Objects may be instances of different concepts at the same time.*

Definition 5 (defined) *A term is called* defined, *if its definition describes necessary and sufficient conditions for the recognition of objects resp. relations.*

Definition 6 (primitive) *A term is called* primitive, *if its definition describes necessary conditions for the recognition of objects resp. relations.*

Before we introduce and model our example domain, we briefly sketch the services of BACK. Consider the simple domain model in Figure 2.1, it contains five concepts definitions. Four concept definitions are primitive (those introduced with the :< operator), which means that the specified conditions are necessary but not sufficient, and one is defined (introduced with :=), which means that the specified conditions are necessary and sufficient. Furthermore, there is one rule (introduced with =>) and four object descriptions (introduced with ::).

product	:<	**anything**
chemical product	:<	product
biological product	:<	product **and not**(chemical product)
plant	:<	**atleast**(1,produces,product)
chemical plant	:=	plant **and all**(produces,chemical product)
some(produces,chemical product)	=>	high risk plant
toxipharm	::	chemical product
biograin	::	biological product
chemoplant	::	chemical plant
toxiplant	::	**atmost**(1,produces) **and** produces:toxipharm

Figure 2.1: A Simple Model

Such a model is regarded as a set of formulae. Given the formal semantics of a DL, such a set of formulae will entail other formulae. Now the service provided by terminological representation systems is basically to answer queries whether some formula is entailed by a modeling. The following types of queries can be answered:

- t_1 ?< t_2
  Is term t_1 more specific than t_2, i.e., is t_1 *subsumed* by t_2? High risk plant subsumes chemical plant, i.e., every chemical plant is a high risk plant.

- t_1 **and** t_2 ?< **nothing**
  Are two terms t_1 and t_2 disjoint? The concepts chemical product and biological product are disjoint, i.e., no object can be both.

- o ?: c
  Is an object o an instance of concept c (object classification)? Toxiplant is recognized as a chemical plant.

- o_1 ?: r:o_2
  Are two objects o_1,o_2 related by a role r, i.e., is o_2 a role-filler for r at o_1? Toxipharm is a role-filler for the role produces at toxiplant.

- **getall**(c)
  Which objects are instances of a concept c (retrieval)? Chemoplant and toxiplant are retrieved as instances of the concept high risk plant.

- o_1 :: r:o_2 rejected
  Is a description o_1 :: r:o_2 inconsistent (consistency check)? The description chemoplant :: produces : biograin is inconsistent, i.e., biograin cannot be produced by chemoplant.

2.1 A Tutorial Example

The example we use throughout this chapter will be risk assessment of industrial plants, or more precisely the classification of plants into risk classes, depending on the security of the production process and the "harmfulness" of the waste produced by the plant. Plants, products, waste, and risks are first class objects in the example domain. Let our description start with the types of plants we like to model.

Industrial plants can be distinguished on different scales: the type of product produced by the plant, the structure of its products, the type of machines used, etc. Primarily, we distinguish plants in the following on the basis of the type of thing a plant produces. These types of things may be distinguished into mechanical, biological, or chemical products, or energy, which can be informally characterized as:

mechanical products are products produced in a mechanical fashion, e.g., screws, paper, desks, computers, cars.

biological products are produced by biological means or by modification of biological products, e.g., sugar cubes, instant soup, beer, lemonade, insulin, bacteria.

chemical products are produced by chemical reactions or by modification of chemical substances, e.g., PVC, cleaning substances, gasoline, painting colors.

A secondary distinction we use concerns the structure of the products produced in the plant, whether they consist of a single material (e.g., screws, sugar cubes and PVC), whether they are compound products made-up from several materials (e.g., books, cigarettes, instant soup, painting color), or whether they are assembled products consisting of single and compound products (e.g., bikes, computers, cars, plants).

Unfortunately, plants can break down. That means the production process does not work properly anymore. Even worse, such a breakdown can pollute the environment. Our goal is the classification of plants according to the risk they represent for their environment. Hence, we are primarily interested in plants which do not work properly and which may interact in an unpredicted way with their environment (e.g., a plant with a broken pipeline spilling oil, or a nuclear power plant polluting its environment). We adopt a simplified model where the status of a plant is distinguished into the following classes:

unmonitored which means that the plant is working fine and is monitored by automatic devices.

monitored which means that the production process has broken down, but no interaction with the environment has occurred yet. The plant is monitored by humans.

alert which means the production process has broken down and an unpredicted interaction of the plant with its environment has taken place.

Each of these classes has an associated risk. Unmonitored plants have the risk to break down and pollute the environment with waste which is produced during the normal production process. Monitored plants have additionally the risk of interacting with their environment in an unpredictable way. Plants in the status alert have the further

risk of polluting their environment with substances which are encapsulated during the normal production process.

A plant not only produces useful products, it also produces waste. Products as well as waste may occur with different degrees of "harmfulness": as non-toxic pollution, as toxic or radioactive contamination, as mechanical pollution in the form of noise and heat. Products and waste can be distinguished on at least three scales according to their direct or indirect polluting influence, according to the duration of a polluting influence, and according to the "victim" of a pollution.

Direct influence here means that a pollution affects somebody or something without intermediate states (e.g., direct contamination with a radioactive material, or direct raise of the temperature of a river through cooling water). On the other side, indirect influence means that the pollution goes through intermediate states (e.g., indirect accumulation of radioactivity via food, or indirect rise of temperature through the green-house-effect).

Dependent on the required degree of granularity the duration of an influence may be measured on different scales. We use here a very course-grained scale and distinguish between short-, medium-, and long-term influences. An explosion or contact with poisoning gas are examples of short-term influences, destruction of the ozone layer and intoxication of a river are medium-term influences, while radioactive pollution of the environment and the green-house-effect are considered as examples for long-term influences.

The "victim" of a pollution can be distinguished into living things (e.g., animals, humans, vegetation) and the non-living environment (e.g., water, earth, atmosphere).

The "harmfulness" of the waste depends on these scales and may be determined by an appropriate function. For example, a pollution of a small region of the environment with a high-toxic chemical substance with a small half-life period will not be as risky as a long term pollution of the atmosphere with a non-toxic material destroying the ozone layer. However, we just use a simple model here and do not worry much about the appropriateness of functions combining these scales of harmfulness into a single measure of the waste's risk.

These different risks, the risk of the waste produced during the normal production process, the risk for a plant not to work properly, and the risk to pollute its environment may be combined into a single overall risk value for the plant. We assume for this purpose a simple function, which combines the different risks into five risk values: high, large, medium, small, and null. This risk value is used for the classification of plants into high risk plants (e.g., nuclear-power plants, molecular biological plants), medium risk plants (e.g., automobile factories, chemical plants), and plants with small or null risk (e.g., screw factories, furniture plants, or food production).

2.2 Modeling a Terminology

The purpose of a terminology or taxonomy is the organization of concepts and roles of a domain dependent on their degree of specificity. Intuitively, a power plant is a plant and thus the term 'power plant' should be more specific than the term 'plant'. But, how can this be realized? Just looking at the strings representing the concepts will surely not suffice. Of course, we could explicitly enumerate all the objects belonging to these concepts and compare set-theoretical their extensions, but clearly this would be a tough task. Not only can these extensional sets be infinite, but also can we not assume that all objects are known in advance. Thus, explicit enumeration is only feasible for concepts representing a small, finite set of objects known in advance.

The usual way adopted in description logics is the intensional definition of concepts. This can be achieved by describing concepts and roles in relation to other concepts resp. roles. The automatic organization of concepts and roles into a terminology can then be realized for terminological languages by a subsumption algorithm comparing term structures. This algorithm is usually called *classifier*.

So let us see how the term 'power plant' can be defined intensionally. A power plant is a plant and it is producing power. Thus, we can relate the concepts 'plant' and 'power' by a role 'produces' to define the more specific concept 'power plant'. Clearly, we can now quite easily infer by term subsumption that every 'power plant' is also a 'plant', because the term 'power plant' is according to the subsumption relation more specific than the term 'plant'.

What we need of course is an appropriate term description language for defining concepts and roles, and a way of determining subsumption relations between concepts and roles. Such a term description language, i.e., the BACK representation language, is described by examples in this section. Chapter 3 and the appendix describe the language constructs in greater depth. The subsumption algorithm implemented in BACK V5 is similar to the algorithm used in the previous version, its formal semantics is given in the appendix.

For the purpose of this tutorial, we use the typographic convention that keywords of the BACK language (terminals of the BNF syntax) are written in **boldface**. Terms which were introduced in BACK as entities (e.g. concept, role, aset . . .) are written in sans serif. We follow the most practical (and perhaps most natural) way of modeling: we define first the terms we like to use in the example domain. Section 2.3 describes how non-terminological inferences can be incorporated into a domain model. In Section 2.4 we describe how assertional information can be expressed. Section 2.5 shows how terminological and assertional information can be retrieved, and how tests can be performed.

2.2.1 Built-in Concepts

At least two terms are of special interest: a term for denoting all objects of a domain, called **anything**, and a term for denoting the empty or incoherent concept, called **nothing**. These terms are built-in concepts of BACK. **Anything** denotes the most general concept which subsumes all other concepts; **nothing** denotes the most specific concept which is subsumed by every concept.

2.2.2 Primitive Concepts

Primitive concepts are used to represent the natural kinds of a domain which cannot be or should not be further defined. They are taken as they are, which means that the classifier will not change their position within the taxonomy. The right-hand side of a primitive concept declaration represents necessary conditions for the classification process.

Example 1: The terms 'plant' and 'product' represent natural kinds of our example domain, hence we define them as primitive concepts. And because these are the most general terms in our domain they are directly subsumed by the concept **anything**.

plant $:<$ anything.
product $:<$ anything.

Clearly, if we extend the representation we could state necessary and sufficient conditions for determining whether an object is a plant resp. a product. These concepts could then be introduced equally well as defined concepts[1]. But note that since we cannot completely define all the concepts of a domain, some defined concepts will always be based on primitive concepts[2].

Of course, there are other necessary conditions plants as well as products must fulfill. A 'plant has a location', it 'is owned by somebody', it 'produces products', 'consumes material and energy', a 'product has a weight and a price', 'consists of parts' etc. Some of these properties will be introduced as roles below.[3]

For the purpose of our example domain these concepts are still quite general because we like to distinguish between mechanical, biological and chemical plants, waste and products. Thus, we refine these primitive concepts in the following way:

Example 2: Mechanical, biological, or chemical plants are plants. Products can be distinguished into mechanical products, chemical products, biological products, or energy. Material and waste can be considered as a kind of product. Waste may be toxic or non-toxic. Radioactive material is a material, and nuclear waste is a form of toxic waste.

mechanical_plant $:<$ plant.
biological_plant $:<$ plant.
chemical_plant $:<$ plant.
mechanical_product $:<$ product.
biological_product $:<$ product.
chemical_product $:<$ product.

[1] Defined concepts will be introduced further below, after we have build-up some initial terminology.

[2] The decision which concepts should be introduced as primitives is beyond the scope of terminological modeling, since this decision about concept granularity depends on the domain and the intended application.

[3] Just for the purpose of this tutorial, we start with most general descriptions, and extend concepts and roles step by step. We note that this modeling process is not practical as long as the domain was not analyzed in advance. Instead, a thorough analysis of the domain should precede any modeling activity.

energy :< product.
material :< product.
waste :< product.
radioactive_material :< material.
toxic_waste :< waste.
non_toxic_waste :< waste.
nuclear_waste :< toxic_waste.

Although we can model a whole taxonomy just with primitive concepts, it is not wise to do so. Since primitive concepts do not express sufficient conditions, it is not possible to classify them automatically. Thus, the user has the complete responsibility to organize the primitive concepts of a taxonomy. However, as we will see after we have introduced roles, some of these concepts can be introduced as defined concepts.

2.2.3 Primitive Roles

Roles are used to represent binary relations. In analogy to primitive concepts, primitive roles are used to define basic relations and represent – like primitive concepts – necessary conditions for the classification process.

Example 3: Plants produce products and need energy. They also co_produce waste. Products may directly contain other products.

produces :< domain(plant) and range(product).
needs :< domain(plant) and range(energy).
co_produces :< domain(plant) and range(waste).
directly_contains :< domain(product) and range(product).

The keyword **and** is used to conjoin terms. Note that in BACK we denote with the term **range** what is usually called in object-oriented models "domain of an attribute". Although the directly_contains role states that a product may directly contain other products, it does not model yet the transitive closure of the role contains, which we like to represent and which will be introduced later. However, because we have introduced material and waste as a kind of product, the directly_contains role covers already cases where a product contains material or waste.

This is just one form of introducing roles, where the domain and the range of the role are explicitly mentioned, and where the role is introduced in advance. Another form of role introduction, called *forward introduction*, is performed automatically if a not yet introduced role is used.

Example 4: Let us reconsider the concept of a plant. A plant is located at a place and is of a certain type. Thus, let us revise the concept plant, and let us introduce the concepts place and type as primitive.

place :< anything.
type :< anything.
plant :< anything and all(located_at,place) and all(is_of_type,type).

These definitions have the following consequences: First, since the concept plant was already introduced before, it is revised now with this new definition. Second, the roles located_at and is_of_type, which were not introduced before, are introduced now (this is called 'forward introduction'). Third, the concept **anything** becomes the domain and range of the newly introduced roles. And fourth, the operator **all** restricts the introduced role locally at the concept to the concept mentioned in the second argument. This is called a *value restriction* and will be explained in more detail below.

2.2.4 Defined Concepts

Besides the definition of primitive concepts, concepts may also be defined. For defined concepts the right-hand side of the concept definition represents necessary and sufficient conditions for concept classification. The classifier will automatically determine the right place in the taxonomy where the concept has to be integrated, and will automatically recognize objects which are instances of a defined concept.

Example 5: Let us declare the concepts water_energy_plant and wind_energy_plant as primitive plants producing energy. A plant producing power from wind or water is a mechanical plant and at the same time it is a wind resp. water energy plant, this definition is sufficient to classify any wind_power_plant and water_power_plant. We may also like to use the defined concept of a mechanical_energy_plant for some reason, which can be defined as a mechanical plant producing energy.

```
wind_energy_plant       :< plant and all(produces,energy).
water_energy_plant      :< plant and all(produces,energy).
wind_power_plant        := mechanical_plant and wind_energy_plant.
water_power_plant       := mechanical_plant and water_energy_plant.
mechanical_energy_plant := mechanical_plant and all(produces,energy).
```

The classifier will determine the proper position in the taxonomy w.r.t. concept subsumption and will detect that the introduced concepts wind_power_plant and water_power_plant have to be subsumed by the concept mechanical_energy_plant, which is subsumed by the concept mechanical_plant. Additionally, the classifier will recognize objects subsumed by the above defined concepts.

Example 6: Defined concepts can also be used to introduce synonym names for concepts. For example, if we like to use the terms factory or workshop instead of plant, we can declare them as synonym as follows:

```
factory  := plant.
workshop := plant.
```

2.2.5 Revision of Concepts

Since we have roles available for relating concepts, we can revise – as promised above – some of the previously introduced primitive concepts and introduce them more precisely as defined concepts. Although the meaning of a defined concept is stronger than the meaning of a primitive concept, the former have the nice property that the classifier will determine their proper position in the taxonomy automatically, and as a consequence, can automatically recognize their object instances.

Example 7: A mechanical, biological, or chemical plant is a plant which produces mechanical, biological, resp. chemical products.

mechanical_plant := plant and all(produces,mechanical_product).
biological_plant := plant and all(produces,biological_product).
chemical_plant := plant and all(produces,chemical_product).

Again it should be noted that if we would declare mechanical, biological, and chemical as primitive concepts, we could equally well introduce the concepts mechanical_product, biological_product and chemical_product as defined concepts.

As should be clear from the previous revision of the concept plant and the above revisions, we can not only revise primitive concepts, but also a primitive concept can be revised into a defined one, and vice versa. The same holds for roles. An example of role revision will occur further below. Note however, that a concept cannot be revised into a role or vice versa.

2.2.6 Disjoint Concepts

In the beginning we introduced the concepts energy and material as primitive concepts. We have also seen that concepts can be conjoined. Thus, we could also formulate a concept which is both 'energy and material'. Although this will perfectly make sense for a physician, it does not make much sense in our risk assessment domain. Because of the intended level of granularity we will never encounter an object which is both energy and material at the same time. Thus, we should revise these concepts and should declare them more precisely as disjoint.

Example 8: Energy and material, as well as energy and waste are clearly distinguished concepts, for which no common instance will exist in the example domain.

energy :< product.
material :< product and not(energy).
waste :< product and not(energy).

The concept operator **not** can only be applied to primitive concepts. Note that these definitions do not declare the concepts material and waste as disjoint. Thus, objects which are both material and waste are still representable. Although we could equally well declare the concepts plant and product as disjoint, we would exclude that plants can be considered as product resp. that plants can be produced in a plant. Again we note that the decision which concepts need to be declared as disjoint is dependent on the domain we like to model as well as on the intended application.

2.2.7 Defined Roles

We can also construct defined roles from primitive ones. As for defined concepts, the right-hand side of a defined role declaration represents necessary and sufficient conditions.

Example 9: Plants produce products. If we need to represent the information which product is produced_by a certain plant, we can define this role as inverse of the role produces. If products should contain products we need to represent the transitive closure of directly_contains. A plant also uses up materials during the production process. A uses_up role depends on the products produced and the materials contained in these products, thus it can be defined in terms of a role composition.

```
produced_by := inv(produces).
contains    := trans(directly_contains).
uses_up     := produces comp contains and range(material).
```

As should be obvious, the **inv** operator constructs the inverse role, the **trans** operator the transitive closure, and **comp** constructs the role composition.

2.2.8 Closed Attribute Domains

Sometimes we want to describe concepts by unstructured atomic values. These values are called *attributes*[4] in BACK, and a set of such values forms an *attribute domain*. We distinguish mainly between two types of attribute domains: closed and open ones. Attributes of closed attribute domains must be given at declaration-time and are considered to be ordered.

Example 10: A risk in our domain may take on the values high, large, medium, small or null. The duration of a pollution may be long, medium or short, while a plant can be in the states: alert, monitored and unmonitored.

```
risk     := attribute_domain([high,large,medium,small,null]).
duration := attribute_domain([long,medium,short]).
state    := attribute_domain([monitored,alert,unmonitored]).
```

Closed attribute domains are implicitly ordered from left to right, so that subranges can be constructed out of them. If an attribute is encountered which was not previously declared to belong to a closed attribute domain, an error will be reported.

2.2.9 Open Attribute Domains

It may be the case that we cannot specify the attributes belonging to an attribute domain at declaration-time; only at run-time they may be available. In this case an open attribute domain has to be declared, which allows us to collect during run-time the occurring attribute values. Open attribute domains are regarded to be unordered.

Example 11: In the context of risk assessment, we may like to relate a plant to the final judgement about the risk it represents. These judgements may stem from different sources and may be described in different terms. For example, judgements may be described by attributes like safe, secure, reliable, riskless, harmless etc. However, at declaration time we do not know which attributes will be used.

[4]Note that the term *attribute* is used in other terminological representation systems to denote a role which is functional and total on its domain.

judgement := attribute_domain.

If it can be determined from the context that an attribute has to belong to an open attribute domain, the corresponding attribute domain is automatically extended by this attribute.

2.2.10 Attribute Sets

Sometimes we like to use a subset of attributes from an attribute domain or sets obtained from set theoretical operations on two attribute domains, e.g., **union**, **intersection** and subtraction (denoted **without** in BACK). Subsets of attribute domains can be chosen in BACK with the **aset** operator.

Example 12: The term risky may refer to the range of risks between high and medium, and the term unrisky to the risks small and null. Controlled_by_humans should consist of the subset of values of the attribute domain state which indicate that a plant is monitored by humans, whereas automatically_controlled should be used to represent the opposite case that the plant is monitored by automatic devices.

```
risky                  := aset(high..medium,risk).
unrisky                := aset(small..null,risk).
controlled_by_humans   := aset([alert,monitored],state).
automatically_controlled := aset([unmonitored],state).
```

As described above closed attribute domains are implicitly ordered from left to right. Note that the use of the range operator '..' is not appropriate and may result in strange subranges, if in an application domain attributes are considered to be unordered. Furthermore, the range operator applied to an open attribute domain, which is considered to be unordered, may result in strange subranges. Attribute sets can also be defined by the set theoretical operators mentioned above; however, in our example domain we have no interesting example for their use.

2.2.11 Number Ranges

Besides nominal attribute domains, it is also possible to use numbers and number intervals. Intervals can be constructed by the number operators **intersection** and '..' which are used to specify and to compute an interval. The operators **lt** (less than), **gt** (greater than), **le** (less equal), and **ge** (greater equal) can be used to specify a range of numbers below or above a certain threshold.

Example 13: The term harmfulness may be defined by a numerical range between 0 and 10. A harmfulness between 5 and 10 will be termed dangerous. If the harmfulness of the waste is less than 3 we call it safe.

```
harmfulness := 0..10.
dangerous   := harmfulness intersection gt(5).
safe        := lt(3).
```

Note that numbers are not restricted to integers, although the above examples seem to imply this. Simple numbers can always be used instead of a number range N..N; N is equivalent to 'N..N', so that no problems arise if the intersection of an interval and a number is constructed.

2.2.12 Extended Role Definitions

Let us return once more to role definitions. Above we have discussed primitive and defined roles whose ranges were restricted to concepts only. However, we also have the possibility to restrict the range of roles to attribute domains, numbers, and strings.

Example 14: Plants have a risk to break down and a risk to pollute the environment. Plants also operate in some mode, which can take on values of the attribute_domain state. Products may have a degree of harmfulness. Products usually have an identification code, which can be considered as an unstructured string. These roles can be expressed as follows:

breakdown :< domain(plant) and range(risk).
pollution :< domain(plant) and range(risk).
op_mode :< domain(plant) and range(state).
degree :< domain(product) and range(harmfulness).
id_code :< domain(product) and range(string).

Whereas the range may be a concept, attribute domain, aset, number, or **string**[5], the domain can only be a concept.

2.2.13 Revising Roles

Although these role definitions represent already the intended domains and ranges correctly, there is still some information missing. For example, a product usually has at most one identification code, thus the role id_code needs to be functional on its domain. And further, since we like to assign to a plant at most one breakdown risk and one pollution risk, and since a plant can be in at most one mode of operation, the corresponding roles should be functional as well. Thus we revise the above role definitions as follows:

breakdown :< domain(plant) and range(risk) type feature.
pollution :< domain(plant) and range(risk) type feature.
op_mode :< domain(plant) and range(state) type feature.
degree :< domain(product) and range(harmfulness) type feature.
id_code :< domain(product) and range(string) type feature.

The modifier **type feature** declares roles to be functional on their domain. The declaration of a role as a feature has the effect that all number restrictions for the role are unified with [0,1], which will be explained below.

2.2.14 Value Restrictions

As we have mentioned already above, the local range of a role attached to a concept can be restricted at declaration time of the concept. We refer to this as *value restriction*. The attached value restriction becomes a definitional part of the concept.

[5] Although the keyword **string** could be regarded as analogous to a predefined open attribute domain of strings, it actually is not an attribute domain. In other words, no subset of **string** can be formed.

Example 15: Let us introduce some additional concepts which restrict the range of roles locally. We can define the concept energy_plant as a plant which produces energy. The concept of a nuclear_power_plant can be defined as an energy_plant which uses up radioactive material and co_produces nuclear waste. We can also define more complex concepts like, e.g., a dangerous_plant, which is a plant producing some products with a dangerous degree of harmfulness, or a risky_chemical_plant, which can be defined as a plant which only produces chemical products and whose breakdown risk is risky.

```
energy_plant       := plant and all(produces,energy).
nuclear_power_plant := energy_plant and
                       all(uses_up,radioactive_material)
                       all(co_produces,nuclear_waste).
dangerous_plant    := plant and
                       some(produces,some(degree,dangerous)).
risky_chemical_plant := plant and
                       all(produces,chemical_product) and
                       all(breakdown,risky).
```

some is a predefined macro in BACK with the meaning 'atleast(1,r and range(c))', which expresses that there is at least one role-filler for r of type c. As the definition of dangerous_plant indicates, value restrictions do not have to be atomic but can consist of complex terms.

2.2.15 Number Restrictions

A concept may be related over roles to other concepts; this is usually expressed by value restrictions. However, a role may have only a certain number of allowed fillers. Such a constraint on the cardinality of potential objects filling a role is called *number restriction*. Number restrictions represent either a minimal or a maximal number of role-fillers, thus they are distinguished into *minimum* resp. *maximum restrictions*. An exact number of fillers may be modeled if the minimum restriction equals the maximum restriction.

Example 16: A plant producing exactly one product, called a mono_plant, can have exactly one filler belonging to the concept product in its produces role. A broken_plant does not produce anything, thus the number of fillers of its produces roles is zero. The concept of a toxic_waste_plant can be defined as a plant, which co_produces at least one type of toxic waste. A solid_product contains only objects of type material and contains exactly one object, while a compound_product contains also only objects of type material, but needs to contain at least two of those objects. An assembled_product contains at least one product, hence its contains role needs to be filled with at least one product. These concepts may be defined as:

```
mono_plant       := plant and
                    atleast(1,produces) and
                    atmost(1,produces).
broken_plant     := plant and atmost(0,produces).
toxic_waste_plant := plant and
```

```
                          atleast(1,co_produces,toxic_waste).
solid_product        := product and all(contains,material) and
                          exactly(1,contains).
compound_product     := product and all(contains,material) and
                          atleast(2,contains).
assembled_product    := product and atleast(1,contains).
```

If no number restrictions are specified for a role the minimum restriction is set to 0 and the maximum restriction is set to 'infinite'. This amounts in saying, that the role may have an arbitrary number of fillers. The expression **atleast** (1,co_produces,toxic_waste) uses the predefined macro **atleast**(n,r,c) of BACK which is expanded into the more complex expression 'atleast(n,r and range(c))'. **Exactly**(n,r) is also a predefined macro with the meaning 'atleast(n,r) and atmost(n,r)'.

Note that there exist small semantical differences between definitions which seem – on the first sight – to express the same statement. For example, while a term of the form 'atleast(n,r) and all(r,c)' states that all role-fillers for r are restricted to range c, and that there are at least n role-fillers for r, the expression 'atleast(n,r and range(c))' states that there are at least n role-fillers for r of type c. The former definition restricts all role-fillers for r, while the later restricts only some of them. Similar differences can be found for the macros **some**, **atmost** and **exactly**.

Since we now have nearly all constructs available for defining concepts we can once more revise some of the previously introduced concepts.

Example 17: A plant is located_at exactly one place and is also of exactly one type. An energy_plant produces at most two forms of energy: heat and electricity. A nuclear_power_plant uses_up – besides other materials – at least one radioactive_material and co_produces – besides other waste – at least nuclear_waste. And finally, a risky_chemical_plant produces at least one chemical_product.

```
plant                :< anything and
                         all(located_at,place) and
                         exactly(1,located_at) and
                         all(is_of_type,type) and
                         exactly(1,is_of_type).
energy_plant         :=  plant and
                         all(produces,energy) and
                         atmost(2,produces).
nuclear_power_plant  :=  energy_plant and
                         atleast(1,uses_up,radioactive_material) and
                         atleast(1,co_produces,nuclear_waste).
risky_chemical_plant :=  plant and
                         all(produces,chemical_product) and
                         atleast(1,produces) and
                         all(breakdown,risky).
```

2.3 Non-Definitional Information

Above we have considered one part of terminological modeling. However, sometimes concepts are not only related over their definition, but also by non-definitional information. Such information can sometimes be represented by implications between concepts (I-links, rules).

Example 18: If a plant co_produces toxic_waste, we want to infer that its pollution risk is risky. If we know that a plant is in mode controlled_by_humans, we can in accordance with the intended meaning of the state attributes monitored and alert conclude that it is necessarily a broken_plant. Or as another example, consider fast breeders: if we know that a certain nuclear power plant is of that type, we may like to conclude for some reason that it is a dangerous_plant.

```
some(co_produces,toxic_waste)                  => all(pollution,risky).
plant and all(op_mode,controlled_by_humans)    => broken_plant.
nuclear_power_plant and is_of_type : fast_breeder => dangerous_plant.
```

As we see from these examples, the left-hand side of the '=>' operator represents the premise of a rule[6], while the right-hand side represents the rule's conclusion. These rules have the effect that if an object is asserted[7] as an instance of the premise, the rule is executed, and the object description is extended by the rule's conclusion. In the first rule we have used the BACK macro **some**, which is predefined as 'atleast(1,r and range(c))'. The last two examples show that any well-formed concept expression can be used as a premise. The last rule uses a *filler expression* (is_of_type : fast_breeder) which expresses that the role is_of_type is filled with an the object fast_breeder. Filler expressions will be described further below.

Example 19: Let us assume as a further rule: if a dangerous_plant is broken we conclude that its pollution risk is high. Although this rule represents more a 'default', we assume that in the risk assessment domain it represents a hard fact, because of the intended application.

```
broken_plant and dangerous_plant => pollution : high.
```

In conjunction with the last two rules of the previous example an assertion of an object has the consequence that all rules are fired as long as they are applicable and until the object description does not change anymore.

2.4 Representing a World

In the sections above we have defined the meanings of terms we want to use in our example domain of risk assessment. These terms correspond to abstract entities, which are used to describe the terminological structure of the domain under consideration.

[6] The meaning of 'rule' used here does not correspond to the usual meaning of this term in AI. Although rules are used for 'forward-chaining' inferences, no conflict resolution has to be performed, since they are regarded as logical constraints.

[7] How objects are created as instances of concepts will be described in the next section.

Of course, terminology is not enough; we want to use it to describe concrete objects and relations.

Before we can do anything with objects and relations we have to put them into existence. Thus, we need a way for asserting objects and relations as instances of concepts and roles. The possible assertion and retraction operations of BACK will be described in this section, while the next section is devoted to queries which can be answered by BACK.

2.4.1 Creating Named Objects

We describe a situation of the real world usually with some named objects which are instances of some concepts. Since an object is an instance of a concept, we need a way of creating named concept instances.

Example 20: Three Miles Island is the well-known nuclear power plant near Harrisburg. For accessing the risk Three Miles Island represents for the environment we need to create an instance of the concept nuclear_power_plant.

'Three Miles Island' :: nuclear_power_plant and located_at : 'Harrisburg'.

Note that upper case names (i.e. 'Three Miles Island' and 'Harrisburg') need to be quoted in Prolog, because otherwise they are regarded as variables. Such an object description triggers the creation of a named object instance of the concept nuclear_power_plant, and fills its located_at role with the value 'Harrisburg'. Additional, information from the concept definition is inherited to the object: Three Miles Island uses_up radioactive_material, co_produces nuclear_waste and produces energy. Also new information is inferred by the classifier, e.g. that 'Harrisburg' must be some place, etc. Furthermore some of the rules are triggered which add new information to the object, e.g. that the pollution risk of 'Three Mile Island' is risky. The names used for objects must follow the 'unique names assumption', i.e., every name can only be used to name one object at a time, and objects having different names are supposed to be different objects.

Example 21: If we know that 'Three Miles Island' is a fast breeder, then we can extend the object description with this new information. Of course, we also need to update the description of the object Three Miles Island if an accident occurs in Three Miles Island. Let us assume that Three Miles Island is broken in the sense that it does not produce energy any longer, and that it co_produces some nuclear waste.

'Three Miles Island' :: is_of_type : fast_breeder.
'Three Miles Island' :: broken_plant.

While the first update only adds new information to Three Miles Island, the second update additionally now triggers another non-terminological rule. Thus, after the second update, the maximum restriction of its produces role is set to 0, which means that Three Miles Island produces nothing, and the pollution role will be filled with value high derived from the corresponding rule shown above.

2.4.2 Retracting Partial Descriptions

Further non-terminological rules may be triggered if new information about an object arrives. However, it may also happen that we have to retract previously given information in later modeling steps.

Example 22: Three Miles Island will be surely repaired one day or the other, and thus we have to update the object again. Let us assume that Three Miles Island is repaired, that it again produces energy, and that its production of nuclear waste again is "normal", Then we need to retract the information that it is a broken_plant.

```
backtell(forget('Three Miles Island' :: broken_plant)).
```

Note that only user-told facts about objects and rules can be retracted by the user. The system, however will not only retract this information,, but also information derived from Three Miles Island being a broken plant, i.e., the information that it produces nothing and that its pollution risk is high.

Although the **backtell** command is the usual command for asserting new information in BACK, we have not used it in the previous examples. This was possible since the operators ':<', ':=', '=>', '::', '?<' and '?:' are recognized as BACK tell- resp. ask-expressions. The only exceptional operator is '=', which needs always to be enclosed in a **backretrieve**, since it is predefined in Prolog. For deciding where to use a **backtell**, **backask**, or **backretrieve** command the following rule of thumb may be useful: *if a command consists of a Prolog predicate rather than an operator, it should be enclosed in a* **backtell**, **backask**, *or* **backretrieve**.

2.4.3 Revising and Retracting Objects

Clearly, if we can retract partial descriptions of objects we have the opportunity to revise them following the motto: revision = subtraction + addition. But since object revision and retraction via a sequence of single retraction operations is a time consuming process, the representation language of BACK includes abbreviating constructs.

Example 23: Suppose we have created an Erroneous Input object in two steps as an instance of the concepts toxic_waste and wind_power_plant. Once we have detected that this object makes no sense in our application we could revise it in two separate steps. However, it is computationally cheaper to do this in one operation.

```
'Erroneous Input' :: toxic_waste.
'Erroneous Input' :: wind_power_plant.
backtell(redescribe('Erroneous Input' :: mechanical_product and
                                         wind_power_plant)).
```

A redescription is computationally cheaper since an update of the ABox needs to be performed only once. This will pay off if several other objects must be updated which are connected over roles to the revised object.

Example 24: Suppose we notice later that we do not need the object Erroneous Input at all. We can retract it completely with a **forget** operation.

backtell(forget('Erroneous Input')).

A complete retraction is computationally cheaper than retracting every partial description in isolation.

2.4.4 Creating Unnamed Objects and Filling Roles

Sometimes we need to represent an object, but have no name for it. Either because we actually do not know its name, or because temporarily we do not know its name. Such situations occur frequently in the context of 'hypothetical reasoning', if we assume that there exists an object with such and such properties, but we cannot identify which object it is.

Example 25: Let us assume we receive the incomplete information that 'an environmental pollution happened with the toxical chemical waste 'Dioxin' in a chemical plant'. Although we do not know yet the name of the plant, we need to represent it for determining preliminary information about the risk of this event.

'Dioxin' :: toxic_waste and chemical_product
X :: chemical_plant and co_produces : 'Dioxin'.

This description creates an object as instance of chemical_plant and fills its co_produces role with the filler Dioxin. The object is in turn recognized by the classifier as a toxic_waste_plant, because Dioxin was previously introduced as a form of toxic_waste. The X represents a Prolog variable which is bound after the creation of the object to a unique constant $uc(i)$ generated by BACK, which can be used to refer later to this object.

The unique identifier generated by the above object description can be used either to extend the object description or to name the object later on.

Example 26: Let us assume that in the last example $uc(261)$ was generated and that we receive the information that the chemical accident happened in plant 17 of ChemoPharm, then we can easily update the object description, so that we can later refer to this object with the name ChemoPharm Plant 17.

backtell(name($uc(261)$,'ChemoPharm Plant 17')).

It should be obvious that the number associated with an unique constant depends on the order in which unnamed objects were processed. Thus, if the formalization preceding this example is slightly modified, e.g. an additional unnamed object is introduced or the order in which unnamed objects were created has changed, we cannot expect that the above uc denotes the right object.

Although this example in isolation seems to suggest that the user could also introduce uc's as names for other uc's, this is not possible in BACK. BACK will check whether the second argument of the **name** command corresponds to an identifier starting with uc. In this case the **name** command will fail, since uc's are generated only by BACK.

2.4.5 Indirectly Referencing Objects

If we neither know the name of an object nor a unique constant denoting it, we can use – in some situations – as a further possibility for referring to objects *indirect references*, where an indirect reference is a description which identifies an object uniquely.

Example 27: Let us return to the broken plant near Harrisburg. We can imagine that several nuclear power plants are represented in our knowledge base under which Three Miles Island is at the moment the only one which is broken. Thus, to represent the information that the broken nuclear power plant uses_up uranium, we can assert:

```
theknown(nuclear_power_plant and broken_plant) ::
        uses_up : ('Uranium' :: radioactive_material).
```

Note that indirect referencing with the construct **theknown** requires that the object is uniquely determinable. If none or several objects are determinable under indirect referencing, **theknown** will fail.

2.4.6 Asserting Unnamed Objects in Nested Descriptions

As for roles during the modeling of the terminology we can also nest descriptions of role-fillers. This is especially useful if we have to fill a role at the end of a role chain and don't know yet any of the intermediate objects in the chain.

Example 28: We may have to represent the information that a mechanical plant produces something which contains a subcomponent, which itself contains chlorofluorocarbon (CFC).

```
'CFC' :: material and chemical_product.
X     :: mechanical_plant and
         produces : (Y :: contains : (Z :: contains : 'CFC')).
```

Instead of variables in nested expressions we can also use unique constants (if a corresponding uc was already created) or an indirect reference.

2.4.7 Closing Roles

Usually we have the possibility to extend an object by an arbitrary number of relations to other objects. However, in certain situations we can be sure that an object can only be related to a certain, fixed number of other objects, and that an extension of the object's role-fillers would be an error. To fix the number of role-fillers at an object we have to **close** the role.

Example 29: Let us assume a plant producing hair spray. Each atomizer contains some gas and the product which should be sprayed. That are all components inside the atomizer, and thus we can close the contains role.

```
hair_spray :: material and chemical_product.
X          :: compound_product and
              contains : close('CFC' and hair_spray).
```

Closing a role has the effect that an atmost restriction is added to the object description which depends on the number of role-fillers given to **close** as argument. Thus, with closing the role-fillers of contains, we state that only the mentioned fillers are related to the object, and no further fillers can occur. An equivalent way to express this is to write 'X :: compound_product and contains : ('CFC' and hair_spray) and atmost(2,contains)'.

Example 30: Let us assume, we describe an atomizer incorrectly as a compound_product which only contains CFC; then this object description is clearly inconsistent w.r.t. the definition of a compound_product which needs to contain at least two products.

X :: compound_product and close(contains : 'CFC').

This object is not recognized as compound_product. Because it has not the required number of role-fillers, it is rejected.

2.4.8 Filling a Role with a Set of Objects

It may happen that we need to fill a role with all known objects at a particular point of time. For example, if we like to build a concept which is defined on the basis of all known objects of a particular type.

Example 31: Let us assume that 'NNPPCD' is the 'National Nuclear Power Plant Controlling Department', which is responsible for controlling all existing nuclear power plants.

'NNPPCD' :: controlling_department and
responsible : allknown(nuclear_power_plants).

Do not worry at the moment how the concept controlling_department is defined, we will define it immediately below. Here **allknown** does the job: it determines all the role-fillers which are instances of nuclear_power_plants at invocation time, and fills the responsible role with this set. Note that **allknown** is nothing more then a macro facility: if after the use of **allknown** new nuclear power plants are added, they will not be recognized as role-fillers for NNPPCD.

We can restrict a role filled with **allknown** by closing the role. This restricts its fillers to only the objects that are known at closing time. Thus, the responsibility of the office can be restricted to only the plants that are currently known.

Example 32: Let us assume that no further nuclear power plants will be build in the future. Thus, we can close the responsible role of NNPPCD, which restricts the responsibility of NNPPCD to all and only the known nuclear power plants at the moment.

'NNPPCD' :: controlling_department and
responsible : close(allknown(nuclear_power_plants)).

2.4.9 Defining Concepts by a Set of Objects

In the beginning of Section 2.2, we have mentioned that in some situations we can define concepts extensionally by enumerating all of their instances if the number of objects constituting such a concept is reasonably small. Since in such definitions objects will play a role, we need to return once more to the definition of concepts. The following illustrates how primitive and defined concept can be defined extensionally.

Example 33: Let us assume that the number of controlling departments in a government is small. This set of objects also does not change very often. Thus, we can represent it as an extensional concept.

controlling_department :< oneof(['NNPPCD', 'CIA', ..., police]) and
atleast(1,responsible).

As should be obvious **oneof** can be regarded as a set construction from an explicitly given list of objects. Although it seems that such a set is similar to an attribute set, there is one major difference between both. Objects are structured, thus the objects in a **oneof** term can have roles and fillers on their own, they are classified as usual, and information about them can be asserted and retracted. This is not the case for attributes, which are regarded as 'atomic values'.

2.5 Querying the System

Of course, once we have created a terminology and asserted some objects and relations, we want to use this information. There are different possibilities for querying BACK. First, because of the distinction between terminological and assertional information. Second, we can distinguish different forms of queries: boolean queries, which deliver only a truth value, and retrieval queries, where we actually like to get some result. We refer in the following with the term *testing* to the first query type and with *retrieval* to the latter type.

We like to point out here that BACK's representation language for concepts and role-fillers is not yet completely described. There are three language constructs which can only be used in queries. However, because of didactical reasons, we delay the description of these constructs until the end of this section. In the following we use the term *entity*[8] in a very broad sense to denote nearly everything, from a concept or role, over a macro or attribute set, to a value and object.

2.5.1 Retrieving Entities

Retrieval of objects can be realized with the **getall** construct, which can be regarded as the BACK analogue to a combination of the 'from' and 'where' modifiers in a 'select' command of conventional, relational database systems.

Example 34: We may like to retrieve e.g. all nuclear_power_plants, all plants which are co_producing some toxic waste, or all products which contain the material 'CFC'.

[8]For a more precise description what an entity can be, see the graphical hierarchy in Figure 3.1 in Chapter 3.

```
backretrieve(getall(nuclear_power_plant)).
backretrieve(getall(plant and some(co_produces,toxic_waste))).
backretrieve(getall(product and contains : 'CFC')).
```

The **getall** construct simply retrieves all objects belonging to the mentioned concept. For retrieving further information about these objects **getall** is usually combined with some output formatting constructs, which will be described in more detail below. As can be seen from these examples, **getall** can be used to formulate complex retrieval queries.

All retrieval commands described in the following may occur in two forms, either prefixed with a variable assignment (X =) or without such a prefix.[9] In the latter case the retrieved information is just pretty printed to the current output stream, while in the former case the Prolog variable is bound to the retrieved result. Just for brevity we describe only the non-prefixed version.

2.5.2 Describing Entities

BACK provides several means (called *output functions* or *actions*) to retrieve different kinds of information for entities, covering more general information such as an entities definition, and more specific information such as an entities filler for a particular role. We start with the action **describe** which is used to retrieve the most specific information known about an entity. The easiest way to use **describe** is to apply it to a simple entity.

Example 35: Let us assume we want to inspect the description of the object 'Dioxin'. This can be realized by the following command:

```
backretrieve(describe 'Dioxin').

>>> 'Dioxin'
      describe: chemical_product
                and toxic_waste
                and (inv(prim(co_produces)) and
                      domain(toxic_waste) and range(plant)
                     ) : 'ChemoPharm Plant 17'
                and oneof(['Dioxin'])
```

Note, that the retrieved description – like the retrieved descriptions below – may contain so called 'primitive components'. Intuitively, 'primitive components' of a primitive concept are exactly those components which make it primitive, i.e., the components the user cannot (or does not want to) define further. Although BACK's parser will recognize primitive components, so that definitions obtained by a retrieval query can be read by BACK, the user can use a 'primitive component' only if it was previouly introduced by BACK. Thus, the user cannot construct arbitrary 'primitive components'.

[9] In contrast to other interaction operations described above, we need to enclose retrieval queries in a *backretrieve* command. This depends on the definition of =/2 in Prolog as an operator with a certain precedence, which cannot be changed without major modifications in the used Quintus libraries.

2.5.3 Applying Output Functions to Multiple Entities

The action **describe** – as well as all other actions and output functions described below – can also be applied to multiple entities. Instead of a simple entity, a list of entities is given as argument, and the action is applied to each element of the list.

Example 36: Let us assume that we need to inspect some concepts and objects at once, e.g. 'Three Miles Island', the known chemical plant which co_produces 'Dioxin' and the concept of a mechanical plant.

```
backretrieve(describe ['Three Miles Island',
                       theknown(chemical_plant and co_produces : 'Dioxin'),
                       mechanical_plant]).

>>> 'Three Miles Island'
    describe: broken_plant
              and dangerous_plant
              and nuclear_power_plant
              and is_of_type : fast_breeder
              and located_at : 'Harrisburg'
              and all(pollution,risky)
              and pollution : high
              and uses_up : 'Uranium'
              and inv(responsible) : 'NNPPCD'
              and oneof(['Three Miles Island'])

>>> theknown(chemical_plant and co_produces:'Dioxin')
    describe: chemical_plant
              and toxic_waste_plant
              and all(pollution,risky)
              and ( co_produces and range(toxic_waste) ) : 'Dioxin'
              and oneof(['ChemoPharm Plant 17'])

>>> mechanical_plant
    describe: plant
              and all(produces,mechanical_product)
```

Another possibility exists for the retrieval of information related to objects. The application of actions or output functions can be combined with **getall**. Such a combined retrieval expression is evaluated by first determining the result set of the **getall** query followed by an application of the action (resp. output function), e.g. **describe**, to each object in the result list.

Example 37: Let us assume that we are interested in the retrieval of descriptions of all instances of materials which are chemical_products.

```
backretrieve(describe getall(material and chemical_product)).
```

```
Instance #1 of getall(material and chemical_product)
      describe: chemical_product
                and material
                and (trans(inv(directly_contains)) and
                      domain(material) and range(product)
                     ) : (uc(268) and uc(265) and uc(264))
                and inv(uses_up) : uc(266)
                and oneof(['CFC'])

Instance #2 of getall(material and chemical_product)
      describe: chemical_product
                and material
                and (trans(inv(directly_contains)) and
                      domain(material) and range(product) ) : uc(268)
                and oneof([hair_spray])
```

Here **getall** produces the desired effect of retrieving all instances of materials which are chemical_products, while the **describe** action produces their description. In contrast to the previous retrieval examples where the entity to be described was given, the term **getall** is evaluated in this example first, and the obtained set of entities is described.

2.5.4 Disambiguating Entities

Although the retrieval mechanism is quite powerful, under certain circumstances we must disambiguate the entity we want to retrieve. I.e., disambiguation via the operator '/' has to be used if some objects, concepts, and attribute domains have the same name. If there are several entities with the same name and the retrieval query is not disambiguated the query will produce an error message and will fail. However, it may happen that the user can not remember whether the entity to be retrieved was introduced as concept, role, aset, attribute_domain, or object. In this case **introduced_as** can be used to disambiguate the entity.

Example 38: Let us assume that the name Three Miles Island is ambiguous and a preceding retrieval query which was not disambiguated failed. Let us further assume that the name was introduced as a concept, an object, and an attribute of an aset. Then a disambiguation via **introduced_as** will produce the following output:

```
backretrieve(X = introduced_as 'Three Miles Island'),
X = [[[conc]-[conc,aset]]]
```

The output produced by **introduced_as** consists of two lists of descriptors separated by '-'. The first list contains at most two elements describing for which "class" the name in question was defined. Here **conc** indicates that Three Miles Island was defined as concept. The second list describes the classes for which an "instance" with the name in question exists. Here **conc** indicates that Three Miles Island is an object instance of a concept, while **aset** indicates that it is an attribute instance of an aset.

2.5.5 Full Description of Entities

The following retrieval actions can be used like the above **describe** action in different forms, either for retrieving information about just one entity, about several entities, or for all objects belonging to a concept. Like stated above, prefixed with a variable, the variable will be bound to the retrieved description. Without prefix the result will be pretty printed.

While the previously described action **describe** produces a minimal description of an entity, a full, possibly redundant description of an entity is produced with the action **describe_fully**.

Example 39: Let us describe some concepts and objects of the previous describe examples to see how a full description of them looks like. Since **describe_fully** produces rather lengthy output, we have excluded the full description of 'Three Miles Island'.

```
backretrieve(describe_fully 'Dioxin').
>>> 'Dioxin'
        describe_fully: anything
                and prim(product)
                and prim(chemical_product)
                and prim(waste)
                and prim(toxic_waste)
                and not(energy)
                and all(inv(co_produces),plant)
                and atleast(1,inv(co_produces))
                and inv(co_produces) : 'ChemoPharm Plant 17'
                and all(inv(prim(co_produces)) and
                        domain(toxic_waste) and range(plant), plant)
                and atleast(1,inv(prim(co_produces)) and
                                domain(toxic_waste) and range(plant))
                and ( inv(prim(co_produces)) and
                      domain(toxic_waste) and range(plant) ) : 'ChemoPharm Plant 17'
                and oneof(['Dioxin'])

backretrieve(describe_fully [theknown(chemical_plant and co_produces : 'Dioxin'),
                             mechanical_plant]).
>>> theknown(chemical_plant and co_produces:'Dioxin')
        describe_fully: anything
                and prim(plant)
                and all(is_of_type,type)
                and atleast(1,is_of_type)
                and atmost(1,is_of_type)
                and all(located_at,place)
                and atleast(1,located_at)
                and atmost(1,located_at)
                and all(co_produces,waste)
                and atleast(1,co_produces)
                and co_produces : 'Dioxin'
                and all(pollution,risky)
                and atmost(1,pollution)
                and all(produces,chemical_product)
```

```
                and all(co_produces and range(toxic_waste),toxic_waste)
                and atleast(1,co_produces and range(toxic_waste))
                and (co_produces and range(toxic_waste)) : 'Dioxin'
                and oneof(['ChemoPharm Plant 17'])

>>> mechanical_plant
    describe_fully: anything
                and prim(plant)
                and all(is_of_type,type)
                and atleast(1,is_of_type)
                and atmost(1,is_of_type)
                and all(located_at,place)
                and atleast(1,located_at)
                and atmost(1,located_at)
                and all(produces,mechanical_product)

backretrieve(describe_fully getall(material and chemical_product)).
Instance #1 of getall(material and chemical_product)
    describe_fully: anything
                and prim(product)
                and prim(chemical_product)
                and prim(material)
                and not(energy)
                and all(inv(contains),product)
                and atleast(3,inv(contains))
                and inv(contains) : (uc(268) and uc(265) and uc(264))
                and all(trans(inv(directly_contains)) and
                        domain(material) and range(product), product)
                and atleast(3,trans(inv(directly_contains)) and
                                domain(material) and range(product))
                and (trans(inv(directly_contains)) and
                     domain(material) and range(product)
                    ) : (uc(268) and uc(265) and uc(264))
                and all(inv(uses_up),plant)
                and atleast(1,inv(uses_up))
                and inv(uses_up) : uc(266)
                and oneof(['CFC'])

Instance #2 of getall(material and chemical_product)
    describe_fully: anything
                and prim(product)
                and prim(chemical_product)
                and prim(material)
                and not(energy)
                and all(inv(contains),product)
                and atleast(1,inv(contains))
                and inv(contains) : uc(268)
                and all(trans(inv(directly_contains)) and
                        domain(material) and range(product), product)
                and atleast(1,trans(inv(directly_contains)) and
                                domain(material) and range(product))
                and ( trans(inv(directly_contains)) and
                      domain(material) and range(product) ) : uc(268)
                and oneof([hair_spray])
```

2.5.6 Retrieving User-Given Descriptions

The above discussed retrieval actions returned not only the description the user gave, but also information that was inferred by BACK. However, the user-given descriptions can be retrieved in isolation with the action **defined_as** as well.

Example 40: Let us use the concepts of the last examples to show how the retrieval of user-given descriptions differs from the above two explained retrieval forms.

```
backretrieve(defined_as 'Dioxin').

>>> 'Dioxin'
      defined_as: 'Dioxin' :: chemical_product and toxic_waste

backretrieve(defined_as ['Three Miles Island',
                         theknown( chemical_plant and
                                   co_produces : 'Dioxin'),
                         mechanical_plant]).

>>> 'Three Miles Island'
      defined_as: 'Three Miles Island' :: broken_plant and
                                          nuclear_power_plant and
                                          is_of_type : fast_breeder and
                                          located_at : 'Harrisburg' and
                                          uses_up : 'Uranium'

>>> theknown(chemical_plant and co_produces:'Dioxin')
      defined_as: 'ChemoPharm Plant 17' :: chemical_plant and
                                          co_produces : 'Dioxin'

>>> mechanical_plant
      defined_as: mechanical_plant := plant and
                                      all(produces,mechanical_product)

backretrieve(defined_as getall( material and chemical_product)).

Instance #1 of getall(material and chemical_product)
      defined_as: 'CFC' :: chemical_product and material

Instance #2 of getall(material and chemical_product)
      defined_as: hair_spray :: chemical_product and material
```

Retrieving the user-given description of terminological entities results in retrieving the current active description, because terminological entities can be redescribed (i.e. overwritten with a new definition). For objects, the union of all user-given active descriptions is retrieved.

2.5.7 Retrieving Combined Information

While the above retrieval operations produced one description per entity, we sometimes need to retrieve the names of certain entities occurring in other entities. Sometimes we also need to retrieve combined information about an entity. For example, we may need to inspect at the same time the user given definition of an entity, the value restrictions of a particular role, and its associated number restrictions. For these purposes the user has the possibility of combining various output functions.

Example 41: Let us consider plants of our formalization. We may need to know the names of all currently known plants, the most specific concept under which they are subsumed, the value restriction of their uses_up role, their associated number restrictions, or all objects used_up by these plants.

```
backretrieve([describe]           for getall(plant)).
backretrieve([describe_fully]     for getall(plant)).
backretrieve([define_as]          for getall(plant)).
backretrieve([introduced_as]      for getall(plant)).
backretrieve([self]               for getall(plant)).
backretrieve([msc]                for getall(plant)).
backretrieve([vr(uses_up)]        for getall(plant)).
backretrieve([nr(uses_up)]        for getall(plant)).
backretrieve([rf(uses_up)]        for getall(plant)).
backretrieve([vr(inv(uses_up))]   for getall(product)).
backretrieve([nr(inv(uses_up))]   for getall(product)).
backretrieve([rf(inv(uses_up))]   for getall(product)).
```

While the first four actions return exactly the same information as described in the previous sections, the output function **self** returns a list of plant names, and **msc** returns a list of possibly incomplete most specific concept names (that is a list of most specific direct super concepts). The other output functions are used for retrieving information about roles resp. inverse roles: **vr** retrieves the value restriction of the specified role, **nr** its number restrictions, and **rf** retrieves the role-fillers. The last three output functions return the **nr**, **nr** resp. **vr** of the inverse role of uses_up.

With a combination of the above described output functions we can also produce more complex results, e.g., a list of object/filler tuples for a certain role, or a list of msc's for the fillers of a certain role.

Example 42: We want to know which plant co_produces which toxic waste. And further we like to retrieve all most specific concepts of all materials contained in all known compound_products.

```
backretrieve(X = [self,rf(co_produces)] for
                 getall(all(co_produces,toxic_waste))).

backretrieve(X = [self,msc] for getall
                 inv(contains) : allknown(compound_product)).
```

In the first case the retrieval result is a list of tuples describing in the first place the plant and in the second place the toxic_waste it produces. The second retrieval results in a list of tuples describing in the first place the material and in the second place a list of the material's msc. The operator **allknown** will be discussed below in Section 2.5.13.

2.5.8 Testing Subsumption

One of the main services of a terminological representation system is the determination of subsumption relations between terms by the classifier.

Example 43: We have defined above nuclear_waste as a primitive form of toxic_waste, and a nuclear_power_plant as a kind of energy_plant which co_produces nuclear_waste. We can determine whether such a plant is also a toxic_waste_plant, if we test whether a subsumption relation exists between these terms. In addition, we can determine by a subsumption test whether toxic_waste is also a product, and whether all plants co_producing waste are producing products.

```
nuclear_power_plant    ?< toxic_waste_plant.
toxic_waste            ?< product.
all(co_produces,waste) ?< all(produces,product).
```

Although the correct answer 'yes' to the latter two questions seems to be strange, it is a consequence of the formalization we made above. Instead of $term_1$ **?**< $term_2$, also **backask**(**subsumes**($term_2$,$term_1$)) can be used to test subsumption. Note that the argument order is different.

2.5.9 Testing Equivalence

Because subsumption is a partial order relation, we can easily determine whether two concepts or roles are equivalent. Here we just show a simple example how equivalence queries can be made. More complex tests can be stated also, but we avoid them for briefness.

Example 44: In the early beginning of this chapter, we have introduced synonyms for the term plant. Was factory a synonym for plant? And was manufactory another synonym?

```
backask(equivalent(factory,plant)).
backask(equivalent(manufactory,plant)).
```

Note that an **equivalent** test needs to be enclosed by the command **backask**. The former test succeeds and yields the result 'yes', thus both terms are synonym. Because manufactory was not yet introduced, the last query is evaluated to 'no'. Thus, the term manufactory cannot be equivalent to plant.

2.5.10 Testing Incoherence and Disjointness

Sometimes the creation of an object leads to an error if BACK can determine that the object is subsumed by an incoherent concept. Thus, it may be useful to determine in advance whether some concepts or roles are incoherent. This can easily be achieved by determining either whether the concept in question is subsumed by **nothing**, by querying whether the concept is **incoherent**, or by testing whether two concepts are **disjoint**.

Example 45: Let us assume we want to create an object which is energy as well as material and are surprised that BACK rejects this object. Thus, we should check whether it is in general possible to use a concept which is both energy and material. A further incoherence concerns products. For example, can their be a product which is a solid_product as well as a compound_product?

energy and material ?< nothing.
backask(disjoint(energy,material)).
backask(incoherent(solid_product and compound_product)).

Like an **equivalent** test, tests for disjointness and incoherence need to be enclosed in a **backask** command. The first example is incoherent because energy and material were declared disjoint; the second disjointness query succeeds for the same reason. The last example is incoherent because of conflicting number restrictions. These are just simple cases; more often we will encounter more difficult cases where the incoherence is not so obvious. Note that the creation of an object as instance of an incoherent concept is not possible; BACK will reject it.

2.5.11 Testing Concept Membership

It may be necessary to test whether an object is an instance of a certain concept. Because we have several ways of creating objects (named, unique, and indirect referenced), BACK offers different possibilities for checking concept membership.

Example 46: Is Three Miles Island a broken plant? Is $uc(266)$ an instance of the concept mechanical_plant? Is the known plant co_producing Dioxin a chemical_plant?

'Three Miles Island'	?: broken_plant.
$uc(266)$	?: mechanical_plant.
theknown(plant and co_produces : 'Dioxin')	?: chemical_plant.

2.5.12 Retrieving the Difference between Entities

For the purpose of inspection and correction, it is sometimes useful to retrieve the difference between two entities of the same type.

Example 47: Suppose at a certain stage in the modeling process we are confused what we intended to represent with the concepts compound_product and assembled_product. Although we can inspect both descriptions, another way of finding out the difference between these concepts is the determination of their "conceptual distance". The **difference** operator is used for this purpose.

```
backretrieve(X = difference(compound_product,assembled_product)).
X = [anything,atleast(2,contains) and all(contains,material)]
```

The result bound to X has to be interpreted as follows: The first element of the list must be added to the entity of the first argument, and the second element must be added to the entity of the second element, to make both semantically equivalent.

2.5.13 Language Constructs Restricted to Queries

As promised in the beginning of this section we return now to three language constructs which still need to be described. While the language constructs introduced in section 2.2 and 2.4 can be used arbitrarily either for modeling or querying, the constructs **or** and **someknown** are restricted to queries only. They are not available for the construction of concepts and the assertion of objects. However, they can be evaluated at query-time.

The use of the **or** operator is restricted to queries only; more precisely, to the outermost level of ABox-queries and to filler expressions of ABox queries.

Example 48: We can use **or** to test, whether Three Miles Island is an instance of a chemical_plant or a nuclear_power_plant, and we can retrieve all plants which are risky_chemical_plants or dangerous nuclear_power_plants.

```
'Three Miles Island' ?: chemical_plant or nuclear_power_plant.
backretrieve(X = [self] for getall(risky_chemical_plant or
                                   (dangerous_plant and nuclear_power_plant))).
```

Note that **or** can only be used for the retrieval of objects with **getall** and for testing whether an object is an instance of a concept term (see ?:).

The concept **or** of the previous example finds its ABox counterpart in the role-filler **or**, which can be used to query for disjunctive role-filler expressions.

Example 49: Does ChemoPharm Plant 17 produce Dioxin or someknown product, which contains CFC? Retrieve all objects, which contain Dioxin or CFC.

```
'ChemoPharm Plant 17' ?: co_produces : ('Dioxin' or
                                        someknown(contains : 'CFC'))
backretrieve(X = [self] for getall(contains : ('Dioxin' or 'CFC'))).
```

Of course, if the number of alternatives in an **or**-query is large it is difficult to list all alternatives. And even if the number of alternatives is small it is sometimes difficult to remember them. Thus, BACK contains a language construct, which can be regarded as a generalized form of the **or** operator, the construct: **someknown**.

Example 50: Let us look again at the example of ChemoPharm Plant 17 co_producing Dioxin introduced above. If we want to find out whether this object co_produces any kind of toxic_waste, we can use the following query.

```
'ChemoPharm Plant 17' ?: co_produces : someknown(toxic_waste).
```

Obviously, the **someknown** operator makes only sense in an ABox query, since BACK allows disjunctive role-filler expressions only in queries.

Chapter 3

Back Manual

This chapter describes the syntax of BACK V5 in detail. Because some of the syntactical constructs (i.e. entities) are used frequently in this chapter, we summarize them by the hierarchy of entities shown in Figure 3.1 for avoiding redundant descriptions in the manual's entries. In general, every entry in this manual consists of a heading line

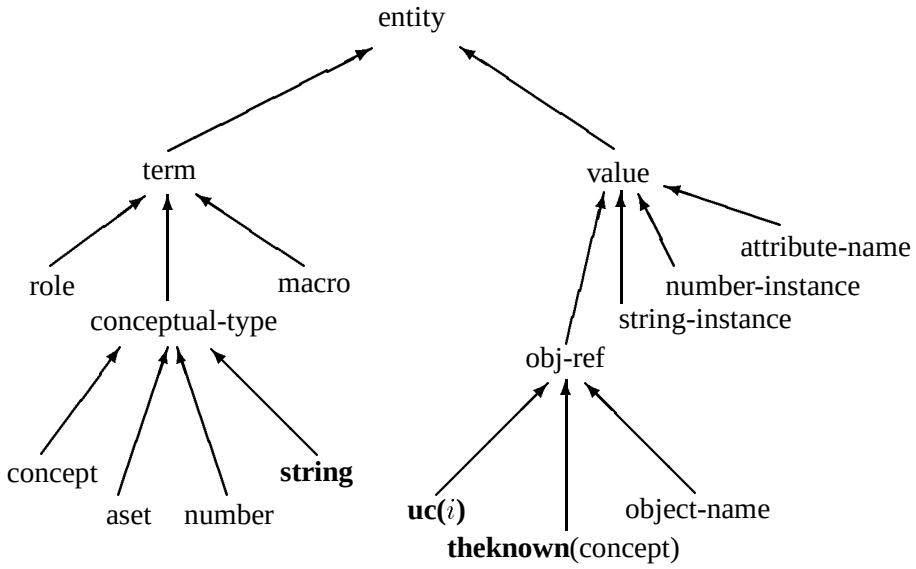

Figure 3.1: Hierarchy of Entities

consisting of the construct's name[1] and a grouping keyword. In the case that several constructs have the same name (e.g. **or**) we group them together. Every subentry will then be distinguished by the name and a grouping keyword. Entries consist of a short summary of the construct, its syntax in BNF, its formal semantics, a detailed description of the construct, some examples, a short description of the difference to BACK V4, and the construct's idiosyncrasies.

[1]For some operators like '..', ':<' etc. we use their Prolog functor as name. A functor consists of a predicate symbol and the number of it's arguments separated by '/'.

:</2 Tell Expression

Synopsis: Introduction or revision of primitive term-names.

Syntax: ⟨*definition*⟩ ::= ⟨*concept*-NAME⟩ :< ⟨*concept*⟩
| ⟨*role*-NAME⟩ :< ⟨*role*⟩

Semantics: $\mathcal{M} \models \mathsf{t}_n :< \mathsf{t}$ iff $[\![\mathsf{t}_n]\!]^{\mathcal{I}} \subseteq [\![\mathsf{t}]\!]^{\mathcal{I}}$

Description: The operator :</**2** is used to introduce *primitive* terms into the knowledge base. For primitive terms only necessary but no sufficient conditions are given. Internally, an introduction of a primitive concept c :< **anything** is transformed into the equivalent definition c := **prim**(c) **and anything** (analogously for primitive roles). **prim**(c) is called a *primitive component*[2] and represents the information contained in c which is not completely specified by the user, i.e., which makes the term primitive.

To revise a primitive introduction, one simply has to give a new definition for a term: if a terminology contains more than one introduction of a term-name $t_n :< t_1, \ldots, t_n :< t_i$, the last introduction is taken to be the actual one; earlier introductions are simply overwritten.

Example:
plant :< **anything**.
chemical_plant :< plant.
produces :< **domain**(plant) **and range**(product).

Idiosyncrasy: The description of a primitive term contains its primitive component, i.e., plant is described by the system as **prim**(plant) **and anything**. This is to ensure that the term returned as description is indeed equivalent to the term being described. Note that a primitive component is only accepted in a definition of a term with the same name as the name occurring within the primitive component. Thus plant **:= prim**(plant) **and anything** is accepted, while dummy **:= prim**(plant) **and anything** is rejected by the system.

See also: :=/**2**

[2] See [Nebel, 1990] for the technical details of introducing primitive components.

:=/2 Tell Expression

Synopsis: Introduction or revision of defined term-names.

Syntax: $\langle$ *definition* $\rangle$ ::= $\langle$ *term*-NAME $\rangle$:= $\langle$ *term* $\rangle$

Semantics: $\mathcal{M} \models \mathsf{t}_n := \mathsf{t}$ iff $[\![\mathsf{t}_n]\!]^{\mathcal{I}} = [\![\mathsf{t}]\!]^{\mathcal{I}}$

Description: The operator :=**/2** is used to introduce *defined* terms into the knowledge base. A term-name is defined and can subsequently be used as an abbreviation of the term given as its definition. Unlike in the case of primitive terms the definition is taken to contain necessary and sufficient conditions, i.e., every instance satisfying the definition is taken to be an instance of the defined name.

To revise a definition, one simply has to give a new definition for a term: if a terminology contains more than one definition for a term-name $t_n := t_1, \ldots, t_n := t_i$, the last definition is taken to be the actual one; earlier definitions are simply overwritten.

Example:
```
mechanical_plant := plant and all(produces,mechanical_product).
co_produces      := produces and range(waste).
contains         := trans(directly_contains).
```

Version 4: In V4 only the definition of concepts was supported, whereas roles could only be introduced as primitive, i.e., defined roles were not allowed.

See also: :</**2**

=>/2 Tell Expression

Synopsis: Specification of a rule.

Syntax: $\langle rule \rangle$ ::= $\langle concept \rangle => \langle concept \rangle$

Semantics: $\mathcal{M} \models c_1 => c_2$ iff $[\![c_1]\!]^{\mathcal{I}} \subseteq [\![c_2]\!]^{\mathcal{I}}$

Description: A rule or constraint is specified: all instances of c_1 are constrained to be also instances of c_2. Rules are taken into account for subsumption queries unless **noibox** is specified. Both c_1 and c_2 can be complex concept terms. Rules between roles are not supported, however.

Example: **some**(co_produces,toxical_waste) => **all**(pollution,high).
nuclear_power_plant **and**
all(is_of_type,'Tschernobyl') => dangerous_plant.

Idiosyncrasy: Cyclic rules, such as c => **all**(r,c) are accepted but inferences for subsumption are incomplete w.r.t. them.

Another incompleteness results from the fact that rules are not treated as material implications but only as forward chaining rules: if an object is known to be a c_1 it will be inferred that it is also a c_2. No contraposition or reasoning by case is employed.

Furthermore, rules are not applied to value restrictions. Thus, a rule $c_1 => c_2$ does not yield a subsumption between **all**(r,c_1) and **all**(r,c_2).

::/2 Tell Expression

Synopsis: Enter new facts into the knowledge base.

Syntax: ⟨*description*⟩ ::= ⟨*obj-ref*⟩ :: ⟨*concept*⟩
| PROLOG-VAR :: ⟨*concept*⟩

Description: The operator **::/2** is used to enter new facts into the knowledge base, and to create new objects. The left hand side, ⟨*obj-ref*⟩ or PROLOG-VAR, determines whether the information of the right hand side, ⟨*concept*⟩, is asserted for an existing object or a new object. If ⟨*obj-ref*⟩ refers to an existing object – by an object name, *unique constant* (**uc(**i**)**), or by a **theknown** expression – then ⟨*concept*⟩ is added to the known object description. Note that in contrast to concept and role definitions, ⟨*concept*⟩ does not replace the object's previous description. If ⟨*obj-ref*⟩ is a Prolog atom not associated with any object, a new object is created whose description is ⟨*concept*⟩, and whose name is ⟨*obj-ref*⟩. If PROLOG-VAR is an unbound Prolog variable, a new object is created whose description is ⟨*concept*⟩; the system generates an internal name, a *unique constant* of the form **uc(**i**)**, and associates it with the object.

If any inconsistency occurs, the assertion is rejected, and **::/2** fails, e.g., if ⟨*concept*⟩ is incoherent or causes an inference that would introduce an inconsistency at another object.

Example:
```
'Harrisburg' :: nuclear_plant.
X            :: plant and name : bio_plant2.
theknown(plant and name : bio_plant2) :: wind_power_plant.
```

Version 4: This operator was called **=/2** in V4. The prefix-operator **new** of V4 has been ommited: if an object name is unknown a new object is created.

Idiosyncrasy: The user must not create objects with a name consisting of a not yet introduced **uc(**i**)**.

See also: **theknown**, **uc(**i**)**

?</2 Ask Expression

Synopsis: Subsumption test.

Syntax: ⟨*ask-expression*⟩ ::= ⟨*term*⟩ ?< ⟨*term*⟩ [**noibox**]

Semantics:

$t_1 ?< t_2$	iff	$\Theta \cup \mathcal{R} \models t_1 \sqsubseteq t_2$
$t_1 ?< t_2$ noibox	iff	$\Theta \models t_1 \sqsubseteq t_2$

Description: The operator performs a boolean test whether the ⟨*term*⟩ on the left hand side is subsumed by the ⟨*term*⟩ on the right hand side. The answer includes the application of rules; if they are to be ignored, the **noibox** option must be used.

Example:

nuclear_plant	?< energy_plant.
atleast(12,r) **and all**(r,d)	?< **atleast**(12,s).
c1	?< c2 **noibox**.

Idiosyncrasy: Note that the order of the arguments of **?<** and **subsumes** differs.

See also: **equivalent**, **subsumes**

?:/2 Ask Expression

Synopsis: Test whether an object instantiates a concept expression.

Syntax: ⟨*ask-expression*⟩ ::= ⟨*obj-ref*⟩ ?: ⟨*concept*⟩ [**noibox**]

Semantics:

o ?: c	iff	$\Gamma \models$ o :: c
o ?: c noibox	iff	$\Theta \cup \mathcal{A} \models$ o :: c

Description: The operator performs a boolean test whether the object referred to by ⟨*obj-ref*⟩ instantiates the description given as ⟨*concept*⟩. ⟨*concept*⟩ may contain disjunctive elements, e.g., disjunctive role-filler expressions. The answer includes the application of rules; if they are to be ignored, the **noibox** option must be used.

Example:
```
'Harrisburg' ?: nuclear_plant.
x23          ?: c2 and s : (y21 or y22) noibox.
```

Version 4: Was **isa**/2.

See also: **getall**

∗=/2 **Macro**

Synopsis: Operator for macro definition.

Syntax:

⟨*macro-definition*⟩ ::= ⟨*macro*⟩ ∗= ⟨*term*⟩
⟨*macro*⟩ ::= ⟨*macro*-NAME⟩[(PROLOG-VAR{,PROLOG-VAR}*)]

See also: **backmacro**

:/2 Concept Term

Synopsis: Specification of role-fillers.

Syntax:

$$\begin{array}{lcl} \langle \textit{concept} \rangle & ::= & \langle \textit{role} \rangle : \langle \textit{filler-expr} \rangle \\ \langle \textit{filler-expr} \rangle & ::= & \langle \textit{value} \rangle \\ & | & (\langle \textit{filler-expr} \rangle) \\ & | & (\langle \textit{description} \rangle)^{\alpha} \end{array}$$

Semantics: $[\![\mathsf{r} : \mathsf{o}]\!]^{\mathcal{I}} = \{d : [\![\mathsf{o}]\!]^{\mathcal{I}} \in [\![\mathsf{r}]\!]^{\mathcal{I}}(d)\}$

Description: An object is specified as a role-filler for a role. This operator allows the use of constants in concept definitions.

Example:

residence :< **range**(country) **type feature**.
italy :: european_country.
italian_company := company **and** residence : italy.
datamont :: company **and** residence : italy.
italian_company ?< **the**(residence,european_country).
no.

Version 4: In V4 this operator was only allowed in the ABox.

Idiosyncrasy: Descriptions of role-fillers have no impact on concept subsumption. That is, asserting the fact that italy is an european country does not make italian_company a subconcept of **the**(residence,european_country).

[. . .] Attribute Set Term

Synopsis: Specification of a list of attributes by enumeration.

Syntax:
⟨*attribute-spec*⟩ ::= '[' ⟨*attribute-list*⟩ ']'
⟨*attribute-list*⟩ ::= ⟨*attribute*-NAME⟩{,⟨*attribute*-NAME⟩}∗

Description: Lists of attribute names can be used for defining attribute sets and for declaring attribute domains. Attribute lists are considered to be ordered, so that subranges can be chosen with the operator **../2**.

Example:
risk := **attribute_domain**([high,large,medium,small,null]).
risky := **aset**([high,large,medium], risk).

See also: **attribute_domain**, **aset**, **../2**

../2 Attribute Set Term

Synopsis: Specification of a range of attributes.

Syntax: ⟨*attribute-spec*⟩ ::= ⟨*attribute*-NAME⟩ .. ⟨*attribute*-NAME⟩

Description: For defining attribute sets on user-defined, closed attribute domains attribute sets can conveniently be defined by specifying a range defined by a first and a last element. The range is evaluated with respect to the list defining the attribute domain. In the example, risky1 and risky2 are equivalent.

Example:
```
risk   := attribute_domain([high,large,medium,small,null]).
risky1 := aset(high..medium, risk).
risky2 := aset([high,large,medium], risk).
```

See also: **[** . . . **]**, **aset**, **attribute_domain**

../2 Number Term

Synopsis: Constructs from a lower and an upper bound a closed numerical interval.

Syntax: $\langle$*number-range*$\rangle$ ::= $\langle$*lower-limit*$\rangle$
| $\langle$*upper-limit*$\rangle$
| $\langle$*lower-limit*$\rangle$ .. $\langle$*upper-limit*$\rangle$

Semantics: $[\![(p_1..p_2)]\!]^{\mathcal{I}} = [\![\mathbf{ge}(p_1)]\!]^{\mathcal{I}} \cap [\![\mathbf{le}(p_2)]\!]^{\mathcal{I}}$

Description: This operator is used to represent the closed numerical interval between the given lower and upper limit, where lower and upper limit are numbers. An interval where the lower limit is equal to the upper limit contains just a single value.

Example: baby := person **and all**(age,0..2).

Idiosyncrasy: Note that the range N..N is equivalent to the number N.

See also: **le, lt**, **ge, gt**, **intersection**

all — Concept Term

Synopsis: Value Restriction.

Syntax: ⟨*concept*⟩ ::= **all**(⟨*role*⟩,⟨*conceptual-type*⟩)

Semantics: $[\![\mathbf{all}(\mathsf{r},\mathsf{c})]\!]^{\mathcal{I}} = \{d : [\![\mathsf{r}]\!]^{\mathcal{I}}(d) \subseteq [\![\mathsf{c}]\!]^{\mathcal{I}}\}$

Description: All fillers for role r must be of type c. Note that this restricts only the fillers locally at a concept. To restrict the fillers of a role globally, the **range** operator must be used.

Example: biological_plant := plant **and all**(produces,biological_product).

Idiosyncrasy: This construct does not imply the existence of any role-filler. Objects having no role-filler for role r(i.e. **atmost**(0,r)), are trivially instances of **all**(r,c) for arbitrary c.

See also: **no**, **some**

allknown — Filler Expression

Synopsis: Embedded conjunctive subquery.

Syntax: ⟨*filler-expr*⟩ ::= **allknown**(⟨*concept*⟩)$^{\alpha}$

Semantics: $\mathtt{allknown(c)} \stackrel{\text{def}}{=} \mathtt{and}\{\mathtt{o} : \Gamma \models \mathtt{o} :: \mathtt{c}\}$
$\mathtt{r} : \mathtt{allknown(c)} \stackrel{\text{def}}{=} \mathtt{anything}\ (\text{if}\ \Gamma \models \mathtt{c} \sqsubseteq \mathtt{nothing})$

Description: With the **allknown** operator an embedded subquery is formulated which returns a filler expression. This filler expression consists of a conjunction of all known instances of the specified concept expression (as if one asks a **getall** query for the given concept, and conjoins all retrieved objects by **and**).

An **allknown** expression is only evaluated once, and is substituted by the resulting filler expression. It is not maintained, however, by BACK. Consequently, if an object is described by means of **allknown**, and later a new instance of the given concept is created, this instance will not be added as a filler to the object described by **allknown**.

Example:
```
?- r :< domain(c) and range(d).
yes
?- y1 :: d, y2 :: d,
   x :: r : allknown(d).
yes
?- x ?: r : allknown(d).
yes
?- x ?: all(r,d). % no: the r-filler set of x is not closed
no
?- y3 :: d, x ?: r : allknown(d).
no
```

Version 4: This operator was called **all** in V4; the change was made to distinguish it from the value restriction operator **all**.

Idiosyncrasy: Note that according to the semantics, all objects will match a query filler expression '⟨*role*⟩: **allknown**(⟨*concept*⟩)' if the extension of ⟨*concept*⟩ is empty.

Only allowed in ABox expressions. A description containing an **allknown** filler-expression depends on the order in which facts are entered into the knowledge base.

See also: **:/2**, **all**, **someknown**, **theknown**

and — Concept Term

Synopsis: Conjunction of concepts.

Syntax: ⟨*concept*⟩ ::= ⟨*concept*⟩ **and** ⟨*concept*⟩

Semantics: $[\![\mathtt{c}_1 \text{ and } \mathtt{c}_2]\!]^{\mathcal{I}} = [\![\mathtt{c}_1]\!]^{\mathcal{I}} \cap [\![\mathtt{c}_2]\!]^{\mathcal{I}}$

Description: The operator **and** is the basic construct for combining concept terms. The resulting term represents the conjunction of both terms.

Example: chemical_plant := plant **and all**(produces,chemical_product).

Version 4: **and** can be used universally to combine concept terms and thus the operator **andwith** which was used in V4 is not needed anymore.

See also: **or**, **not**

and — Role Term

Synopsis: Conjunction of roles.

Syntax: ⟨*role*⟩ ::= ⟨*role*⟩ **and** ⟨*role*⟩

Semantics: $[\![\mathsf{r}_1 \textbf{ and } \mathsf{r}_2]\!]^{\mathcal{I}} = [\![\mathsf{r}_1]\!]^{\mathcal{I}} \cap [\![\mathsf{r}_2]\!]^{\mathcal{I}}$

Description: The operator **and** is used for combining role tems. The resulting term represents the conjunction resp. intersection of both roles.

Example: to_the_south_east_of := to_the_south_of **and** to_the_east_of.

Version 4: In the previous version this operator was only allowed for primitive role introductions.

Idiosyncrasy: The **and** operator for roles is not allowed in ABox tells and queries. Conjunctions consisting of domain and range restrictions only are not allowed within defined role introductions, i.e., r := **domain**(c) **and range**(d). is not accepted.

See also: **domain**, **range**

and — Filler Expression

Synopsis: Conjunction of filler expression.

Syntax: ⟨*filler-expr*⟩ ::= ⟨*value*⟩ **and** ⟨*filler-expr*⟩

Description: The **and** operator allows for the conjunction of filler expressions. As usual, **and** binds stronger than **or**.

Example:
o1 :: c **and** r : (o2 **and** o3 **and** o4).
o1 ?: c **and** r : (o2 **or** o3 **and** o4).
backretrieve(**getall** c **and** r : (**theknown**(d **and** s : y1) **and**
theknown(d **and** s : y2))).

Idiosyncrasy: For each role expression ⟨*role*⟩: ⟨*filler-expr*⟩, the filler expression must be enclosed in parentheses if it contains an **and** or an **or**.

See also: **or**

anything — Concept Term

Synopsis: Built-in topmost concept.

Syntax: ⟨*concept*⟩ ::= **anything**

Semantics: $[\![\mathtt{anything}]\!]^{\mathcal{I}} = D$

Description: **anything** is the topmost concept and can be used to build up primitive concept hierarchies.

Example: product :< **anything**.
plant :< **anything**.

Idiosyncrasy: **anything** is disjoint from the other topmost conceptual types **aset**, **number**, and **string**.

See also: **aset**, **nothing**, **number**, **string**.

aset — Attribute Set Term

Synopsis: Operator for denoting the topmost attribute set and constructing attribute sets by extension.

Syntax:

⟨*aset*⟩ ::= **aset**
| **aset**(‘[’⟨*attribute-list*⟩‘]’)
| **aset**(⟨*attribute-spec*⟩,⟨*domain*-NAME⟩)

Semantics: $[\![\mathtt{aset}(\mathtt{t}_1,\ldots,\mathtt{t}_n)]\!]^{\mathcal{I}} = \{[\![\mathtt{t}_i]\!]^{\mathcal{I}} : 1 \leq i \leq n\}$

Description: The operator **aset** without argument represents the topmost attribute set, which can be considered as a predefined open attribute domain. Other more specific attribute sets are specified by enumerating their extensions, i.e., the set of attributes they denote. An attribute set is defined either on the built-in attribute domain, or on an attribute domain explicitly declared by the user. In this case, the name of the attribute domain must be explicitly provided.

Example:
risky := **aset**([high,large,medium], risk).
risky ?< **aset**.
yes

Idiosyncrasy: **aset** is disjoint from **anything** and the other topmost conceptual types **number** and **string**. Note that in BACK the term *attribute* refers to constants of an attribute domain and not to functional roles like in other terminological systems.

See also: **[** . . . **]**, **../2**, **anything**, **attribute_domain**, **number**, **string**

atleast — Concept Term

Synopsis: Minimum restriction of roles.

Syntax: ⟨*concept*⟩ ::= **atleast(**⟨INTEGER⟩,⟨*role*⟩**)**

Semantics: $[\![\mathbf{atleast}(n, \mathsf{r})]\!]^{\mathcal{I}} = \{d : |[\![\mathsf{r}]\!]^{\mathcal{I}}(d)| \geq n\}$

Description: There are at least n role-fillers for role r.

Example: mono_plant := plant **and atleast**(1,product) **and atmost**(1,product).

Version 4: **atleast** can be used anywhere in ABox descriptions and hence the operator **card** used in V4 is not needed anymore.

See also: **atmost**, **exactly**

atleast — Macro

Synopsis: Qualifying minimum restriction.

Syntax: ⟨*macro-concept*⟩ ::= **atleast**(⟨INTEGER⟩,⟨*role*⟩,⟨*conceptual-type*⟩)

Semantics: $[\![\mathbf{atleast}(n, \mathsf{r}, \mathsf{c})]\!]^{\mathcal{I}} = \{d : |[\![\mathsf{r}]\!]^{\mathcal{I}}(d) \cap [\![\mathsf{c}]\!]^{\mathcal{I}}| \geq n\}$

Description: There are at least n role-filler of type c at role r.

Example: compound_product := product **and atleast**(2,contains,material).

Version 4: Could not be expressed in V4.

Idiosyncrasy: The macro **atleast**(n,r,c) is internally expanded into **atleast**(n,r **and range**(c))

See also: **atmost**, **exactly**

atmost — Concept Term

Synopsis: Maximum restriction of roles.

Syntax: ⟨*concept*⟩ ::= **atmost(**⟨INTEGER⟩,⟨*role*⟩**)**

Semantics: $[\![\mathbf{atmost}(n, \mathsf{r})]\!]^{\mathcal{I}} = \{d : |[\![\mathsf{r}]\!]^{\mathcal{I}}(d)| \leq n\}$

Description: There are at most n role-fillers at role r.

Example: mono_plant := plant **and atleast**(1,product) **and atmost**(1,product).

Version 4: **atmost** can be used anywhere in ABox descriptions and hence the operator **card** used in V4 is not needed anymore.

See also: **atleast**, **exactly**

atmost — Macro

Synopsis: Qualifying maximum restriction.

Syntax: ⟨*macro-concept*⟩ ::= **atmost**(⟨INTEGER⟩,⟨*role*⟩,⟨*conceptual-type*⟩)

Semantics: $[\![\mathbf{atmost}(n, \mathsf{r}, \mathsf{c})]\!]^{\mathcal{I}} = \{d : |[\![\mathsf{r}]\!]^{\mathcal{I}}(d) \cap [\![\mathsf{c}]\!]^{\mathcal{I}}| \leq n\}$

Description: There are at most n role-fillers of type c at role r.

Example: clean_plant := **atmost**(0,produces,toxical_waste).

Version 4: Could not be expressed in V4.

Idiosyncrasy: The macro **atmost**(n,r,c) is internally expanded into **atmost**(n,r **and range**(c))

See also: **atleast**, **exactly**

attribute_domain — Tell Expression

Synopsis: Declaration of new attribute domains.

Syntax:

⟨*declaration*⟩ ::= ⟨*domain*-NAME⟩ := **attribute_domain**
| ⟨*domain*-NAME⟩ := **attribute_domain**(⟨*attribute-list*⟩)

Description: New attribute domains are declared with the **attribute_domain** keyword. They are either open (first alternative) or closed (second alternative). In the first case new attributes can be added any time, whereas in the second case the set is fixed at the time of declaration. Attribute domains are mutually disjoint. A separate name space is opened for each newly declared domain.

Example:

keyword := **attribute_domain**.
switch_state := **attribute_domain**([on,off]).

Version 4: In BACK V4 the built-in attribute domain **attributes** was the union of all user-defined attribute domains. Attribute domains were not necessarily disjoint, since there was only one name space for attributes.

See also: **aset**

backask — Interaction

Synopsis: Performs a boolean query according to the question supplied in the argument.

Syntax: ⟨*interaction*⟩ ::= **backask**(⟨*ask-expression*⟩[**noibox**])

Description: This operator is used to perform a boolean query in the form of an ask-expression on the contents of the knowledge base. With the additional operator **noibox**, the answering process does not take information inferred by IBox rules into account.

Example: **backask**('Harrisburg' ?< nuclear_power_plant **noibox**).
compound_product ?< product.

Version 4: This predicate unifies the functionality of **tboxask**, **aboxask** and **iboxask**.

Idiosyncrasy: The **backask** operator can be omitted for expressions containing the operators **?</2** or **?:/2**, since they uniquely identify an expression as a **backask**.

See also: **backretrieve**

backdump Interaction

Synopsis: Dumps the internal representation of the current knowledge base.

Syntax: ⟨*interaction*⟩ ::= **backdump**[(⟨*file*-NAME⟩)]

Description: **backdump** without argument dumps the the contents of all boxes (TBox, ABox and IBox) to the standard output. With an argument specifying a ⟨*file*-NAME⟩ it dumps the contents into the specified file, so that it can be restored later with **backload**.

Example: **backdump**.
backdump('MyFavoriteFilename').

Version 4: This predicate combines the functionality of **tboxdump** and **abox-dump**.

Idiosyncrasy: The form of the file name depends on your local site, but should be quoted according to the Prolog convention if it contains special characters.

See also: **backload**

backinit — Interaction

Synopsis: Initializes the BACK system partially or totally.

Syntax:

⟨*interaction*⟩	::=	**backinit**[(⟨*box*⟩)]
⟨*box*⟩	::=	**tbox**
	|	**ibox**
	|	**abox**

Description: **backinit** without argument initializes the BACK system completely, that means that the internal data structures are initialized, that state variables are set back to their value at start-up time, and that previously defined macros are removed. With an argument the boxes are initialized as follows:

tbox All boxes are initialized.

ibox Only the IBox is initialized, but also information inferred by the application of rules is removed from the ABox.

abox Only the Abox is initialized.

Example:
backinit.
backinit(**ibox**).

Version 4: This predicate combines the functionality of **tboxinit** and **aboxinit**. In contrast to BACK V4, the abox needs not to be initialized explicitly.

Idiosyncrasy: Note that **backinit** = **backinit**(**tbox**). Note further that after installing the BACK system, it is not yet initialized. Thus, a **backinit** should be issued to initialize the system after installation.

backload — Interaction

Synopsis: Loads a dumped internal representation from a file.

Syntax: ⟨*interaction*⟩ ::= **backload(**⟨*file*-NAME⟩**)**

Description: **backload** loads a previously dumped knowledge base from file ⟨*file*-NAME⟩ back into the BACK system, so that the state of the BACK system is restored to the state of BACK before the knowledge base was dumped.

Example: **backload**('MyFavoriteFilename').

Version 4: This predicate combines the functionality of **tboxload** and **aboxload**.

Idiosyncrasy: The form of the file name depends on your local site, but should be quoted according to the Prolog convention if it contains special characters.

See also: **backdump**

backmacro — Interaction

Synopsis: Definition of a macro.

Syntax: ⟨*interaction*⟩ ::= **backmacro**(⟨*macro-definition*⟩)

Description: The macro-facility can be used to define new term-forming operators or to rename existing term-forming operators. Note that the Prolog-variables occurring in the term on the right-hand side must all be bound by arguments on the left-hand side of the macro.

Example: **backmacro**(all1(R,C) $*$ = **all**(R,C) **and atleast**(1,R)).
backmacro(min(N,R) $*$ = **atleast**(N,R)).

Idiosyncrasy: The macro-facility is restricted to macros for terms. Whole interaction sequences with BACK V5 can be easily defined as Prolog-predicates:

```
my_init :- backinit,
           backstate(verbosity=infos).
```

backread — Interaction

Synopsis: Reads BACK commands from a file.

Syntax: ⟨*interaction*⟩ ::= **backread**(⟨*file*-NAME⟩)

Description: Reads from the specified file interaction operations for building up and accessing a BACK knowledge base. As soon as **backread** encounters a failure, either a syntax error or a **backask** or **backretrieve** which fails, it stops automatically.

The file read by **backread** may contain further interaction operations. Thus, it is possible to issue **backinit**, **backread**, **backwrite**, **backload**, and **backdump** operations from the file read.

For assuring that a file is read into an initialized empty BACK system the file should contain as first statement a **backinit**. For assuring that BACK understands some predefined macros a further statement can be explicitly issued to load them: **backread**(<macro_file_name>).

Depending on the verbosity setting, messages will be produced and written to the current standard output.

Example: **backread**(risk_accessment).

Version 4: This predicate combines the functionality of **tboxread** and **aboxread**.

Idiosyncrasy: The form of the file name depends on your local site, but should be quoted according to the Prolog convention if it contains special characters.

Although a file read by **backread** may contain arbitrary calls to Prolog, it should be noted that this is an additional feature of the Prolog implementation of BACK, which might not be supported in other implementations.

See also: **backwrite**, **backstate**

backretrieve — Interaction

Synopsis: Retrieve information from the system.

Syntax:

⟨*interaction*⟩	::=	**backretrieve**(⟨*retrieval*⟩[**noibox**])
⟨*retrieval*⟩	::=	[PROLOG-VAR =] [⟨*generator*⟩] ⟨*arguments*⟩
⟨*arguments*⟩	::=	⟨*entity*⟩[/⟨*disambig*⟩]
	|	'['⟨*entity*⟩[/⟨*disambig*⟩]{,⟨*entity*⟩[/⟨*disambig*⟩]}* ']'
⟨*disambig*⟩	::=	**conc**
	|	**obj**
	|	⟨*domain*-NAME⟩**^cls**
	|	⟨*domain*-NAME⟩**^obj**

Description: With the **backretrieve** command information is retrieved from the knowledge base. The result is either pretty-printed to the current output stream, or is bound to the PROLOG-VAR. The answer takes the application of rules into account; if they are to be ignored, the **noibox** option must be used. Depending on the setting of the **retrieval** state, a **backretrieve** may fail or still succeed, if an error is encountered in its arguments. In the former case **backretrieve** prints an appropriate error message, in the latter case as many results as possible will be returned.

The retrieval consists of the application of retrieval actions or a tuple generator to the specified arguments. For the possible actions and the generator see the references below. Arguments are either specified explicitly, or are retrieved by means of a **getall** query.

backretrieve requires all specified arguments to be unambigous. Since BACK allows for overlapping name spaces it may be necessary to disambiguate arguments explicitly. This is done by appending to the name a disambiguator, separated from the name by a slash '/'. To express that a name refers to an object or concept, the corresponding keywords **obj** or **conc** are used. To express that a name refers to an attribute-domain the name of the attribute domain has to be used in conjunction with the operator '^' and the keyword **cls**; to express that a name refers to an attribute the name of the attribute domain has to be used in conjunction with the operator '^' and the keyword **obj**.

The disambiguator may alternatively be applied to the entire argument specification list. The entities retrieved with **getall** are interpreted as objects. If one of the explicitly specified names is ambiguous this is treated as an error, yielding an appropriate message or structure to be returned.

Example:

```
?- c :< anything, c :: c.
yes
?- backretrieve(getall c).
```

[c]
yes
?- **backretrieve**(R = **defined_as** c).
[wrong_argument(ambiguous(c))]
yes
?- **backretrieve**(**defined_as** c/**conc**).
>>> c/conc
defined_as:
c :< **anything**
yes
?- **backretrieve**(R=[**self**, **msc**] **for** [c/**obj**]).
R = [[c,[c]]]
yes

Version 4: This operator combines the functionality of **tboxask**, **aboxask**, and **iboxask** of V4.

Users of BACK V4 will notice that the operator **getallrel** is not supported in V5. The extension of a role r can be retrieved by the pattern '**backretrieve**([**self**, **rf**(r)] **for getall anything**)'. The result list may contain objects with an empty filler list. If the user knows the range (-type) of the role the query may be formulated more precisely, e.g., if r has range-type **anything**: '**backretrieve**([**self**, **rf**(r)] **for getall** r : **someknown**(**anything**))'.

See also: **backask**, **defined_as**, **describe**, **describe_fully**, **for**, **getall**, **introduced_as**, **self**, **msc**

backstate — Interaction

Synopsis: Displays, modifies or retrieves the values of global state variables.

Syntax:

⟨*interaction*⟩ ::= **backstate**[(⟨*state*⟩)]

⟨*state*⟩ ::= **verbosity** = **silent**
| **verbosity** = **error**
| **verbosity** = **warning**
| **verbosity** = **infos**
| **verbosity** = **trace**
| **introduction** = **forward**
| **introduction** = **noforward**
| **revision** = **true**
| **revision** = **false**
| **retrieval** = **fail**
| **retrieval** = **succeed**
| **tboxrevision** = **fail**
| **tboxrevision** = **succeed**
| **aboxfilled** = **false**
| **aboxfilled** = **true**
| **aboxfilled** = **abox**
| **iboxfilled** = **false**
| **iboxfilled** = **true**

Description: **backstate** without arguments displays the settings of BACK's global state variables. BACK's state variables are distinguished into user modifiable and read-only variables.

backstate with argument can be used to set or retrieve the value of user-modifiable variables. In the case that the value of a user-modifiable variable is a constant, the state variable is set to this value, if the value is valid. If instead a Prolog variable is used, the actual value of the state variable is retrieved. Modifiable variables are:

verbosity determines which types of output messages are produced.

- **silent** no output is produced
- **errors** only errors are reported
- **warnings** errors and warnings are issued
- **info** additional information is reported
- **trace** produces an exhaustive trace of what happens in BACK.

introduction determines whether undefined names are introduced automatically.

- **noforward** Forward introduction of names is not performed. Thus, names have to be defined before they are used.

forward If a name is used without being previously defined, it will be introduced automatically as a primitive term.

revision determines whether concepts and roles can be revised.

false means revision of concepts or roles is not possible.

true means revisions can be performed.

retrieval determines how failures during retrieval are handled.

fail the retrieval action will fail, if an error is encountered.

succeed means, the retrieval action will retrieve as much as possible and will succeed.

tboxrevision determines how a TBox revision affects the ABox.

fail If a TBox revision fails, the state of the ABox is restored to the state before the revision was tried.

succeed A revision of the TBox should succeed always. If an ABox incoherence is detected, an implicite ABox revision is performed on user-given object definitions.

Note that while forward introduction is useful for small test cases, it may be problematic for modeling large domains – spelling errors will be difficult to detect, since misspelled terms are automatically introduced as new terms.

Read-only state variables can only be retrieved by the user and are set by BACK depending on the processing state. Read-only variables are:

aboxfilled indicates whether objects were created.

false indicates that no object was created.

true indicates that at least one object was created in the TBox.

abox indicates that at least one object was created in the ABox. This value will not be superseded by later creation of TBox objects.

iboxfilled indicates whether rules were asserted.

false indicates that no rule was asserted.

true indicates that at least one rule was asserted.

Instead of the full name of state variables and their values unambiguous abbreviations can be used.

Example: **backstate**(**verbosity** =**errors**).
backstate(ve = err).
backstate(**aboxfilled** = X).

Version 4: This predicate combines the functionality of **tboxstate** and **aboxstate**. Most of the states available in BACK V4 are no longer available.

backtell — Interaction

Synopsis: Tells the BACK system the information conveyed in the argument.

Syntax: ⟨*interaction*⟩ ::= **backtell**(⟨*tell-expression*⟩)

Description: This operator is used to assert new information in the form of tell-expressions.

Example: **backtell**(power_plant := plant **and all**(produces,energy)).
energy :< product.

Version 4: This predicate combines the functionality of **tboxtell**, **aboxtell** and **iboxtell**.

Idiosyncrasy: You may drop the **backtell** for tell-expressions consisting of the operators **:=/2**, **:</2**, **=>/2** or **::/2**, since they uniquely identify an expression as a **backtell**.

backwrite — Interaction

Synopsis: Writes the contents of the knowledge base partially or totally in human readable form into a file.

Syntax:

⟨*interaction*⟩	::=	**backwrite(**⟨*file*-NAME⟩[,⟨*box*⟩])
⟨*box*⟩	::=	**tbox**
	|	**ibox**
	|	**abox**

Description: **backwrite** with one argument writes the definitions read in by **backtell** or **backread** into the file specified by the ⟨*file*-NAME⟩, so that it can be read by an user or by **backread**. With two arguments, the contents of one or several boxes is written into the file, according to the following rules:

- If TBox is specified, only the TBox is written.
- If IBox is specified, the IBox and TBox are written.
- If ABox is specified, all boxes are written.

Example: **backwrite**(modified_risk_accessment).

Version 4: This predicate unifies the functionality of **tboxwrite** and **aboxwrite**, and realizes additional write operations for the IBox.

Idiosyncrasy: Note that **backwrite**(⟨*file*-NAME⟩) = **backwrite**(⟨*file*-NAME⟩,**abox**). The form of the file name depends on your local site, but should be quoted according to the Prolog convention if it contains special characters.

See also: **backread**

close — Filler Expression

Synopsis: Close a filler expression locally.

Syntax: ⟨*filler-expr*⟩ ::= **close**(⟨*filler-expr*⟩)

Semantics: $\mathsf{r} : \mathbf{close}(fe) \stackrel{\text{def}}{=} \mathsf{r} : fe \text{ and } \mathbf{atmost}(n, \mathsf{r})$
where n is the cardinality of the filler-expression

Description: With the **close** operator the filler expression given as the argument is closed locally: the number of fillers in the filler expression is determined, and a corresponding **atmost** restriction is added to the overall description. Thus 'r : **close**(x1 and x2 and x3)' ≡ '**atmost**(3,r) and r : (x1 and x2 and x3).' If the filler expression contains a disjunction or a **someknown** operator, the number of *sure* fillers is used in the **atmost** restriction. *Sure* fillers are the fillers which are contained in every conjunct of the filler expression in disjunctive normal form: 'r : **close**(x1 and (x2 or x3))' ≡ '**atmost**(1,r) and r : ((x1 and x2) or (x1 and x3))' since only x1 is *sure*. Note also that **close** is applied locally on the expression passed as its argument; 'r : (x1 **and close**(x2))' is *not* '**atmost**(2,r) **and** r : (x1 **and** x2))' but '**atmost**(1,r) **and** r : (x1 **and** x2)),' and is thus incoherent. A **close** expression is only evaluated once, and is then substituted by the resulting filler expression. It is not maintained, however, by the system.

The **close** operator is most useful when applied in combination with a **allknown** subquery where the number of (sure) fillers may not be known a priori.

Example: x :: c **and close**(y1 **and** y2).
backretrieve(**getall** c **and** r: **close**(**allknown**(d))).

See also: **:/2**, **allknown**, **someknown**, **theknown**

comp — Role Term

Synopsis: Infix operator for composition of two roles.

Syntax: $\langle role \rangle$ ::= $\langle role \rangle$ **comp** $\langle role \rangle^{r}$

Semantics: $[\![\mathsf{r}_1 \ \mathbf{comp} \ \mathsf{r}_2]\!]^{\mathcal{I}} = [\![\mathsf{r}_1]\!]^{\mathcal{I}} \circ [\![\mathsf{r}_2]\!]^{\mathcal{I}}$

Description: The role operator **comp** produces the composition of two roles. It leads to incompletenesses in several cases, e.g. in combination with the **trans** operator. Additionally during ABox inferences, instances of role chains longer than two are not recognized, if they depend on an instance of a role, which is itself a composition.

Example: is_grandfather_of := is_father_of **comp** is_parent_of.

Version 4: Role composition was not possible in V4.

Idiosyncrasy: The comp/2 operator is not allowed in ABox tells and queries.

See also: **inv**, **trans**

defined_as — Retrieval

Synopsis: Action for retrieving user-given definitions of entities.

Syntax: ⟨*action*⟩ ::= **defined_as**

Description: This action outputs the sorted and non-redundant user-given definition of a concept, role, attribute domain, aset, number, or object which corresponds to the actual definition of the entity given by the user and which was read by BACK. For objects it combines all descriptions given by the user.

Example:
t :< a **and** r : **close**(b **and** c).
backretrieve(**defined_as** t).
>>> t
defined_as:
t :< a **and** r : **close**(b **and** c)

Idiosyncrasy: The arguments on which **defined_as** operates have to be unambiguous.

See also: **backretrieve**, **describe**, **describe_fully**, **introduced_as**

describe — Retrieval

Synopsis: Action for retrieving minimal definitions of entities.

Syntax: ⟨*action*⟩ ::= **describe**

Description: This action constructs a minimal description – in terms of user-given names – of concepts, roles, attribute domains, asets, numbers, or objects. The definition is minimal in the sense, that only a most specific definition is returned. However, in contrast to **defined_as**, this description may contain additional information, which was inferred by BACK.

Example:
t :< a **and** r : **close**(b **and** c).
backretrieve(**describe** t).
>>> t
describe: a
and **prim**(t)
and **atmost**(2,r)
and r : (b **and** c)

Idiosyncrasy: The arguments on which **describe** operates have to be unambiguous.

Primitive concepts and roles are internally represented in BACK V5 as defined concepts resp. roles which contain an additional – so called – primitive component. This primitive component will occur in definitions constructed by **describe**. Although the parser of BACK will correctly recognize primitive components, the user can use this construct in a definition only if the corresponding primitive component was already introduced by BACK. Thus, the user cannot introduce arbitrary primitive components.

See also: **backretrieve**, **defined_as**, **describe_fully**, **introduced_as**

describe_fully — Retrieval

Synopsis: Action for retrieving the complete definition of entities.

Syntax: ⟨*action*⟩ ::= **describe_fully**

Description: This action constructs the maximal description of concepts, roles, attribute sets, asets, numbers and objects. The definition is maximal in the sense, that it contains in addition to the user-given definition all internally inferred information.

Example:

t :< a **and** r : **close**(b **and** c).
backretrieve(**describe_fully** t).
>>> t
describe_fully
anything
and prim(a)
and prim(t)
and all(r,**anything**)
and atleast(2,r)
and atmost(2,r)
and r : (b **and** c)

Idiosyncrasy: The arguments on which **describe_fully** operates have to be unambiguous.

Since in some cases BACK constructs unnamed concepts resp. roles internally, a description constructed by **describe_fully** may contain internally generated keys for which no user-given name exists. Thus, the output produced by **describe_fully** is not necessarily understandable directly for an user. However, for debugging purposes of a knowledge base, the produced information reflects the actual information represented and inferred by BACK and will thus be useful.

Primitive concepts and roles are internally represented in BACK V5 as defined concepts resp. roles, which contain an additional – so called – primitive component. This primitive component will occur in definitions constructed by **describe_fully**. Although the parser of BACK will correctly recognize primitive components, the user can use this construct in a definition only if the corresponding primitive component was already introduced by BACK. Thus, the user cannot introduce arbitrary primitive components.

See also: **backretrieve**, **defined_as**, **describe**, **introduced_as**

difference — Retrieval

Synopsis: Difference between two entities.

Syntax: ⟨*retrieval*⟩ ::= **difference**(⟨*entity*⟩,⟨*entity*⟩)

Description: The semantic difference between two entities is computed. Returned is a list of two terms, which make the first and the second entity semantically equivalent when they are added to the entities. If the entities are objects, the difference between their conceptual descriptions is computed.

Example:
p1 :< **anything**.
p2 :< **anything**.
r1 :< **domain**(p1) **and range**(p2).
p3 :< p2.
c4 := p1 **and all**(r1,p3).
c5 := **atleast**(1,r1).
backretrieve(X = difference(c4,c5)).
X = [**atleast**(1,r1),**all**(r1,p3)]

Idiosyncrasy: If there is a difference concerning two value restrictions, the value restrictions which are returned are not necessarily minimal, but minimal with respect to the concepts already existing in the TBox. In other words, no new concepts are created to answer a difference query.

disjoint — Ask Expression

Synopsis: Tests the disjointness of terms.

Syntax: ⟨*ask-expression*⟩ ::= **disjoint**(⟨*term*⟩,⟨*term*⟩) [**noibox**]

Description: **disjoint** performs a boolean test to determine whether the two terms given as arguments are disjoint, i.e., it conjoins both terms and determines whether the conjoined definition is subsumed by **nothing**. The answer includes the application of rules; if they are to be ignored, the **noibox** option must be used.

Example: **disjoint**(energy,material).
energy **and** material ?< **nothing**.

See also: **?</2**, **incoherent**, **nothing**

domain — Role Term

Synopsis: Restricts the domain of a role to the specified concept.

Syntax: ⟨*role*⟩ ::= **domain**(⟨*concept*⟩)

Semantics: $[\![\mathtt{domain}(\mathsf{c})]\!]^{\mathcal{I}} = [\![\mathsf{c}]\!]^{\mathcal{I}} \times D$

Description: The role operator **domain** restricts the domain of a role, i.e. the first argument of an role instance has to be an instance of the specified concept. The domain has to be of type **concept**, because instances of other types, **aset**, **number** and **string**, are not allowed to have role-fillers.

Example: is_father_of := is_parent_of **and domain**(male).

Version 4: In the previous version this operator was only allowed within primitive role introductions.

Idiosyncrasy: Defined role introductions, containing only a domain restriction, i.e., r := **domain**(c), are forbidden.

Note that the term **domain** usually denotes in database and object-oriented systems what is called **range** in BACK.

See also: **and**, **range**

equivalent — Ask Expression

Synopsis: Tests the equivalence of terms.

Syntax: ⟨*ask-expression*⟩ ::= **equivalent**(⟨*term*⟩,⟨*term*⟩) [**noibox**]

Description: **equivalent** performs a boolean test to determine whether the two terms given as arguments are equivalent, i.e., it tests whether each term is subsumed by the other. The answer includes the application of rules; if they are to be ignored, the **noibox** option must be used.

Example: **equivalent**(workshop,plant).
subsumes(workshop,plant), **subsumes**(plant,workshop).
workshop ?< plant, plant ?< workshop.

See also: **?</2**, **subsumes**

exactly Macro

Synopsis: Minimum and maximum restriction.

Syntax: ⟨*macro-concept*⟩ ::= **exactly**(⟨INTEGER⟩,⟨*role*⟩)
| **exactly**(⟨INTEGER⟩,⟨*role*⟩,⟨*conceptual-type*⟩)

Semantics: $[\![\mathbf{exactly}(n, \mathsf{r})]\!]^{\mathcal{I}} = \{d : |[\![\mathsf{r}]\!]^{\mathcal{I}}(d)| = n\}$
$[\![\mathbf{exactly}(n, \mathsf{r}, \mathsf{c})]\!]^{\mathcal{I}} = \{d : |[\![\mathsf{r}]\!]^{\mathcal{I}} \cap [\![\mathsf{c}]\!]^{\mathcal{I}}| = n\}$

Description: There are exactly n role-fillers at role r, resp. there are exactly n role-fillers of type c.

Example: named_thing := **anything and exactly**(1,name).

Idiosyncrasy: The macro **exactly**(n,r) is internally expanded to **atleast**(n,r) **and atmost**(n,r). The macro **exactly**(n,r,c) is internally expanded to **atleast**(n,r **and range**(c)) **and atmost**(n,r **and range**(c)).

See also: **atleast**, **atmost**

for — Retrieval

Synopsis: Application of a tuple generator on retrieved entities.

Syntax: ⟨*generator*⟩ ::= '['⟨*output-function*⟩ {, ⟨*output-function*⟩}*']' **for**

Description: With the **for** operator it is possible to specify which extra information is going to be retrieved for a set of BACK entities. This set of entities is determined by what follows the **for**, i.e., a **getall** query, a list of entity references, or a single entity reference.

The list preceding the **for** serves as output specification: It is applied to each element of the set of argument entities, and substitutes each entity by an output tuple represented by a Prolog list. Each tuple contains an element for each ⟨*output-function*⟩ in the output specification – the structure of tuple elements depends on the used ⟨*output-function*⟩– and their order is determined by the order of the output specification.

The permitted ⟨*output-function*⟩ include those actions that may be applied directly on retrieved entities (**defined_as**, **describe**, **describe_fully**, **introduced_as**, **msc**, **self**); additionally the operators **nr**, **rf**, and **vr** are supported.

If **backretrieve** is called without the optional Prolog variable to bind the result to, the tuples are pretty printed to the current output stream.

Example:
?- **backretrieve**(**getall** c).
[x1,x2,x3. . ., x23]
yes
?- **backretrieve**([**self**, **rf**(r)] **for getall** c).
[[x1,[y1,y2]], [x2,[ ]], [x3,[y1,y4,y5]],. . ., [x23,[y45]]]
yes
?- **backretrieve**([[x]] = [**self**] **for** x).
yes

Version 4: In V4 several tuples could be returned for a single object, e.g., if it had several role-fillers. Now, there is exactly one tuple for each entity on which the output specification is applied.

See also: **backretrieve**, **defined_as**, **describe**, **describe_fully**, **getall**, **introduced_as**, **msc**, **nr**, **rf**, **self**, **vr**

forget — Tell Expression

Synopsis: Retraction of previously asserted information.

Syntax:

⟨*revision*⟩ ::= **forget** ⟨*rule*⟩
| **forget** ⟨*obj-ref*⟩
| **forget** ⟨*obj-ref*⟩ :: ⟨*concept*⟩

Description: With the **forget** operator information can be retracted from the knowledge base. This applies for three kinds of information: inference rules, partial object descriptions, and entire objects. The retraction will remove all semantical consequences based on the retracted information. Rules have to be passed as argument in the same way they were introduced. If an object reference alone is passed as argument the entire object will be removed. For objects, any part of previously given descriptions may be retracted in an arbitrary combination. System derived facts can not be retracted. If **forget** is called with a description not explicitly told, an error message is raised and the call fails.

Definitions of other BACK entities can not be retracted; instead they can be overwritten by a new definition. If the descriptions of an object should be replaced by a new one, it is advisable to use directly **redescribe** rather than a sequence of **forget** operations in combination with a subsequent assertion.

Example:

?- **backtell**(**forget** c1 => c2).
yes
?- x :: c1, x :: c2 **and all**(r,c3), x :: r:(y **and** z).
yes
?- x ?: **atleast**(2,r).
yes
?- y ?: c3.
yes
?- **backtell**(**forget** x :: **atleast**(2,r)).
ERROR **atleast**(2,r) was not told for x
no
?- **backtell**(**forget** x :: c1 **and all**(r,c3)).
yes
?- y ?: c3.
no
?- **backtell**(**forget** z).
yes
?- x ?: r:z.
no

See also: **redescribe**

ge, gt — Number Term

Synopsis: Constructs a numerical interval with infinite upper bound.

Syntax:

⟨*lower-limit*⟩ ::= **ge**(⟨*number*-INSTANCE⟩)
| **gt**(⟨*number*-INSTANCE⟩)
| ⟨*number*-INSTANCE⟩

Semantics:

$[\![\mathbf{ge}(p)]\!]^{\mathcal{I}} = \{p_1 : p_1 \geq p\}$
$[\![\mathbf{gt}(p)]\!]^{\mathcal{I}} = \{p_1 : p_1 > p\}$

Description: These operators construct from a given number an interval with an infinite upper bound and either closed lower bound (in case of **ge**) or open lower bound (in case of **gt**).

Example:

adult := person **and all**(age,**ge**(18)).
retired_woman := woman **and all**(age,**gt**(63)).

See also: **../2**, **le, lt**, **intersection**

getall — Retrieval

Synopsis: Retrieval of instances.

Syntax:
$\langle arguments \rangle$ ::= **getall** $\langle concept \rangle$
| **getall** $\langle aset \rangle$
| **getall string**

Semantics: $o \in \mathtt{getall}(c)$ iff $\Gamma \models o :: c$

Description: With the **getall** keyword a query is issued that retrieves all *known instances* of a given $\langle term \rangle$. $\langle term \rangle$ is limited to concepts, asets, and the built-in type **string**. The known instances of a concept are those explicitly introduced objects which instantiate the queried concept. The known instances of a aset are those attributes that were introduced for the queried aset. The known instances of **string** are those string values that have been used in the knowledge base as fillers of a role with range **string**.

Example: **backretrieve**(**getall** c **and** r : x **and** s : (y or z)).
backretrieve(R = **getall aset**(high .. medium,risk)).

Version 4: The operator **getallrel** is not supported in V5, neither is a role an admissible argument for the **getall** operator. But see the entry for **backretrieve**.

Idiosyncrasy: **getall** queries for number ranges are not supported. First, number instances in BACK V5 are not limited to integers; thus an enumeration of all known numbers in a given range is impossible. Second, we do not consider the enumeration of thousands of values, as e.g. with '**getall** 0 .. 1999', as really useful.

See also: **backretrieve**, **defined_as**, **describe**, **describe_fully**, **for**, **introduced_as**, **self**, **msc**

incoherent — Ask Expression

Synopsis: Tests whether a term is incoherent.

Syntax: ⟨*ask-expression*⟩ ::= **incoherent**(⟨*term*⟩) [**noibox**]

Description: **incoherent** tests whether the term given as argument is incoherent, i.e., whether it is subsumed by **nothing**. The answer includes the application of rules; if they are to be ignored, the **noibox** option must be used.

Example: **incoherent**(**atleast**(2,r) **and atmost**(1,r)).
incoherent(energy **and** material).
atleast(2,r) **and atmost**(1,r) ?< **nothing**.

See also: **?</2**, **disjoint**, **nothing**

intersection — Attribute Set Term

Synopsis: Operator for intersecting two attribute sets.

Syntax: ⟨*aset*⟩ ::= (⟨*aset*⟩ **intersection** ⟨*aset*⟩)

Semantics: $[\![(\mathtt{a}_1 \ \mathbf{intersection} \ \mathtt{a}_2)]\!]^{\mathcal{I}} = [\![\mathtt{a}_1]\!]^{\mathcal{I}} \cap [\![\mathtt{a}_2]\!]^{\mathcal{I}}$

Description: The **intersection** operator for attribute sets corresponds to the **and** operator for concepts. It denotes the intersection of two attribute sets. Note that the intersection of two disjoint asets results in an empty aset which is equivalent to **nothing**.

Example:
```
risk        := attribute_domain([high,large,medium,small,null]).
risky       := aset([high,large,medium], risk).
medium_risk := risky intersection aset([medium,small,null])
```

See also: **aset**, **union**, **without**

intersection — Number Term

Synopsis: Constructs the intersection of two numerical intervals.

Syntax: ⟨*number*⟩ ::= (⟨*number*⟩ **intersection** ⟨*number*⟩)

Semantics: $[\![(p_1 \ \mathbf{intersection} \ p_2)]\!]^{\mathcal{I}} = [\![p_1]\!]^{\mathcal{I}} \cap [\![p_2]\!]^{\mathcal{I}}$

Description: Constructs the intersection interval of two numerical intervals. If both intervals are disjoint, i.e., have no value in common, the result will be the empty interval which is equivalent to **nothing**.

Example: working_person := person **and** working **and**
all(age,**ge**(16) **intersection le**(63)).

See also: **../2**, **le, lt**, **ge, gt**

introduced_as — Retrieval

Synopsis: Action for disambiguating entities.

Syntax: ⟨*action*⟩ ::= **introduced_as**

Description: This action disambiguates entities. It produces as output two lists of descriptors separated by '-'.

The first list contains at most two descriptors which describe the "classes" for which the entity in question exists. One of these descriptors is either **conc**, **role**, **aset**, or **number**, while the other descriptor can be the name of an attribute domain.

The second list may contain an arbitrary list of descriptors describing all "classes" for which the entity in question is an "instance". It may consist of the descriptors **conc**, **aset**, **string**, or the names of attribute domains in which the entity in question is an instance.

Example:
```
c     := all(r,d).
foo   := aset([a,b,c]).
name :< range(string).
c     :: c and name : c.
backretrieve(L = [self,introduced_as] for c/obj).
      L = [[c,[conc]-[aset,string,conc]]]
```

Version 4: This was not needed in V4, since only a single name space was maintained.

Idiosyncrasy: Note that **conc** in the second list denotes objectes. While the name of an attribute domain in the first list denotes the attribute domain itself, the name of an attribute domain denotes in the second list the attribute domain in which the entity in questions occurs as attribute.

See also: **backretrieve**, **defined_as**, **describe**, **describe_fully**

inv — Role Term

Synopsis: Construction of inverse roles.

Syntax: ⟨*role*⟩ ::= **inv**(⟨*role*⟩)

Semantics: $[\![\mathbf{inv}(\mathsf{r})]\!]^{\mathcal{I}} = \{\langle d, e\rangle \in D \times D : \langle e, d\rangle \in [\![\mathsf{r}]\!]^{\mathcal{I}}\}$

Description: **inv** is a role operator for defining the inverse of a role. The argument of **inv** may be a role name or an arbitrary role term.

Example: is_child_of := **inv**(is_parent_of).

Version 4: In V4 **inv** was only allowed in ABox queries.

Idiosyncrasy: Note that you can not invert a role if its range is of type **aset**, **number** or **string**.

See also: **comp**, **trans**

le, lt — Number Term

Synopsis: Constructs a numerical interval with infinite lower bound.

Syntax: ⟨*upper-limit*⟩ ::= **le**(⟨*number*-INSTANCE⟩)
| **lt**(⟨*number*-INSTANCE⟩)
| ⟨*number*-INSTANCE⟩

Semantics: $[\![\mathbf{le}(p)]\!]^{\mathcal{I}} = \{p_1 : p_1 \leq p\}$
$[\![\mathbf{lt}(p)]\!]^{\mathcal{I}} = \{p_1 : p_1 < p\}$

Description: These operators construct from a given number an interval with infinite lower bound and an either closed upper bound (in case of **le**) or an open upper bound (in case of **lt**).

Example: child := person **and all**(age,**lt**(18)).
poor_person := person **and all**(income_in_DM,**le**(1000)).

See also: **../2**, **ge, gt**, **intersection**

msc — Retrieval

Synopsis: Retrieving an entity's MSC set.

Syntax: ⟨*action*⟩ ::= **msc**

Description: The operator **msc** retrieves the MSC set of an object or attribute. The MSC set is the set of the most specific concepts instantiated by an object, or the most specific attribute sets an attribute belongs to. The MSC set is represented by a list which contains the names of the most specific concepts or attribute sets. In the case of objects, the elements in the MSC set are either the concept **anything**, or concepts defined by the user; in the case of attributes, the elements in the MSC set are either the predefined attribute set **aset**, or attribute domains or attribute sets defined by the user.

Example:

?- **backretrieve**(**describe** x17).
>>> x17
describe: c1
and c2
and atleast(2,r)
and r : uc(24)
and oneof([x17])
yes
?- **backretrieve**(R=[**self**,**msc**] **for** x17).
R = [[x17,[c1,c2]]]
yes

See also: **defined_as**, **describe**, **describe_fully**, **for**, **getall**, **introduced_as**, **nr**, **rf**, **self**, **vr**

name — Tell Expression

Synopsis: Renaming of objects.

Syntax: ⟨*revision*⟩ ::= **name(**⟨*obj-ref*⟩,⟨*object*-NAME⟩**)**

Description: This operator is used to name objects which were previously named by BACK automatically with an unique constant **uc(**i**)**. Note that once an **uc(**i**)** object was renamed this way, it cannot be renamed again.

Example: **backtell**(**name**(uc(124),'My favorite object name')).

Version 4: This operation was performed in V4 through the introduction of an *alias name* with the operator **new**.

Idiosyncrasy: With this operation it is only possible to rename previously introduced **uc(**i**)**'s; renaming of named objects or naming of not yet introduced objects is not possible.

See also: **uc(**i**)**

no — Macro

Synopsis: Nonexistence restriction.

Syntax: ⟨*macro-concept*⟩ ::= **no**(⟨*role*⟩,⟨*conceptual-type*⟩)
| **no**(⟨*role*⟩)

Semantics: $[\![\mathbf{no}(\mathsf{r})]\!]^{\mathcal{I}} = \{d : [\![\mathsf{r}]\!]^{\mathcal{I}}(d) = \emptyset\}$
$[\![\mathbf{no}(\mathsf{r},\mathsf{c})]\!]^{\mathcal{I}} = \{d : [\![\mathsf{r}]\!]^{\mathcal{I}}(d) \cap [\![\mathsf{c}]\!]^{\mathcal{I}} = \emptyset\}$

Description: There are no role-fillers at role r, resp. there are no role-fillers of type c at role r.

Example: broken_plant := plant **and no**(produces).

Version 4: **no**(r,c) could not be expressed in V4.

Idiosyncrasy: The macro **no**(r) is internally expanded into **atmost**(0,r). The macro **no**(r,c) is internally expanded into **atmost**(0,r **and range**(c)).

See also: **all**, **some**

not — Concept Term

Synopsis: Negation of primitive concepts.

Syntax: $\langle concept \rangle$ ::= **not**($\langle concept \rangle$)$^{\pi}$

Semantics: $[\![\mathbf{not}(\mathsf{c}_p)]\!]^{\mathcal{I}} = D \setminus [\![\mathsf{c}_p]\!]^{\mathcal{I}}$

Description: The **not** operator can be used to negate primitive concepts. Therefore its argument has to be the name of a concept defined by c :< term.

If c is not a primitive concept but a defined one, the term is rejected. Note that only the primitive component is negated, thus **not**(c) is actually interpreted as **not**(**prim**(c)). If c is introduced, for example, as c :< **anything and atleast**(1,r), than **not**(c) does *not* contain the information **atmost**(0,r). Since the negation is symmetric from c_1 := **not**(c_2) it will be inferred that no instance of c_2 is an instance of c_1, and vice versa. Thus **not** can be used to express disjointness.

Example:
energy :< product.
material :< product **and not**(energy).
waste :< product **and not**(energy).

Version 4: Was expressed as disjointness restriction in V4. To guarantee upward compatibility the system still accepts disjointness restrictions. They should not be used, however, since they can lead to inefficient revisions. E.g. consider the following tells:

energy :< product.
material :< product.
disjoint(energy,material).

These will be transformed by the system into the tells:

energy :< product.
material :< product.
material :< product **and not**(energy).

Idiosyncrasy: c must be a primitive concept. Only its primitive component is negated.

See also: **and**, **or**

not — Role Term

Synopsis: Negation of primitive roles.

Syntax: $\langle role \rangle$ ::= **not**($\langle role \rangle$)$^{\pi}$

Semantics: $[\![\mathbf{not}(\mathsf{r}_p)]\!]^{\mathcal{I}} = D \times D \setminus [\![\mathsf{r}_p]\!]^{\mathcal{I}}$

Description: The **not** operator can be used to negate primitive roles. Therefore its argument has to be the name of a role defined by r :< term. With **not** disjointness of primitive roles can be defined in the same way as for concepts. It is handled by the system quite similar to the **not** operator for primitive concepts.

Example: to_the_south_of :< **anyrole**
to_the_north_of :< **not**(to_the_south_of).

Version 4: Disjointness of roles could not be expressed in V4.

nothing — Term

Synopsis: The incoherent concept.

Syntax: ⟨*concept*⟩ ::= **nothing**

Semantics: $[\![\mathbf{nothing}]\!]^{\mathcal{I}} = \emptyset$

Description: The incoherent concept which has no instances at all. It can be used to check the incoherence of concepts, e.g., c ?< **nothing**, or to check whether value restrictions are incoherent, e.g., c ?< **all**(r,**nothing**). It is also useful for specifying non-primitive disjointness via rules, e.g., **atleast**(2,r) **and atmost**(3,s) => **nothing**.

Example: material **and** energy ?< **nothing**.

Idiosyncrasy: **nothing** stands for incoherent concepts, asets, numbers, and roles.

See also: **anything**, **disjoint**, **incoherent**

nr — Retrieval

Synopsis: Retrieving an entity's cardinality information for a given role.

Syntax: ⟨*output-function*⟩ ::= **nr**(⟨*role*⟩)
| **nr**(**inv**(⟨*role*⟩))

Description: The operator **nr** retrieves the cardinality information (*number-restriction*) of a BACK entity. For each entity to which it is applied **nr** adds to the output tuple a structure m–n where m is the most specific minimum cardinality of the filler set, and n is the most specific maximum cardinality of the filler set. If the maximum cardinality is unrestricted, **in** is returned as the maximum.

Example:
?- x :: **atmost**(4,r) **and** r:(y **and** z).
yes
?- **backretrieve**(R = [**self**, **nr**(r)] **for** [x,y]).
R = [[x,2–4], [y,0-**in**]]
yes

Idiosyncrasy: **nr** is only applicable to entities that may have roles and fillers, i.e., to concepts and their instances.

See also: **defined_as**, **describe**, **describe_fully**, **for**, **getall**, **introduced_as**, **rf**, **self**, **vr**

number — Number Term

Synopsis: Built-in topmost number.

Syntax: ⟨*number*⟩ ::= **number**

Description: **number** is the topmost number.

Example:
20th_century := 1900 .. 1999.
20th_century ?< **number**.
yes

Idiosyncrasy: **number** is disjoint from **anything** and the other topmost conceptual types **aset** and **string**.

See also: **aset**, **anything**, **string**

oneof — Concept Term

Synopsis: Extensional concept definition.

Syntax: ⟨*concept*⟩ ::= **oneof**(‘[’⟨*object*-NAME⟩){,⟨*object*-NAME⟩}*‘]’

Semantics: $[\![\mathbf{oneof}([\mathsf{o}_1, \ldots, \mathsf{o}_n])]\!]^{\mathcal{I}} = \{[\![\mathsf{o}_1]\!]^{\mathcal{I}}, \ldots, [\![\mathsf{o}_n]\!]^{\mathcal{I}}\}$

Description: An extensional concept term is defined by its instances o_1,...,o_n. This construct is similar to an **aset** description, but the attributes mentioned in an **aset** cannot be further described in the ABox. The instances mentioned in a **oneof** description, on the other hand, are full-fledged ABox objects and can have roles on their own, etc.

Example: skandinavian_country := **oneof**([denmark,finland,norway,sweden]).

Version 4: Could not be expressed in V4.

Idiosyncrasy: If intensional and extensional specifications are mixed (e.g., skandinavian_country := country **and oneof**([denmark, finland, norway, sweden]) it does not follow semantically that the constants mentioned extensionally are instances of the defined concept. Thus the above definition does not entail that denmark is a skandinavian_country, since it is not asserted that denmark is a country.

See also: **aset**

or — Concept Term

Synopsis: Concept disjunction.

Syntax: $\langle concept \rangle \quad ::= \quad \langle concept \rangle$ **or** $\langle concept \rangle^{?\top}$

Semantics: $[\![\mathtt{c}_1 \textbf{ or } \mathtt{c}_2]\!]^{\mathcal{I}} = [\![\mathtt{c}_1]\!]^{\mathcal{I}} \cup [\![\mathtt{c}_2]\!]^{\mathcal{I}}$

Description: The instances of $\mathtt{c}_1$ **or** $\mathtt{c}_2$ are all objects which are either instances of $\mathtt{c}_1$ or of $\mathtt{c}_2$, or of both. This concept constructor can only be used in ABox queries at the top level and is implemented as a simple union of the instances retrieved for the combined queries.

Example: **backretrieve**(**getall**(chemical_plant **or** biological_plant)).

Version 4: In V4 the operator **union** was used to retrieve the union of two ABox queries.

Idiosyncrasy: Since this operator is implemented as a simple union of the instances retrieved for the combined query, the system does not treat disjunction in a complete way. Thus a query **getall**(**atleast**(1,r) **or atmost**(0,r)) does not return all objects (as would be semantically correct), but only those for which it is known that they have at least one role-filler or that they cannot have a role-filler.

See also: **and**, **not**

or — Filler Expression

Synopsis: Disjunction of filler expression.

Syntax: ⟨*filler-expr*⟩ ::= ⟨*value*⟩ **or** ⟨*filler-expr*⟩$^{?\alpha}$

Description: The **or** operator allows for the disjunction of filler expressions. As usual, **or** binds weaker than **and**.

Example:
```
o1 ?: c and r : (o2 or o3 or o4).
o1 ?: c and r : ((o2 or o3) and o4).
backretrieve(getall c and r : (theknown(d and s : y1)
                                or theknown(d and s : y2))).
```

Idiosyncrasy: Disjunction of filler expressions is only allowed in ABox queries. For each role expression ⟨*role*⟩:⟨*filler-expr*⟩, the filler expression must be enclosed in parentheses if it contains an **and** or an **or**.

See also: **and**

range — Role Term

Synopsis: Restricts the range of a role.

Syntax: ⟨*role*⟩ ::= **range**(⟨*conceptual-type*⟩)

Semantics: $[\![\mathtt{range(c)}]\!]^{\mathcal{I}} = D \times [\![\mathtt{c}]\!]^{\mathcal{I}}$

Description: **range** is a role operator that restricts the fillers of a role to instances of a given entity, i.e., concepts, numbers, asets or string. A role can be inverted only if its range is of type concept. Defined role introductions, containing only a range restriction, i.e. r := **range**(c) are forbidden.

Example: is_son_of := is_child_of **and range**(male).

Version 4: In V4 the operator was only allowed in primitive role introductions.

Idiosyncrasy: The **range** operator is not allowed in ABox tells and queries. If a role is introduced without an explicit range restriction, then the system sets its range to **anything**. Especially all forward introduced roles get **range**(**anything**).

See also: **inv**, **domain**

redescribe — Tell Expression

Synopsis: Redescribe an object.

Syntax: ⟨*revision*⟩ ::= **redescribe**(⟨*obj-ref*⟩ :: ⟨*concept*⟩)

Description: With the **redescribe** command an object's current description is replaced by the description passed as the argument. Together with the replaced description all of its semantic consequences are removed from the knowledge base.

redescribe may be thought of as a shortcut for first retrieving an object's current description, retracting it from the knowledge base by **forget**, and asserting a new description. Using **redescribe** is faster than performing the sequence of single steps. On the other hand, if the new description is a monotonic extension of the former description, a normal object tell should be issued rather than **redescribe**.

Example:

```
?- x :: c1, x :: c2 and all(r,c3), x :: r:(y and z).
yes
?- y ?: c3.
yes
?- backtell(redescribe( x :: c1 and r:y)).
yes
?- y ?: c3.
no
?- x ?: r:z.
no
```

See also: ::/2, **forget**

rf — Retrieval

Synopsis: Retrieving an entity's fillers for a given role.

Syntax: ⟨*output-function*⟩ ::= **rf**(⟨*role*-NAME⟩)
| **rf**(**inv**(⟨*role*-NAME⟩))

Description: The operator **rf** retrieves the role-fillers of a BACK entity. For each entity to which it is applied **rf** adds to the output tuple a list containing the entity's known fillers for the specified role; if the entity has no known fillers the list is empty. Fillers are given by their names, or *unique constant*s (**uc(***i***)**) if no name has been provided.

Example:
?- **backretrieve**([**self**, **rf**(r)] **for getall** c).
[[x1,[y1,y2]], [x2,[ ]], [x3,[y1,y4,y5]],. . . , [x23,[y45]]]
yes
?- **backretrieve**(R = [**rf**(**inv**(r)),**self**, **rf**(s)] **for** [y1,x1]).
R = [[[x1,x3,uc(182)],y1,[z1,z2]],[[ ],x1,[z3]]]
yes

Version 4: In V4 only one filler at a time was retrieved; if a tuple was in the result relation but no filler existed for a requested role, an uninstantiated Prolog variable was returned. In the second of the above examples, for y1 six different tuples would have been returned (all possible combinations of the retrieved fillers, e.g., [x3,y1,z2]); for x1 a single tuple [V,x1,z3] would have been returned, where V is a variable.

Idiosyncrasy: **rf** is only applicable to entities that may have roles and fillers, i.e., to concepts and their instances.

See also: **defined_as**, **describe**, **describe_fully**, **for**, **getall**, **introduced_as**, **msc**, **nr**, **self**, **vr**

rvm_some — Macro

Synopsis: Existence role value map.

Syntax: ⟨*macro-concept*⟩ ::= **rvm_some**(⟨*role*⟩,⟨*role*⟩)

Semantics: $[\![\mathbf{rvm\_some}(\mathsf{r}_1, \mathsf{r}_2)]\!]^{\mathcal{I}} = \{d : [\![\mathsf{r}_1]\!]^{\mathcal{I}}(d) \cap [\![\mathsf{r}_2]\!]^{\mathcal{I}}(d) \neq \emptyset\}$

Description: There is at least one object being a role-filler both at r_1 and at r_2. This operator is symmetric, i.e. , rvm_some(r_1,r_2) is equivalent to rvm_some(r_2,r_1).

Example: happy_employee := **rvm_some**(colleague,friend).

Version 4: Could not be expressed in V4.

Idiosyncrasy: The macro **rvm_some**(r_1,r_2) is expanded into **atleast**(1,r_1 **and** r_2)

See also: **rvm_no**

rvm_no — Macro

Synopsis: Nonexistence role value map.

Syntax: ⟨*macro-concept*⟩ ::= **rvm_no**(⟨*role*⟩,⟨*role*⟩)

Semantics: $[\![\mathbf{rvm\_some}(\mathsf{r}_1, \mathsf{r}_2)]\!]^{\mathcal{I}} = \{d : [\![\mathsf{r}_1]\!]^{\mathcal{I}}(d) \cap [\![\mathsf{r}_2]\!]^{\mathcal{I}}(d) = \emptyset\}$

Description: There is no object being a role-filler both at r_1 and at r_2. This operator is symmetric, i.e. , rvm_no(r_1,r_2) is equivalent to rvm_no(r_2,r_1).

Example: unhappy_employee := **rvm_no**(colleague,friend).

Version 4: Could not be expressed in V4.

Idiosyncrasy: The macro **rvm_no**(r_1,r_2) is expanded into **atmost**(0,r_1 **and** r_2)

See also: **rvm_some**

self — Retrieval

Synopsis: Retrieving an entity's name.

Syntax: ⟨*action*⟩ ::= **self**

Description: The operator **self** retrieves the name of a BACK entity. If the entity has no user-given name its internal identifier is returned in external representation, i.e., a *unique constant* **uc(***i***)** for objects, **conc**(*i*) for concepts, etc. Typically **self** is used in two situations: 1) to find the name of a described object, and 2) to include the entity in larger tuples when required by further processing of the result relation.

Example:
?- **backretrieve**(**getall** c19).
[x1,x2]
yes
?- **backretrieve**([**self**] **for getall** c19).
[[x1], [x2]]
yes
?- **backretrieve**(**self theknown**(c68 and r : y76)).
[x23]
yes

See also: **defined_as**, **describe**, **describe_fully**, **for**, **getall**, **introduced_as**, **msc**, **nr**, **rf**, **vr**

some — Macro

Synopsis: Existence restriction.

Syntax: ⟨*macro-concept*⟩ ::= **some**(⟨*role*⟩,⟨*conceptual-type*⟩)
| **some**(⟨*role*⟩)

Semantics: $[\![\mathbf{some}(\mathsf{r})]\!]^{\mathcal{I}} = \{d : [\![\mathsf{r}]\!]^{\mathcal{I}}(d) \neq \emptyset\}$
$[\![\mathbf{some}(\mathsf{r},\mathsf{c})]\!]^{\mathcal{I}} = \{d : [\![\mathsf{r}]\!]^{\mathcal{I}}(d) \cap [\![\mathsf{c}]\!]^{\mathcal{I}} \neq \emptyset\}$

Description: There is at least one role-filler at role r, resp. there is at least one role-filler of type c at role r.

Example: assembled_product := product **and some**(contains,product).

Version 4: **some**(r,c) could not be expressed in V4.

Idiosyncrasy: The macro **some**(r) is internally expanded into **atleast**(1,r). The macro **some**(r,c) is internally expanded into **atleast**(1,r **and range**(c))

See also: **all**, **no**, **atleast**

someknown — Filler Expression

Synopsis: Embedded disjunctive subquery.

Syntax: ⟨*filler-expr*⟩ ::= **someknown**(⟨*concept*⟩)$^{?\alpha}$

Semantics: $\mathtt{someknown(c)} \stackrel{\text{def}}{=} \mathbf{or}\{\mathtt{o} : \Gamma \models \mathtt{o} :: \mathtt{c}\}$
$\mathtt{r} : \mathtt{someknown(c)} \stackrel{\text{def}}{=} \mathbf{nothing}\ (\text{if}\ \Gamma \models \mathtt{c} \sqsubseteq \mathbf{nothing})$

Description: With the **someknown** operator an embedded subquery is formulated which returns a filler expression. This filler expression consists of a disjunction of all known instances of the specified concept expression (as if one asks a **getall** query for the given concept, and conjoins all retrieved objects by **or**).

A **someknown** expression is only evaluated once, and is substituted by the resulting filler expression. It is not maintained, however, by the system.

Note that according to the semantics, no object will match a filler expression ⟨*role*⟩:**someknown**(⟨*concept*⟩) if the extension of ⟨*concept*⟩is empty.

Example:
```
?- d1 :< d, r :< domain(c) and range(d).
yes
?- y1 :: d1, y2 :: d1,
   x :: r : (y3 and y4).
yes
?- x ?: r : someknown(d1).
no
?- y3 :: d1, x ?: r : someknown(d1).
yes
```

Version 4: This operator was called **some** in V4; the change was made to distinguish it from the restriction operator **some** which other terminological systems provide, and which BACK supports as a macro.

Idiosyncrasy: Only allowed in ABox queries. A description containing a **someknown** filler-expression depends on the order in which facts are entered into the knowledge base.

See also: :/2, **allknown**, **theknown**

string — Term

Synopsis: Built-in topmost string.

Syntax: ⟨*conceptual-type*⟩ ::= **string**

Semantics: $[\![\mathtt{string}]\!]^{\mathcal{I}} = S$

Description: **string** is the topmost string having all other strings as instances.

Example: name :< **domain**(person) **and range**(**string**).
hans :: person **and** name : 'Hans Meier'.

Idiosyncrasy: Strings in BACK are arbitrary Prolog atoms and need to be enclosed – according to the Prolog convention – in single-quotes if they contain special characters. **string** is disjoint from **anything** and the other topmost conceptual types **aset** and **number**.

See also: **aset**, **anything**, **number**

subsumes — Ask Expression

Synopsis: Subsumption test.

Syntax: ⟨*ask-expression*⟩ ::= **subsumes**(⟨*term*⟩ ⟨*term*⟩) [**noibox**]

Description: This operator performs a boolean test whether the ⟨*term*⟩ contained in the first argument subsumes the ⟨*term*⟩ contained in the second argument. Actually it is equivalent to **?**<. The answer takes the application of rules into account; if they are to be ignored, the **noibox** option must be used.

Example: **backask**(**subsumes**(**atleast**(12,s),**atleast**(12,r) **and all**(r,d))).
backask(**subsumes**(energy_plant,nuclear_plant)).
backask(**subsumes**(c2,c1) **noibox**).

Idiosyncrasy: Note that **subsumes** differs from **?**< only in the order of the arguments.

See also: **?**<, **equivalent**

the Macro

Synopsis: Uniqueness restriction.

Syntax: ⟨*macro-concept* ⟩ ::= **the**(⟨*role* ⟩,⟨*conceptual-type* ⟩)

Semantics: $[\![\mathbf{the}(\mathsf{r},\mathsf{c})]\!]^{\mathcal{I}} = \{d : |[\![\mathsf{r}]\!]^{\mathcal{I}}(d)| = 1 \wedge [\![\mathsf{r}]\!]^{\mathcal{I}}(d) \subseteq [\![\mathsf{c}]\!]^{\mathcal{I}}\}$

Description: There is exactly one role-filler at role r, and this role-filler is of type c. This operator makes value restrictions for functional roles (features) more readable, since **all**(age,**ge**(18)) easily has the connotation that an object has more than one age.

Example: adult := **the**(age,**ge**(18)).

Idiosyncrasy: The macro **the**(r,c) is internally expanded into **exactly**(1,r) **and** **all**(r,c)

See also: **all**, **some**, **no**

theknown — Object Term

Synopsis: Object-reference by description.

Syntax: ⟨*obj-ref*⟩ ::= **theknown**(concept)$^{\alpha}$

Semantics: $\mathbf{theknown}(\mathsf{c}) \stackrel{\text{def}}{=} \mathsf{o}$ (if o is the only o_i such that $\Gamma \models \mathsf{o}_i :: \mathsf{c}$)

Description: With the **theknown** operator an object can be referred to by the given description. A **theknown** expression is only evaluated once, and is substituted by the identifier of the referred object. It is not maintained, however, by the system. The reference must be unambiguous: if the given concept has more than one instance, or none, the object reference is undefined; this may cause surrounding goals to raise an error, and to fail.

Example:
```
?- d1 :< d, r :< domain(c) and range(d).
yes
?- y1 :: d, y2 :: d1,
   x :: r : (y1 and theknown(d1)).
yes
?- x ?: r : theknown(d).
no
?- theknown(d1) ?: d.
yes
```

Version 4: This operator was called **the** in V4.

Idiosyncrasy: Only allowed in ABox expressions. A description containing a **theknown** filler-expression depends on the order in which facts are entered into the knowledge base.

See also: :/2, **allknown**, **someknown**, **uc(**i**)**

trans — Role Term

Synopsis: Transitive closure of roles.

Syntax: $\langle role \rangle$ ::= **trans**($\langle role \rangle$)$^{\tau}$

Semantics: $[\![\mathtt{trans}(\mathsf{r})]\!]^{\mathcal{I}} = ([\![\mathsf{r}]\!]^{\mathcal{I}})^{+}$

Description: **trans** is a role operator constructing the transitive closure of its argument. The combination of **comp** and **trans** leads to incompleteness in some cases. For transitively closed roles, number restrictions are not inferred from restrictions at their transitive closure.

Example: is_ancestor_of := **trans**(is_parent_of).

Version 4: **trans** is a new operator in version 5.

Idiosyncrasy: The **trans** operator is not allowed in ABox tells and queries.

See also: **comp**, **inv**

type — Tell Expression

Synopsis: Type specification.

Syntax:

⟨*tell-expression*⟩ ::= ⟨*definition*⟩[**type** ⟨*modifier*⟩]
| ⟨*rule*⟩
| ⟨*description*⟩
| ⟨*revision*⟩
| ⟨*declaration*⟩

⟨*modifier*⟩ ::= **concept**
| **role**
| **feature**
| **aset**
| **number**
| **string**

Description: The type constructor specifies the type of an expression as being a concept, an aset, a number, a string, a role, or a feature. This constructor serves several purposes: it can be used to make lists of tell-expressions more readable for the user even if all expressions are unambigous for the system; it can be used to make certain tell-expressions unambigous (e.g., those involving the operator **and** which is used for concepts as well as for roles); and it is used to distinguish features (functional roles) from roles.

Example:

product :< **anything type concept**.
energy :< product **type concept**.
produces :< **domain**(plant) **and range**(product) **type role**.
a := b **and** c **type concept**.
wife :< **domain**(man) **type feature**.

Version 4: In V4 the **type** constructor was not needed, since ambiguities did not arise, and features were not supported.

uc(*i*) — **Object Term**

Synopsis: System generated unique identifier for objects.

Syntax: ⟨*obj-ref*⟩ ::= **uc(*i*)**$^{\alpha}$

Description: A *unique constant* is a unique identifier that the system generates for each object; its external representation is **uc(*i*)** where i is the unique identifier. The user will be confronted with a **uc(*i*)** when system output has to refer to an object for which no other name was provided upon its introduction. The **uc(*i*)** may be used also for referring to an object. Note that when an input file is changed and read again, an object may get assigned a different **uc(*i*)**. The user must not introduce an object with a name **uc(*i*)** when **uc(*i*)** has not already been generated by the system.

Example:
```
?- X :: person and name : peter.
X = uc(13)
?- mary :: has_friend : uc(13).
yes
?- uc(99) :: person and name : jack.
no
```

Version 4: The new notation **uc(*i*)** replaces the uc_i format which is still accepted for compatibility reasons.

Idiosyncrasy: Note, that the unique constant depends on the order of input and the prior state of BACK.

See also: ::/2, **name**, **theknown**

union — Attribute Set Term

Synopsis: Operator for forming the union of two attribute sets.

Syntax: ⟨*aset*⟩ ::= ⟨*aset*⟩ **union** ⟨*aset*⟩

Semantics: $[\![\mathtt{a}_1 \ \mathbf{union} \ \mathtt{a}_2]\!]^{\mathcal{I}} = [\![\mathtt{a}_1]\!]^{\mathcal{I}} \cup [\![\mathtt{a}_2]\!]^{\mathcal{I}}$

Description: The **union** operator for attribute sets corresponds semantically to the (in BACK V5 non-existing) **or** operator for concepts. It denotes the union of two attribute sets.

Example:
```
risky   := aset([high,large,medium], risk).
unrisky := aset([medium,small,null], risk).
risks   := risky union unrisky
```

See also: **aset**, **intersection**, **without**

vr — Retrieval

Synopsis: Retrieving an entity's value restriction for a given role.

Syntax:

⟨*output-function*⟩ ::= ⟨*action*⟩
| **vr**(⟨*role*-NAME⟩)
| **vr**(**inv**(⟨*role*⟩))

Description: The operator **vr** retrieves the value restriction of a BACK entity. For each entity to which it is applied, **vr** adds to the output tuple a *single* concept that represents the conjunction of the entity's known value restrictions for the specified role.

Example:

```
?- c1 :< anything, c2 :< anything,
   x :: all(r,c1) and all(r,c2).
yes
?- backretrieve(R = [self, vr(r)] for x).
R = [[x, conc(1298)]]
yes
?- c3 := c1 and c2.
yes
?- backretrieve(R = [self, vr(r)] for x).
R = [[x, c3]]
yes
```

Idiosyncrasy: **vr** is only applicable to entities that may have roles and fillers, i.e., on concepts and their instances.

The application of **vr** returns a value restriction as has been determined up to that point. It does not initiate the application of any inference services. For objects this means that the returned restriction may be incomplete: The abstraction of value restrictions for closed filler sets is triggered by according concept entries in the taxonomy. It therefore can happen that a queried object posses a more special value restriction than returned by **vr**.

See also: **defined_as**, **describe**, **describe_fully**, **for**, **getall**, **introduced_as**, **nr**, **rf**, **self**

without — Attribute Set Term

Synopsis: Operator for subtracting an attribute set from another.

Syntax: $\langle aset \rangle$ ::= $\langle aset \rangle$ **without** $\langle aset \rangle$

Semantics: $[\![\mathtt{a}_1 \text{ without } \mathtt{a}_2]\!]^{\mathcal{I}} = [\![\mathtt{a}_1]\!]^{\mathcal{I}} \setminus [\![\mathtt{a}_2]\!]^{\mathcal{I}}$

Description: The **without** operator performs the intersection of an attribute set with the set complement of the other one. Note that if a subtraction results in the empty aset it is equivalent to **nothing**.

Example:
```
risk    := attribute_domain([high,large,medium,small,null]).
risky   := aset([high,large,medium], risk).
norisk := risk without (risky union aset([small]))
```

See also: **aset**, **union**, **without**

Bibliography

[Baader and Hollunder, 1992] F. Baader and B. Hollunder. Embedding defaults into terminological knowledge representation formalisms. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR'92)*, San Mateo, 1992. Morgan Kaufmann.

[Brachman and Schmolze, 1985] Ronald J. Brachman and James G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, April 1985.

[Brachman, 1977] Ronald J. Brachman. *A Structural Paradigm for Representing Knowledge*. PhD thesis, Havard University, 1977.

[Brachman, 1979] Ronald J. Brachman. On the epistemological status of semantic networks. In N. V. Findler, editor, *Associative Networks: Representation and Use of Knowledge by Computers*, pages 3–50. Academic Press, New York, N.Y., 1979.

[Donini *et al.*, 1991a] F.M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The Complexity of Concept Languages. In *KR '91*, pages 151–162, 1991.

[Donini *et al.*, 1991b] F.M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. Tractable concept languages. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 458–463, 1991.

[Donini *et al.*, 1992] F.M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Adding epistemic operators to concept languages. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR'92)*, San Mateo, 1992. Morgan Kaufmann.

[Hayes, 1977] Patrick J. Hayes. In defence of logic. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 559–565, Cambridge, Mass., 1977.

[Minsky, 1968] M. Minsky. *Semantic Information Processing*. MIT Press, Cambridge (Mass), 1968.

[Minsky, 1975] Marvin Minsky. A framework for representing knowledge. In P. Winston, editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, New York, N.Y., 1975.

[Nebel and Peltason, 1990] B. Nebel and Ch. Peltason. Terminological Reasoning and Information Management. KIT Report 85, Department of Computer Science, Technische Universität Berlin, Berlin, Germany, October 1990.

[Nebel, 1990] Bernhard Nebel. *Reasoning and Revision in Hybrid Representation Systems*, volume 422 of *Lecture Notes in Artificail Intelligence*. Springer-Verlag, Berlin, Germany, 1990.

[Pearce and Wagner, 1992] D. Pearce and G. Wagner, editors. *Logics in AI, Proceedings of JELIA'92*, Berlin, 1992. Springer.

[Quillian, 1968] M.R. Quillian. *Semantic Memory*, pages 216–270. In Minsky [1968], 1968.

[Rich, 1991] Charles Rich. Special issue on implemented knowledge representation and reasoning systems. *SIGART Bulletin*, 2(3), June 1991.

[Royer and Quantz, 1992] V. Royer and J.J. Quantz. Deriving inference rules for terminological logics. In Pearce and Wagner [1992], pages 84–105.

[Woods, 1975] William A. Woods. What's in a link: Foundations for semantic networks. In D. G. Bobrow and A. M. Collins, editors, *Representation and Understanding: Studies in Cognitive Science*, pages 35–82. Academic Press, New York, N.Y., 1975.

Appendix A

Installation of BACK

BACK V5 is currently implemented under Quintus Prolog for SUN stations running Unix SunOS 4.1.2. Past versions of BACK were designed to be as independent as possible from particular Prolog dialects. Since BACK V5 takes advantage of Quintus Prolog specific libraries, we couldn't maintain this strategy. However, we intend to provide an adaptation to SWI-Prolog (and maybe also for C-Prolog and SICSTUS-Prolog) in the near future. We will see what we can do about it.

1. Ensure that a supported Prolog dialect is installed on your local site.
2. Create a new directory and copy 'BACK.tar.Z' into it.
3. Uncompress the file 'BACK.tar.Z' with the command:

   uncompress BACK.tar.Z
4. Untar afterwards the file 'BACK.tar' with the command:

   tar -xf BACK.tar
5. The directory contains now the following files:

   Documentation & Installation

   Readme.Back51 Contents of files, Installation procedure and remarks.
   Install.Back51 Prolog file for installing BACK.
   WhatIsNew.Back51 Documentation of changes from BACK V5.0.
   Doc.Back50 Syntax card and documentation of major changes from BACK V4 to BACK V5.

   TBox & ABox

   backops.pl Operator definitions.
   dynamic.quintus Declarations of dynamic predicates for Quintus Prolog.
   quintus.library Loader for used Quintus library files.
   tbox1.pl & tbox2.pl TBox and IBox of BACK.
   abox.pl Abox of BACK.
   btl.pl Term language interface.

util.pl Some utilities.

back.patch BACK's patch file (is empty after every major release).

Tests & Examples These files are contained in a subdirectory called 'tests':

- alltests.pl
- cmk.abox
- ibox_tests.pl
- jantest.abox
- jantest.iabox
- jantest.retrieval
- jantest.tbox
- macro_tests.pl
- revision_tests.pl
- role_tests.pl
- value_tests.pl

Syntax Converter Tools for converting from previous syntax versions to BACK V5

README Brief description of the purpose of the following files.

translate.pl Translation tool.

b5tf.pl Definition of translation rules.

v42v5 Translation rules for BACK V4.2 syntax to BACK V5 syntax

mb2v5_1 Translation rules for μBACKsyntax to BACK V5 syntax (Part 1)

mb2v5_2 Translation rules for μBACKsyntax to BACK V5 syntax (Part 2)

6. Now you are ready to install BACK. For this purpose call Quintus Prolog, consult the file 'Install.Back51' and compile BACK with the command:

   :- back.

7. After BACK was compiled, it needs to be initialized with the command:

   :- backinit.

8. Now you should save the current state of the Prolog system into a binary file with the Prolog command

   :- save_program(<FileName>).

   for avoiding that BACK needs to be installed from the scratch every time you need it.

Appendix B

Syntax Overview

In the syntax overview we use the following conventions of the extended Backus-Naur Form (EBNF):

- optional arguments are put into brackets, e.g., [optional];
- when brackets are intended as terminal symbols, they are quoted, e.g., '['list']';
- iteration is indicated by braces, e.g., {,element}*;
- parentheses are always terminal symbols, e.g. **backinit**(tbox);

Some constructs in the syntax overview are only applicable in a restricted way (as described above). We mark this by the following signs:

π argument must be primitive;

τ operator may only be used in TBox tells and queries, i.e. not in terms used to describe objects;

? operator may only be used in ABox queries, i.e. in object ?: concept or in the **getall** part of retrievals;

α operator may only be used in ABox expressions, i.e. in terms used to describe objects;

! operator may only be used in tells;

$\top$ operator may only be used at the top-most level.

Interaction

⟨*interaction*⟩ ::= **backinit**[(⟨*box*⟩)]
| **backtell**(⟨*tell-expression*⟩)
| **backask**(⟨*ask-expression*⟩[**noibox**])
| **backstate**[(⟨*state*⟩)]
| **backretrieve**(⟨*retrieval*⟩[**noibox**])
| **backmacro**(⟨*macro-definition*⟩)
| **backread**(⟨*file*-NAME⟩)
| **backload**(⟨*file*-NAME⟩)
| **backwrite**(⟨*file*-NAME⟩[,⟨*box*⟩])
| **backdump**[(⟨*file*-NAME⟩)]

⟨*retrieval*⟩ ::= [PROLOG-VAR =] [⟨*generator*⟩] ⟨*arguments*⟩
| **difference**(⟨*entity*⟩,⟨*entity*⟩)

⟨*generator*⟩ ::= ⟨*action*⟩
| '[⟨*output-function*⟩{,⟨*output-function*⟩}*']' **for**

⟨*action*⟩ ::= **describe**
| **describe_fully**
| **defined_as**
| **introduced_as**
| **self**
| **msc**

⟨*output-function*⟩ ::= ⟨*action*⟩
| **vr**(⟨*role*-NAME⟩)
| **vr**(**inv**(⟨*role*⟩))
| **nr**(⟨*role*-NAME⟩)
| **nr**(**inv**(⟨*role*⟩))
| **rf**(⟨*role*-NAME⟩)
| **rf**(**inv**(⟨*role*⟩))

⟨*arguments*⟩ ::= ⟨*arg-spec*⟩[/⟨*disambig*⟩]
| **getall**(⟨*concept*⟩)
| **getall**(⟨*aset*⟩)
| **getall**(**string**)

⟨*arg-spec*⟩ ::= ⟨*entity*⟩
| '['⟨*entity*⟩[/⟨*disambig*⟩] {,⟨*entity*⟩[/⟨*disambig*⟩]}*']'

⟨*disambig*⟩ ::= **conc**
| **obj**
| ⟨*domain*-NAME⟩**^cls**
| ⟨*domain*-NAME⟩**^obj**

⟨*macro-definition*⟩ ::= ⟨*macro*⟩ *∗*= ⟨*term*⟩
⟨*macro*⟩ ::= ⟨*macro*-NAME⟩[(PROLOG-VAR{,PROLOG-VAR}*)]

⟨*state*⟩ ::= **verbosity** = **silent**
| **verbosity** = **error**
| **verbosity** = **warning**
| **verbosity** = **info**
| **verbosity** = **trace**
| **introduction** = **forward**
| **introduction** = **noforward**
| **revision** = **true**
| **revision** = **false**
| **retrieval** = **fail**
| **retrieval** = **succeed**
| **tboxrevision** = **fail**
| **tboxrevision** = **succeed**
| **aboxfilled** = **false**
| **aboxfilled** = **true**
| **aboxfilled** = **abox**
| **iboxfilled** = **false**
| **iboxfilled** = **true**

⟨*box*⟩ ::= **tbox**
| **ibox**
| **abox**

Tell/Ask Expressions

⟨*tell-expression*⟩ ::= ⟨*definition*⟩[**type** ⟨*modifier*⟩]
| ⟨*rule*⟩
| ⟨*description*⟩
| ⟨*revision*⟩
| ⟨*declaration*⟩

⟨*definition*⟩ ::= ⟨*term*-NAME⟩ := ⟨*term*⟩
| ⟨*concept*-NAME⟩ :< ⟨*concept*⟩
| ⟨*role*-NAME⟩ :< ⟨*role*⟩

⟨*rule*⟩ ::= ⟨*concept*⟩ => ⟨*concept*⟩

⟨*description*⟩ ::= ⟨*obj-ref*⟩ :: ⟨*concept*⟩
| PROLOG-VAR :: ⟨*concept*⟩

⟨*revision*⟩ ::= **forget**(⟨*rule*⟩)
| **forget**(⟨*obj-ref*⟩ :: ⟨*concept*⟩)
| **forget**(⟨*obj-ref*⟩)
| **redescribe**(⟨*obj-ref*⟩ :: ⟨*concept*⟩)
| **name**(⟨*obj-ref*⟩,⟨*object*-NAME⟩)

⟨*declaration*⟩ ::= ⟨*domain*-NAME⟩ := **attribute_domain**
| ⟨*domain*-NAME⟩ := **attribute_domain**(⟨*attribute-list*⟩)

⟨*modifier*⟩ ::= **concept**
| **role**
| **feature**
| **aset**
| **number**
| **string**

⟨*ask-expression*⟩ ::= ⟨*term*⟩ ?< ⟨*term*⟩
| ⟨*obj-ref*⟩ ?: ⟨*concept*⟩
| **disjoint**(⟨*term*⟩,⟨*term*⟩)
| **subsumes**(⟨*term*⟩ ⟨*term*⟩)
| **equivalent**(⟨*term*⟩,⟨*term*⟩)
| **incoherent**(⟨*term*⟩)

Terms

⟨*entity*⟩ ::= ⟨*term*⟩
| ⟨*value*⟩

⟨*term*⟩ ::= ⟨*conceptual-type*⟩
| ⟨*role*⟩
| ⟨*macro*⟩

⟨*conceptual-type*⟩ ::= ⟨*concept*⟩
| ⟨*aset*⟩
| ⟨*number*⟩
| **string**

⟨*value*⟩ ::= ⟨*obj-ref*⟩
| ⟨*attribute*-NAME⟩
| ⟨*number*-INSTANCE⟩
| ⟨*string*-INSTANCE⟩

⟨*obj-ref*⟩ ::= ⟨*object*-NAME⟩
| **uc**(⟨INTEGER⟩)$^{\alpha}$
| **theknown**(concept)$^{\alpha}$

Concept Terms

⟨*concept*⟩ ::= ⟨*concept*-NAME⟩
| **anything**
| **nothing**
| ⟨*concept*⟩ **and** ⟨*concept*⟩
| ⟨*concept*⟩ **or** ⟨*concept*⟩$^{?\top}$
| **not**(⟨*concept*⟩)$^{\pi}$
| **all**(⟨*role*⟩,⟨*conceptual-type*⟩)
| **atleast**(⟨INTEGER⟩,⟨*role*⟩)
| **atmost**(⟨INTEGER⟩,⟨*role*⟩)
| **oneof**(‘[’⟨*object*-NAME⟩{,⟨*object*-NAME⟩}*‘]’)
| ⟨*role*⟩:⟨*filler-expr*⟩

Role Terms

⟨*role*⟩ ::= ⟨*role*-NAME⟩
| ⟨*role*⟩ **and** ⟨*role*⟩
| **not**(⟨*role*⟩)$^{\pi}$
| **domain**(⟨*concept*⟩)
| **range**(⟨*conceptual-type*⟩)
| **inv**(⟨*role*⟩)
| ⟨*role*⟩ **comp** ⟨*role*⟩$^{\tau}$
| **trans**(⟨*role*⟩)$^{\tau}$

Attribute Set Terms

⟨*aset*⟩ ::= **aset**
| ⟨*aset*-NAME⟩
| ⟨*aset*⟩ **union** ⟨*aset*⟩
| (⟨*aset*⟩ **intersection** ⟨*aset*⟩)
| ⟨*aset*⟩ **without** ⟨*aset*⟩
| **aset**(‘[’⟨*attribute-list*⟩‘]’)
| **aset**(⟨*attribute-spec*⟩,⟨*domain*-NAME⟩)

⟨*attribute-spec*⟩ ::= ‘[’⟨*attribute-list*⟩‘]’
| ⟨*attribute*-NAME⟩ .. ⟨*attribute*-NAME⟩

⟨*attribute-list*⟩ ::= ⟨*attribute*-NAME⟩{,⟨*attribute*-NAME⟩}*

Number Terms

⟨*number*⟩ ::= **number**
| ⟨*number*-NAME⟩
| ⟨*number-range*⟩
| (⟨*number*⟩ **intersection** ⟨*number*⟩)
| ⟨*number*-INSTANCE⟩

⟨*number-range*⟩ ::= ⟨*lower-limit*⟩
| ⟨*upper-limit*⟩
| ⟨*number*-INSTANCE⟩.. ⟨*number*-INSTANCE⟩

⟨*lower-limit*⟩ ::= **gt**(⟨*number*-INSTANCE⟩)
| **ge**(⟨*number*-INSTANCE⟩)

⟨*upper-limit*⟩ ::= **lt**(⟨*number*-INSTANCE⟩)
| **le**(⟨*number*-INSTANCE⟩)

Filler Expressions

⟨*filler-expr*⟩ ::= ⟨*value*⟩
| (⟨*description*⟩)$^{!\alpha}$
| **close**(⟨*filler-expr*⟩)
| **someknown**(⟨*concept*⟩)$^{?\alpha}$
| **allknown**(⟨*concept*⟩)$^{\alpha}$
| ⟨*value*⟩ **and** ⟨*filler-expr*⟩
| ⟨*value*⟩ **or** ⟨*filler-expr*⟩$^{?\alpha}$
| (⟨*filler-expr*⟩)

Macro Library

⟨*macro-concept*⟩ ::= **some**(⟨*role*⟩,⟨*conceptual-type*⟩)
| **some**(⟨*role*⟩)
| **the**(⟨*role*⟩,⟨*conceptual-type*⟩)
| **no**(⟨*role*⟩,⟨*conceptual-type*⟩)
| **no**(⟨*role*⟩)
| **exactly**(⟨INTEGER⟩,⟨*role*⟩)
| **atleast**(⟨INTEGER⟩,⟨*role*⟩,⟨*conceptual-type*⟩)
| **atmost**(⟨INTEGER⟩,⟨*role*⟩,⟨*conceptual-type*⟩)
| **exactly**(⟨INTEGER⟩,⟨*role*⟩,⟨*conceptual-type*⟩)
| **rvm_some**(⟨*role*⟩,⟨*role*⟩)
| **rvm_no**(⟨*role*⟩,⟨*role*⟩)

Appendix C

Formal Semantics Overview

We begin by giving a model-theoretic semantics for the terminological logic underlying BACK V5. To do so, we first summarize the syntax of concepts, roles, and formulae. We use t for terms in general, i.e. concepts or roles, the index n for names, e.g. c_n, and the index p for primitive components, e.g. c_p:

$$
\begin{array}{rcl}
\mathsf{c} & ::= & \textbf{anything} \mid \textbf{nothing} \mid \mathsf{c}_p \mid \mathsf{c}_n \mid \textbf{not}(\mathsf{c}_p) \mid \mathsf{c}_1 \textbf{ and } \mathsf{c}_2 \mid \mathsf{r}{:}\mathsf{o} \\
 & \mid & \textbf{all}(\mathsf{r},\mathsf{c}_1) \mid \textbf{atleast}(\mathrm{n},\mathsf{r},\mathsf{c}_1) \mid \textbf{atmost}(\mathrm{n},\mathsf{r},\mathsf{c}_1) \\
\mathsf{r} & ::= & \textbf{nothing} \mid \mathsf{r}_p \mid \mathsf{r}_n \mid \textbf{not}(\mathsf{r}_p) \mid \mathsf{r}_1 \textbf{ and } \mathsf{r}_2 \mid \textbf{inv}(\mathsf{r}_1) \mid \mathsf{r}_1 \textbf{ comp } \mathsf{r}_2 \\
 & \mid & \textbf{domain}(\mathsf{c}) \mid \textbf{range}(\mathsf{c}) \\
\gamma & ::= & \mathsf{t}_1 \sqsubseteq \mathsf{t}_2 \mid \mathsf{o} :: \mathsf{c}
\end{array}
$$

We assume the usual model-theoretic semantics where a model $\mathcal{M}$ of a set of TL-formulae Γ is a pair $\langle D, \mathcal{I} \rangle$. The interpretation function $[\![\cdot]\!]^{\mathcal{I}}$ maps concepts into subsets of the domain D, roles into subsets of $D \times D$, and object-names injectively into D, respecting the following equations (we use $r(d)$ to denote $\{e : \langle d, e \rangle \in r\}$):

$$
\begin{aligned}
[\![\mathbf{anything}]\!]^{\mathcal{I}} &= D && \text{(C.1)} \\
[\![\mathbf{nothing}]\!]^{\mathcal{I}} &= \emptyset && \text{(C.2)} \\
[\![\mathbf{not}(\mathsf{t}_p)]\!]^{\mathcal{I}} &= D \setminus [\![\mathsf{t}_p]\!]^{\mathcal{I}} && \text{(C.3)} \\
[\![\mathsf{t}_1 \textbf{ and } \mathsf{t}_2]\!]^{\mathcal{I}} &= [\![\mathsf{t}_1]\!]^{\mathcal{I}} \cap [\![\mathsf{t}_2]\!]^{\mathcal{I}} && \text{(C.4)} \\
[\![\mathbf{all}(\mathsf{r},\mathsf{c})]\!]^{\mathcal{I}} &= \{d : [\![\mathsf{r}]\!]^{\mathcal{I}}(d) \subseteq [\![\mathsf{c}]\!]^{\mathcal{I}}\} && \text{(C.5)} \\
[\![\mathbf{atleast}(n,\mathsf{r},\mathsf{c})]\!]^{\mathcal{I}} &= \{d : |[\![\mathsf{r}]\!]^{\mathcal{I}}(d) \cap [\![\mathsf{c}]\!]^{\mathcal{I}}| \geq n\} && \text{(C.6)} \\
[\![\mathbf{atmost}(n,\mathsf{r},\mathsf{c})]\!]^{\mathcal{I}} &= \{d : |[\![\mathsf{r}]\!]^{\mathcal{I}}(d) \cap [\![\mathsf{c}]\!]^{\mathcal{I}}| \leq n\} && \text{(C.7)} \\
[\![\mathsf{r} : \mathsf{o}]\!]^{\mathcal{I}} &= \{d : [\![\mathsf{o}]\!]^{\mathcal{I}} \in [\![\mathsf{r}]\!]^{\mathcal{I}}(d)\} && \text{(C.8)} \\
[\![\mathbf{inv}(\mathsf{r}_1)]\!]^{\mathcal{I}} &= \{\langle d, e \rangle : \langle e, d \rangle \in [\![\mathsf{r}]\!]^{\mathcal{I}}\} && \text{(C.9)} \\
[\![\mathsf{r}_1 \textbf{ comp } \mathsf{r}_2]\!]^{\mathcal{I}} &= [\![\mathsf{r}_1]\!]^{\mathcal{I}} \circ [\![\mathsf{r}_2]\!]^{\mathcal{I}} && \text{(C.10)} \\
[\![\mathbf{domain}(\mathsf{c})]\!]^{\mathcal{I}} &= [\![\mathsf{c}]\!]^{\mathcal{I}} \times D && \text{(C.11)} \\
[\![\mathbf{range}(\mathsf{c})]\!]^{\mathcal{I}} &= D \times [\![\mathsf{c}]\!]^{\mathcal{I}} && \text{(C.12)}
\end{aligned}
$$

Satisfaction of formulae is then defined as follows:

$$
\begin{aligned}
\mathcal{M} \models \mathsf{t}_1 \preceq \mathsf{t}_2 \quad &\text{iff} \quad [\![\mathsf{t}_2]\!]^{\mathcal{I}} \sqsubseteq [\![\mathsf{t}_1]\!]^{\mathcal{I}} && \text{(C.13)} \\
\mathcal{M} \models \mathsf{o} :: \mathsf{c} \quad &\text{iff} \quad [\![\mathsf{o}]\!]^{\mathcal{I}} \in [\![\mathsf{c}]\!]^{\mathcal{I}} && \text{(C.14)}
\end{aligned}
$$

A structure $\mathcal{M}$ is a model of a formula γ iff $\mathcal{M} \models \gamma$; it is a model of a set of formulae Γ iff it is a model of every formula in Γ. A formula γ is *entailed* by a set of formulae Γ (written $\Gamma \models \gamma$) iff every structure which is a model of Γ is also a model of γ. A set of TL-formulae Γ is *satisfiable* iff it has a model, otherwise it is *inconsistent*.

The attentive reader may have noticed some discrepancies between the syntax of BACK V5 and the syntax of the terminologic logic given above. For one thing, definitions in BACK V5 have the form $\mathsf{t}_n :< \mathsf{t}$ or $\mathsf{t}_n := \mathsf{t}$. These definitions correspond to the (sets of) formulae $\mathsf{t}_n \sqsubseteq \mathsf{t}$ and $\{\mathsf{t}_n \sqsubseteq \mathsf{t}, \mathsf{t} \sqsubseteq \mathsf{t}_n\}$ respectively. Furthermore, a rule $\mathsf{c}_1 => \mathsf{c}_2$ corresponds to a formula $\mathsf{c}_1 \sqsubseteq \mathsf{c}_2$.

Thus a list of **backtell**s can be seen as a set of TL-formulae Γ. The semantics of a **backask** then is reducible to the entailment of a formula by Γ. There are two peculiarities, however. First, we do not take into account ABox descriptions when computing term subsumption, second we allow the **noibox** operator to ignore rules. If we thus have a set of TL-formulae Γ, consisting of a set of definitions Θ, a set of rules $\mathcal{R}$ and a set of descriptions $\mathcal{A}$ we define the semantics of **backask** as follows:

$$\mathbf{backask}(\mathsf{t}_1 \,?< \mathsf{t}_2) \quad \text{iff} \quad \Theta \cup \mathcal{R} \models \mathsf{t}_1 \sqsubseteq \mathsf{t}_2 \tag{C.15}$$

$$\mathbf{backask}(\mathsf{t}_1 \,?< \mathsf{t}_2)\ \mathbf{noibox} \quad \text{iff} \quad \Theta \models \mathsf{t}_1 \sqsubseteq \mathsf{t}_2 \tag{C.16}$$

$$\mathbf{backask}(\mathsf{o} \,?: \mathsf{c}) \quad \text{iff} \quad \Gamma \models \mathsf{o} :: \mathsf{c} \tag{C.17}$$

$$\mathbf{backask}(\mathsf{o} \,?: \mathsf{c})\ \mathbf{noibox} \quad \text{iff} \quad \Theta \cup \mathcal{A} \models \mathsf{o} :: \mathsf{c} \tag{C.18}$$

Another semantic issue concerns the completeness of rules. Given the semantics above, rules are like material implications and thus contraposition and reasoning by case should be possible. However, these latter inferences are not implemented in BACK V5, and thus, this is a source of incompleteness of the current version. The alternative would be to specify a semantics which treats rules as triggered forward-chaining rules (see [Baader and Hollunder, 1992] and [Donini *et al.*, 1992]). Given such a semantics BACK V5 would be complete with respect to rules.

In the semantics above, interpretation functions map roles into subsets of $D \times D$. BACK V5 supports special kinds of roles, namely *features*, whose interpretation has to be functional. Thus, a model of a set of TL-formulae has to fulfill some additional requirements:

$$\forall d, d', d'' \in D[\langle d, d' \rangle \in [\![\mathsf{r}]\!]^{\mathcal{I}} \wedge \langle d, d'' \rangle \in [\![\mathsf{r}]\!]^{\mathcal{I}} \rightarrow d' = d''] \tag{C.19}$$

if r is a feature.

Besides "ordinary" concepts BACK V5 also supports special concepts, namely *asets*, *numbers*, and *strings*. To capture this formally we would have to extend the domain D by additional sets standing for the interpretations of these conceptual types. The interpretation of *number* would be the set of real numbers. We do not give the formal details here, since we think that the meaning of the term-forming operators provided for asets and numbers are self-explanatory.

We now give a semantics for the **getall** part of **backretrieve**. In general this functionality returns the instances of a concept:

$$\mathsf{o} \in \mathbf{getall}(\mathsf{c}) \quad \text{iff} \quad \Gamma \models \mathsf{o} :: \mathsf{c} \tag{C.20}$$

Note that the use of the **noibox** operator leads to taking $\Theta \cup \mathcal{A}$ instead of Γ, i.e. the set of rules $\mathcal{R}$ is ignored.

Finally a word concerning *filler-expressions*, i.e. expressions that can be used in connection with the fills construct r:o. Instead of specifying a single object as a filler of role r, the user can specify a filler-expression, which mostly are interpreted as macros. Some of these expressions may only be used in ABox descriptions or only in queries (see the syntax). The operators **and** and **or** can be used to combine filler-expressions. Note that disjunctions are only allowed in queries. The operators **someknown** and **allknown** are interpreted as macros, disjoining and conjoining all the instances of a concept currently known in the knowledge base:

$$\mathbf{someknown}(\mathsf{c}) \stackrel{\text{def}}{=} \mathbf{or}\{\mathsf{o} : \Gamma \models \mathsf{o} :: \mathsf{c}\} \tag{C.21}$$

$$\mathbf{allknown}(\mathsf{c}) \stackrel{\text{def}}{=} \mathbf{and}\{\mathsf{o} : \Gamma \models \mathsf{o} :: \mathsf{c}\} \tag{C.22}$$

Note that we use **and**$\{\mathsf{o}_1, ..., \mathsf{o}_n\}$ for the term o_1 **and** ... **and** o_n. Note further that these operators are treated as macros and *not* as semantic descriptions: thus when evaluating a term containing these macros they are expanded, i.e. substituted by the disjunction or conjunction of the *currently* known instances of c. Therefore, the use of **someknown** and **allknown** is order-dependent, whereas the other constructs of BACK V5 are order-independent.

The attentive reader will have noticed a problem with our semantics for **someknown** and **allknown**: what if there are currently no instances of c? In that case it is not possible to expand the macro **someknown**(c). We can, however, expand the whole concept term in which the filler-expression occurs:

$$\mathsf{r} : \mathbf{someknown}(\mathsf{c}) \stackrel{\text{def}}{=} \mathbf{nothing} \quad \text{if } \Gamma \models \mathsf{c} \sqsubseteq \mathbf{nothing} \tag{C.23}$$

$$\mathsf{r} : \mathbf{allknown}(\mathsf{c}) \stackrel{\text{def}}{=} \mathbf{anything} \quad \text{if } \Gamma \models \mathsf{c} \sqsubseteq \mathbf{nothing} \tag{C.24}$$

The operator **theknown** can be used to refer to an object by using a description:

$$\mathbf{theknown}(\mathsf{c}) \stackrel{\text{def}}{=} \mathsf{o} \quad \text{if o is the only } \mathsf{o}_i \text{ such that } \Gamma \models \mathsf{o}_i :: \mathsf{c} \tag{C.25}$$

Note again that **theknown** is syntactically expanded as a macro and is thus order-dependent.

Finally, a word concerning the **close** operator. It is also treated as a macro and adds an **atmost**(n,r) restriction.

$$\mathsf{r} : \mathbf{close}(fe) \stackrel{\text{def}}{=} \mathsf{r} : fe \ \mathbf{and}\ \mathbf{atmost}(n, \mathsf{r}) \quad \text{where } n \text{ is the cardinality of the filler-expression} \tag{C.26}$$

Thus the **close** is local and takes into account only the role-fillers specified in the filler-expression and not the ones currently known. It is therefore order-independent.

Appendix D

Programming Interface

The programming interface (PIF) provides a "low-level" access to the BACK-system; thus, circumventing the parser of BACK. Since the actual PIF is restricted in certain ways and differs from the PIF of BACK V4, the following remarks are necessary:

- In contrast to BACK's user interface – PIF operates solely on names. All expressions of PIF use names of the kind indicated in the description below as arguments. The only exceptions are Min and Max which use a number, Type which uses the type of an entity, and Source.
- None of the arguments needs to be instantiated. Backtracking is performed over all known names of the appropiate type.
- There are no error messages. Calls with unknown names or ill-formed calls will simply fail.
- The 'kind' expression can be used to determine the type and the source of a name. While the returned value for source consists either of the constant 'user', 'predef' or 'anonym', the type may be bound to 'conc' resp. 'role/X', where X denotes the type of the role's range. In case a non-user defined name is found, the backretrieve actions can be used to generate an equivalent description of the name.
- Incoherent entities can only be dealt with by predicates containing the prefix 'incoherent_' in their name. All other predicates, except for 'super_prim_concept', 'primitive_concept' and 'defined_concept', will always fail on incoherent concepts, roles, asets, or numbers. In addition, they will never generate incoherent entities upon backtracking.
- Since PIF provides functionality for determining the "supers" and "direct supers" of an entity, the following names are used for denoting the "top" elements:

  anything for concepts

  aset for asets

  number for numbers

  string for strings

The determined supers of an entity will never include names of equivalent entities. The predicate 'role' can be used to retrieve the names of all existing roles, since the top role 'anyrole' is no longer available.

- Predicates containing the prefix 'equivalent_' in their name and predicates for determining the different types of "supers" of an entity will never succeed with two identical names. Thus, they also will not return an entity as its own super or its own equivalent.

- Additionally predicates containing the substring 'super' in their name will not return equivalent entities.

- The uninstantiated predicate 'disjoint' generates *all* disjoint entities and not only disjoint entities of one domain. Thus, it is better to instantiate at least one argument.

The predicates 'synonym' and 'individual_concept' are no longer available, because of the change of the syntax and functionality from BACK V4 to BACK V5. Since the functionality of BACK V5 now incorporates 'defined roles', PIF was extended by the predicates 'role', 'primitive_role', 'defined_role, and 'super_prim_role'.

tboxget(⟨*tbox-get-expr*⟩)

⟨*tbox-get-expr*⟩ ::= **primitive**(⟨*term-name*⟩)
| **defined**(⟨*term-name*⟩)
| **super**(⟨*term-name*⟩, ⟨*term-name*⟩)
| **direct_super**(⟨*term-name*⟩, ⟨*term-name*⟩)
| **equivalent**(⟨*term-name*⟩, ⟨*term-name*⟩)
| **disjoint**(⟨*term-name*⟩, ⟨*term-name*⟩)
| **incoherent**(⟨*term-name*⟩)
| **kind**(⟨*term-name*⟩, ⟨*type*⟩, ⟨*source*⟩)

| **primitive_concept**(⟨*concept-name*⟩)
| **defined_concept**(⟨*concept-name*⟩)
| **super_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **super_prim_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **direct_super_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **user_direct_super_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **equivalent_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **disjoint_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **user_disjoint_prim_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **incoherent_concept**(⟨*concept-name*⟩)
| **value_restriction**(⟨*concept-name*⟩, ⟨*role-name*⟩, ⟨*any-name*⟩)
| **user_value_restriction**(⟨*concept-name*⟩, ⟨*role-name*⟩, ⟨*any-name*⟩)
| **number_restriction**(⟨*concept-name*⟩, ⟨*role-name*⟩, ⟨*min*⟩, ⟨*max*⟩)
| **user_number_restriction**(⟨*concept-name*⟩, ⟨*role-name*⟩, ⟨*min*⟩, ⟨*max*⟩)
| **restriction**(⟨*concept-name*⟩, ⟨*role-name*⟩, ⟨*any-name*⟩, ⟨*min*⟩, ⟨*max*⟩)
| **user_restriction**(⟨*concept-name*⟩, ⟨*role-name*⟩, ⟨*any-name*⟩, ⟨*min*⟩, ⟨*max*⟩)

| **role**(⟨*role-name*⟩)
| **primitive_role**(⟨*role-name*⟩)
| **defined_role**(⟨*role-name*⟩)
| **super_role**(⟨*role-name*⟩, ⟨*role-name*⟩)
| **super_prim_role**(⟨*role-name*⟩, ⟨*role-name*⟩)
| **direct_super_role**(⟨*role-name*⟩, ⟨*role-name*⟩)
| **equivalent_role**(⟨*role-name*⟩, ⟨*role-name*⟩)
| **disjoint_role**(⟨*role-name*⟩, ⟨*role-name*⟩)
| **incoherent_role**(⟨*role-name*⟩)
| **range**(⟨*role-name*⟩, ⟨*any-name*⟩)
| **user_range**(⟨*role-name*⟩, ⟨*any-name*⟩)
| **domain**(⟨*role-name*⟩, ⟨*concept-name*⟩)
| **user_domain**(⟨*role-name*⟩, ⟨*concept-name*⟩)

| **super_aset**(⟨*aset-name*⟩, ⟨*aset-name*⟩)
| **equivalent_aset**(⟨*aset-name*⟩, ⟨*aset-name*⟩)
| **disjoint_aset**(⟨*aset-name*⟩, ⟨*aset-name*⟩)
| **incoherent_aset**(⟨*aset-name*⟩)
| **element**(⟨*attribute-name*⟩, ⟨*aset-name*⟩)
| **attribute_domain**(⟨*aset-name*⟩, ⟨*attribute-domain-name*⟩)

| **super_number**(⟨*number-name*⟩, ⟨*number-name*⟩)
| **equivalent_number**(⟨*number-name*⟩, ⟨*number-name*⟩)
| **disjoint_number**(⟨*number-name*⟩, ⟨*number-name*⟩)
| **incoherent_number**(⟨*number-name*⟩)

iboxget(⟨*ibox-get-expr*⟩)

⟨*ibox-get-expr*⟩ ::= **super_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **direct_super_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **user_direct_super_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **equivalent_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **disjoint_concept**(⟨*concept-name*⟩, ⟨*concept-name*⟩)
| **incoherent_concept**(⟨*concept-name*⟩)
| **value_restriction**(⟨*concept-name*⟩, ⟨*role-name*⟩, ⟨*any-name*⟩)
| **number_restriction**(⟨*concept-name*⟩, ⟨*role-name*⟩, ⟨*min*⟩, ⟨*max*⟩)
| **restriction**(⟨*concept-name*⟩, ⟨*role-name*⟩, ⟨*any-name*⟩, ⟨*min*⟩, ⟨*max*⟩)

| **equivalent_role**(⟨*role-name*⟩, ⟨*role-name*⟩)
| **disjoint_role**(⟨*role-name*⟩, ⟨*role-name*⟩)
| **incoherent_role**(⟨*role-name*⟩)