XEROX

HUMBLE
Reference Manual

Xerox Special Information Systems

HUMBLE™ V2.0 Reference Manual

Kurt W. Piersol

August 1987

Xerox Special Information Systems
Vista Laboratory
250 North Halstead Street
P.O. Box 5608
Pasadena, CA 91107-0608
(818) 351-2351

8/87

1. Introduction
    1.1. What is HUMBLE? 1
    1.2. Background Reading 2
    1.3. Examples 2
2. Knowledge Bases 3
3. HUMBLE Entities 4
    3.1. The Structure of Entities 4
        3.1.1. Entity Type Definitions 7
            3.1.1.1. The Type Above Specification 8
            3.1.1.2. The Prompts 8
            3.1.1.3. The Parameter List 9
            3.1.1.4. The Main Parameter List 9
    3.2. Parameters 10
        3.2.1. The Uncertainty System 11
        3.2.2. Parameter Definitions 12
            3.2.2.1. Describes 12
            3.2.2.2. Parameter Types 12
                3.2.2.2.1. YN Parameters 13
                3.2.2.2.2. MV Parameters 13
                3.2.2.2.3. Number Parameters 13
                3.2.2.2.4. String Parameters 14
                3.2.2.2.5. Enumerated Parameters 14
            3.2.2.3. Prompting 15
                3.2.2.3.1. The Text of the Prompt 15
                3.2.2.3.2. The Prompt Flag 16
            3.2.2.4. Explanation 17
            3.2.2.5. Remarks 17
            3.2.2.6. Altering the Order in which Deducing Rules are Fired 17
            3.2.2.7. The Change Block 18
    3.3. The Scope of Parameters 19
4. HUMBLE Rules 21
    4.1. Introduction 21
    4.2. Basics 21
        4.2.1. HUMBLE Rules and Smalltalk – 80 21
        4.2.2. The Structure of a HUMBLE Rule 22
        4.2.3. Rule Statements 23
            4.2.3.1. If – Then – Else Statements 23
                4.2.3.1.1. The Premise 23
                4.2.3.1.2. The Action 24
                    4.2.3.1.2.1. Making Conclusions 24
                    4.2.3.1.2.2. Evaluating Other Statements 25
                    4.2.3.1.2.3. Firing Other Rules 26
                    4.2.3.1.2.4. Setting New Goals 27
            4.2.3.2. If – Any, If – All, and If – None Statements 27

4.3. Rule Execution 29
4.4. Syntax BNF 30
4.4.1. BNF Conventions 30
4.4.2. Syntax BNF 31
4.4.3. Special Boolean Operations 33
4.4.4. Smalltalk Escape Sequences 35
4.5. Example Rules 36
4.5.1. Typical Rules 36
4.5.2. Searching Rules 37
4.5.3. Nested Rules 38
4.6. Error Messages During Compilation 38
5 The HUMBLE Manager 42
5.1. Introduction 42
5.2. Adding and Removing Knowledge Bases 43
5.3. Editing Knowledge Bases 43
5.4. Consulting Knowledge Bases 43
5.4.1. Controlling Output During a Consultation 44
5.4.2. Tracing Execution During a Consultation 45
5.5. Saving Knowledge Bases 45
5.6. Listing Knowledge Bases 45
6. The HUMBLE Knowledge Base Editor 47
6.1. Window Description 47
6.1.1. The Entity Type List 47
6.1.2. The Parameter Definition List 48
6.1.3. The Rule List 49
6.1.4. The Editor Pane 50
6.1.4.1. Basic Editing 50
6.1.4.2. Rules - - > 50
6.1.4.3. Actions - - > 52
7. The HUMBLE Graphic Utility 53
7.1. The Graphic Display 53
7.2. Editing Elements 53
7.3. Testing your Knowledge Base, using Find/Execute 53
8. The HUMBLE Listener Window 54
8.1. Introduction 55
8.2. Beginning a Consultation 55
8.3. Explaining Results 55
8.4. Explaining Text 56
8.5. Examining Alternative Conclusions 56
8.6. Examining Entities 57
9. Example and Tutorial: Building a HUMBLE Knowledge Base 58
9.1. The Problem 58
9.2 The Entities and Parameters 58
9.3 The Rules 61
9.4 A Sample Consultation 64

9.5. A Refinement 65
10. The HUMBLE Programmer's Interface 67
10.1. Introduction 67
10.2. Required Reading 67
10.3. The KnowledgeBases Global 68
10.4. Individual Knowledge Bases 69
10.4.1. Executing the Rules 70
10.4.1.1. Backward Chaining 70
10.4.1.2. Forward Chaining 71
10.4.2. Inspecting the Entities 71
10.4.2.1. Examining Entities 72
10.4.2.1.1. Sub – Entities 72
10.4.2.1.2. Accessing the Parameters 73
10.4.2.1.3. Accessing the Entity Type 74
10.4.2.1.4. Other Useful Information 74
10.4.2.2. Examining Parameters 75
10.4.2.2.1. Accessing the Hypotheses 75
10.4.2.2.2. Accessing the Parameter Definition 77
10.4.2.3. Examining Hypotheses 78
10.4.3. Resetting a Knowledge Base 78
10.4.4. Explaining Conclusions 79
10.4.5. Examining Alternatives 79
10.4.6. Redirecting Output 80
10.4.7. Redirecting Input 80
10.4.8. Tracing Execution 81
10.4.9. Initializing the Entities 81
10.5. Altering a Certainty Model 83
10.5.1. Combination Messages 83
10.5.2. Certainty Constants 84
11. Using HUMBLE in Popular Expert Systems Architectures 85
11.1. Simulations 86
11.1.1. MazeMaster: a Humble Simulation Tutorial 87
11.1.2. The MazeMaster Object Classes 87
11.1.3. The MazeMaster Connection 88
11.1.4. What We Were Trying to Say 89
11.2. Blackboard Systems 89
11.2.1. The Blackboard Concept 89
11.2.2. HUMBLE Knowledge Bases as Knowledge Sources 90
11.3. Smalltalk – 80 and Frame Based Systems 91
11.3.1. The Frame Concept 91
11.3.2. Humble Knowledge Bases within Frames 92

1. Introduction

1.1. What is HUMBLE?

Before delving into the specifics of creating an expert system using HUMBLE, it might be wise to at least get a starting set of vocabulary in place. A HUMBLE knowledge base is a set of *rules* about some set of *entities* which you have defined. The entities define what sort of things, or ideas you will be making logical conclusions about, while the rules define what sorts of conclusions are permissible.

A knowledge base can be anything from a set of ten rules about where you like to go to lunch to a set of a thousand rules about how to decide whether a patient has anthrax or cholera. Rules can be written on just about any subject it is possible to think logically about.

One of the features common to most expert systems is an ability to reason in a logical manner about data which is uncertain. HUMBLE supports this feature too. Data in a HUMBLE knowledge base (this part of a knowledge base is often called the fact base), as well as rules, can have *certainty factors* attached to them, to indicate that the data or rule may or may not be absolutely correct. These certainty factors allow HUMBLE to be applied to data in real world situations.

Once a HUMBLE knowledge base has been created, a user can ask it for answers, and in turn be asked questions by HUMBLE. In these cases, HUMBLE will use its rules and information to come up with appropriate questions which it can use to get the data needed for an answer. It will then be able to explain its answer in English, giving the user a chance to evaluate the results.

1.2. Background Reading

There are a few books and articles which would be useful to read and understand before delving into this manual. The book *Rule Based Expert Systems*, edited by Buchanan & Shortliffe, gives a lot of background material and technical information on expert systems of the sort which HUMBLE can build. If you are unfamiliar with using Smalltalk, by all means read *Smalltalk–80: The Interactive Programming Environment* by Goldberg.

1.3. Examples

During the remainder of this manual, references will occasionally be made to a knowledge base called ROX. This is provided with HUMBLE and is a working example of one kind (the most common kind) of HUMBLE knowledge base. It may prove useful to have a copy of 'ROX.listing' handy as you read (it should appear as an appendix to this document). The listing will appear strange at first, but don't worry. It will become clear as you read further.

2. Knowledge Bases

In HUMBLE, the stored expertise in an expert system resides in a *knowledge base*. The topic on which the expert system is written is often referred to as the *problem domain*, or simply the domain. A knowledge base has two main parts, the static knowledge and the dynamic knowledge. Each of these two parts has a number of terms which are often used to describe it.

The dynamic knowledge, for instance, contains all of the information which changes from one consulation with the system to another. As such, it is often called the *fact base* (the term we will use most often in this manual), or the *working memory*. The fact base contains a number of entities, which will describe in detail a little later. This part of a knowledge base is strictly temporary.

The other part, the static knowledge, is a collection of all the information about the domain which doesn't change. Included in the static knowledge are descriptions of what sorts of entities can exist in the fact base, as well as how to use the information in fact base to make inferences and thus create new information. This description of what sorts of inferences are permissible is called the *rule base*. Each rule describes under what conditions a given conclusion can be drawn. The collection of definitions for entity types serves as a framework in which some part of the world can be described, and therefore have rules written about it.

3. HUMBLE Entities

Entities are the objects about which HUMBLE will make inferences. Each entity in the system is a representation of some thing or concept in the real world. As such, they serve as the context in which rules are evaluated, much as natural laws in the real world operate on objects and ideas. Defining the entities about which your knowledge base will reason is probably at least as important as defining the rules it will use. By designing your entities poorly, you can make the task of completing your knowledge base very difficult indeed.

It makes sense, then, to consider your entity types very carefully before you start to write rules. Try to map out a clear and complete description of the area of knowledge you wish to capture. There are probably clear divisions in your problem, and it will go much easier if you try to determine what they are. There is no need to map out exactly what the rules are at this stage, but you should at least get some framework of entity types in mind, one in which you can express your rules. This extra time in the beginning will pay huge dividends later.

Don't be intimidated, though. If you don't get your framework exactly right, you can always change it later. Minor or mid-sized changes can be accomplished in HUMBLE relatively easily, but major changes can take a great deal of time, especially if many rules have already been created. So get a good rough framework set up, and then refine it as you add rules.

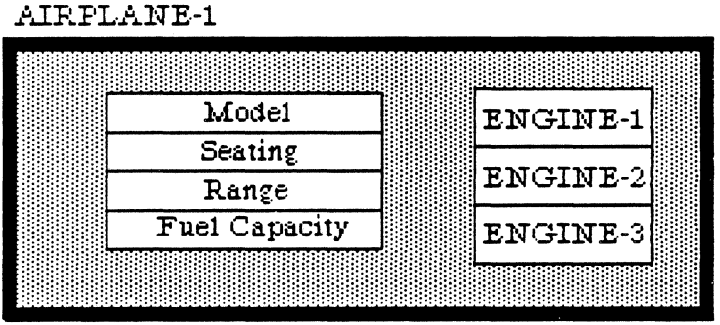
3.1. The Structure of Entities

As mentioned earlier, HUMBLE rules operate on entities. These entities fall into categories, called *entity types*. It is up to you to define the entity types which make sense for the topic your expert system will cover.

Entities are characterized by Parameters, which describe how entities of the same type differ from one another. For instance, an automobile entity might have parameters for: color, model, maker, options, etc.. Parameters are described in greater detail a

little later on. The parameters which entities of a given type will possess are specified in the entity type definition.

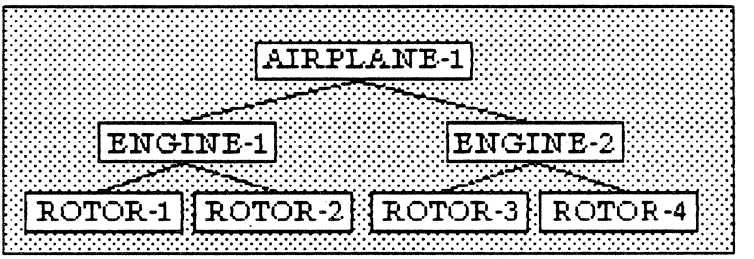
Along with parameters, an entity can hold other entities. These entities, called sub–entities, have their own sets of parameters and sub–entities, and so on. If you have had some experience with computers, you might notice that this follows a classical pattern of storing and retrieving data, called a tree. Sub–entities are a good way of breaking a problem into independent, but similar subsections. The entire structure, including the top entity and its sub–entities, and their sub–entities in turn, is called an entity tree. The place an entity holds in an entity tree is determined by a specification in its entity type definition. Knowing the place an entity holds in the tree is important when it comes time to write rules. The primary reason for this is that a rule can only have access to parameters in the entity it will execute against or one above the entity in the tree. More on this concept later.



A concrete example may be easiest to understand. Suppose you are building an expert system which will determine probable faults in an airplane. It would be reasonable to make the top entity type an airplane. Assuming that there is more than one engine, it would be useful to run the same set of rules on each engine independently of the others. This is exactly the situation where sub–entities are useful. You would create an engine entity type, and specify that it was part of the airplane entity type. Lets say that you also have a set of rules which can determine whether a given rotor within an engine has worn out. You might want to create an entity type for rotors, and specify

that a rotor is a part of an engine entity. An example of an entity tree created from these sorts of entities might look like this.

An Entity Tree



3.1.1. Entity Type Definitions

As noted above, entity type definitions are descriptions of how entities of a given type will behave. The typical template for an entity type definition which HUMBLE's Editor provides looks like:

EntityTypeName
typeAbove: anEntityType
createPrompt: Are there any <entity>s to consider?
addPrompt: Are there any other <entity>s to consider?
assumePrompt: I am creating an <entity>
defaultName: <entity>
parameters: #()
mainParameters: #()

Each line of the template defines some part of the behavior this type of entity will exhibit. A typical example, from ROX, looks like this:

Mineral
typeAbove: Rock
createPrompt: Are there any identifiable minerals in the rock
addPrompt: Are there any other minerals in the rock
assumePrompt: I am creating one mineral of the rock
defaultName: MINERAL
parameters: #(mineralName amount)
mainParameters: #(mineralName amount)

A rotor entity in the airplane example might look like this:

Rotor
typeAbove: Engine
createPrompt: Are there any rotors which may have defects?
addPrompt: Are there any other rotors to consider?
assumePrompt: I am creating a rotor
defaultName: ROTOR

parameters: #(numberOfBlades hoursInUse rotorDefective
)

mainParameters: #(rotorDefective)

3.1.1.1. The Type Above Specification

typeAbove: is the line which specifies the place in an entity tree which this type of entity can occupy. In the example given previously, the definition for the engine entity type would have a line saying:

typeAbove: airplane.

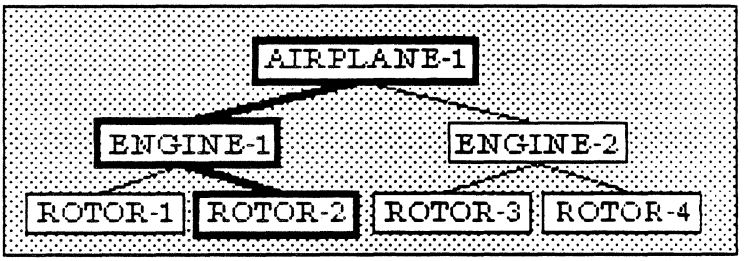
The definition for the rotor entity type has a line with:

typeAbove: engine.

The entity type for airplane is a special case, because it is the type of entity that should exist at the top of the entity tree. Because there is no type above it in an entity tree, it will have a line specifying:

typeAbove: nil.

An Entity Tree Path



3.1.1.2. The Prompts

Entity type definitions have three lines devoted to prompting the user, the **createPrompt**, **addPrompt**, and **assumePrompt**. Each of the three might be used during the creation of an Entity, depending on the specific information given during a consultation.

The **createPrompt** is used when a rule being fired implies that some entity of a given type may exist. HUMBLE will look to see if any have been created so far, and if not it will use the **createPrompt** to ask the user whether one of that type of entity exists.

The **addPrompt** is used only in situations where the **createPrompt** has successfully been used to create the first of a given type of entity. The **addPrompt** is then used to determine if there are any more of that type of entity to be considered.

The **assumePrompt** is used when the rules imply that there must be at least one of this type of entity. It simply informs the user that an entity has been created, so that he will not be surprised when asked about it.

In any case, the **defaultName** line is used to produce a name for the new entity. The standard practice is to create the name by appending a number to the default name. For instance, in ROX the default name for Rock entities is ROCK. Therefore, the first entity created of that type is named ROCK – 1.

3.1.1.3. The Parameter List

This is a list of all parameters which are associated with this entity type. You needn't worry about filling in this line, since the editor takes care of that detail for you. It appears mostly as a convenience, to provide an easy cross reference. Whenever HUMBLE is given a parameter definition, it will automatically add its name to the line of the appropriate entity type definition. Unfortunately, HUMBLE does not protect you from altering this list. Doing so is probably a good way to introduce errors into your knowledge base, and so should be avoided.

3.1.1.4. The Main Parameter List

The **mainParameters** line specifies which parameters you consider essential. Whenever an entity is created, each parameter on the main parameters list is immediately

determined by HUMBLE, in the order specified in the list. This is used for telling a knowledge base what parameters must be filled in for an entity to make any sense at all. It is also a nice way of causing the system to ask certain questions right at the start in a certain order.

To place a parameter on the main parameters list, just copy it from the parameter list and add it between the parentheses of the main parameter list.

3.2. Parameters

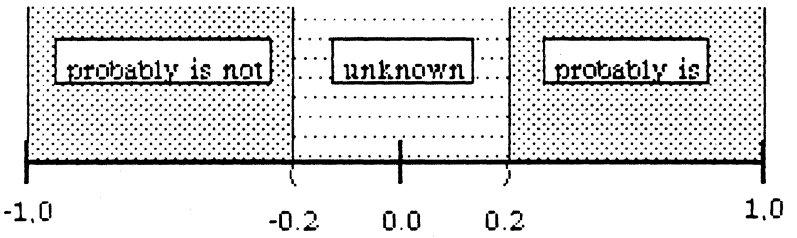
The Parameters of an entity are where the real information about that entity lies. If you are computer oriented, you can think of an entity as a collection of parameters added to a collection of pointers to where it can find more parameters. Each parameter is some relatively small bit of data about the entity. Different parameters can be of differing types: Strings, Numbers, etc.

Deciding what parameters are appropriate for any particular entity type is probably the major problem expert system designers face. A good representation will make completing your task easy, but a bad one can make it impossible. Our recommendation is to first write down some rules on paper to get a sense of what kind of things you are expressing in the rules. Then use that information to design a set of parameters for your entity. Think about each parameter and decide if it might not be better represented as a sub–entity. A little work like this at the beginning will help a lot in creating your expert system.

3.2.1. The Uncertainty System

Every parameter in a HUMBLE knowledge base can contain data which is *uncertain*. What this means is that each value may have some attached value which indicates how strongly HUMBLE believes that the value is the correct one for that parameter. HUMBLE uses these attached values to mimic the sort of reasoning a human expert performs in similar situations.

HUMBLE deals with uncertain data by attaching a *certainty factor* to each bit of data in the fact base. These certainty factors, in a standard HUMBLE knowledge base, are numbers ranging from −1.0 to 1.0. The system takes these factors into account whenever it draws a conclusion or makes a test. A 1.0 indicates absolute certainty that the value is the correct one. −1.0 indicates that it is absolutely certain that the value is not the correct one for whatever parameter it is associated with. 0.0 indicates a complete lack of knowledge.



HUMBLE decides that it is safe to declare a test true whenever the certainty of the result is either greater than 0.2 (for true). In general, 0.2 and −0.2 are called the cutoff certainties, because they represent the point where HUMBLE feels safe in making positive or negative declarations about the value of a parameter or the validity of a test.

These certainties are constantly considered during a consultation with HUMBLE. The exact algorithms are not necessarily of interest here, but in general they behave just as you might expect them to. If a conclusion depends on two factors which are uncertain, then the conclusion will be even less certain than either of the two factors considered. If several rules

support the same value, then the certainties add together to create a greater certainty.

HUMBLE supports the ability to have several different potential values for each parameter in the fact base simultaneously. Each of these potential values is referred to as a *hypothesis*. The *best hypothesis* is the hypothesis with the greatest certainty attached to it.

3.2.2. Parameter Definitions

Parameter definitions are the description you must create to define a parameter. The template for a parameter definition, which HUMBLE's Editor window produces, looks like:

```
ParameterDefinitionName
    describes: anEntityType
    type: YN,String,Number, MV, or #( array of possible values )
    prompt: What is the value of parameter for & ?
    promptFlag: askFirst, askLast, or nil
    explanation: "Not yet explained"
    remark: "Not Yet Commented"
    changeBlock: [:parameter |]
```

Each line of the template defines some part of the behavior this type of parameter will exhibit. The following sections will define each line in detail.

3.2.2.1. Describes

This line of the definition indicates which type of Entity this parameter is associated with.

3.2.2.2. Parameter Types

The type line of a parameter definition specifies what sort of parameter this is, YN (yes/no), MV (multi–value), Number, or String. This is probably going to be dictated by the nature of the thing you are describing. Each type has a different set of

characteristics and uses, described in the sections which follow.

3.2.2.2.1. YN Parameters

YN Parameters are yes/no parameters. They represent some facet of an entity which is either true or false. An example might be whether or not a switch is on, whether or not a dog has fleas, or whether or not the patient survived.

A typical example of this sort of parameter is the parameter **fossiliferous** in the ROX example knowledge base.

3.2.2.2.2. MV Parameters

MV parameters are parameters which can hold more than one value. In many respects they are like keeping a simple list of things. Some examples of MV parameters might be: the list of diseases a patient has contracted in the past, The list of problems with an automobile, or perhaps the names of all employees in some small company.

Essentially, all you can do with an MV parameter is to either add some element to the list or test whether something is already in the list.

A typical example of this sort of parameter is the parameter attributes in the ROX example knowledge base.

3.2.2.2.3. Number Parameters

Number parameters contain a single number. You can do any of the standard numeric operations on number parameters.

It is difficult to use them to calculate values as you would in a standard programming language. I'll try to explain. First of all, HUMBLE is a system which deals with uncertainty. The approach to uncertainty which HUMBLE takes requires the system to consider ALL of the possible values for a parameter and then choose among them once all of the evidence is in. It is this very feature, which allows HUMBLE to mimic human

reasoning about uncertain data, which prevents it from being effectively used for computation. If you try to make the conclusion:

```
A is: A + 1
```

assuming A is now 5, in an uncertain system, it will now have two possibilities: that A is 5, and that A is 6. See the problem? You will probably have to attempt to add 1 more than once in order for the system to have enough evidence to actually believe A to be 6. This makes it very difficult indeed to perform predictable mathematical computation using uncertain parameters. If you feel a deep need to do this sort of computation, there are ways to do it outside of the certainty mechanism. See section 4.4.4: 'Smalltalk Escape Sequences', for more information on how to go about this.

A typical example of this sort of parameter is the parameter grainSize in the ROX example knowledge base.

3.2.2.2.4. String Parameters

String parameters contain a single string of characters. They are usually used as names or adjectives for the entity.

Other than the usual tests, (such as >, <, =, >=, <=), HUMBLE allows you to concatenate strings together. You use the comma (,) operator to do this. As with Numbers, rules which do many concatenations can be difficult to predict.

A typical example of this sort of parameter is the parameter name in the ROX example knowledge base.

3.2.2.2.5. Enumerated Parameters

Enumerated parameters are a special case of String and Number Parameters. An enumerated parameter is specified by giving an array of possible values which the parameter can take. The array must be either all numbers or all strings.

For example, the line:

```
type: #('a' 'b' 'c' 'd')
```

specifies that this parameter can only have string values of 'a', 'b', 'c', or 'd'. Other than this specification of possible values, they are just like any other String or Number parameter.

A typical example of this sort of parameter is the parameter class in the ROX example knowledge base.

3.2.2.3. Prompting

There comes a time when a HUMBLE knowledge base needs to ask the user a question in order to obtain the value of some parameter. Two parts of a parameter definition control how this is done. The **prompt:** and **promptFlag:** lines of the definition control the text of the prompt and the order in which information is sought, respectively.

3.2.2.3.1. The Text of the Prompt

The text of the prompt is set using the **prompt:** section. This piece of the definition is essentially a string phrased exactly as you would want the system to ask the question. You can, however, also add some program information into the question by using the & character.

For instance, the following line in a parameter definition (assuming the parameter was part of an entity called PATIENT – 1):

```
prompt: What is the temperature of & ?
```

would produce the question:

```
What is the temperature of PATIENT – 1?
```

If you want to plug in the value of some parameter as part of the question, you can place the name of the desired parameter immediately after the & character. If our first example is changed to:

```
prompt: What is the temperature of &patientName?
```

the result might be:

```
What is the temperature of 'Phillip Davis'?
```

assuming, of course, that the name of the patient was indeed Phillip Davis.

3.2.2.3.2. The Prompt Flag

Whenever HUMBLE is asked to ascertain the value of some parameter, about the first thing HUMBLE does is check the value of this flag.

If the flag is set to **askFirst**, the system will try to get the user to answer what the value is, and will only try to infer it if the user says he doesn't know. This is usually, but not exclusively, used in the case where the parameter is for input only.

If the flag is set to **askLast**, the system will attempt to deduce the value, and ask the user only if it cannot infer the answer itself.

If the flag is set to **nil**, then the system will attempt to deduce the value, and if it cannot find a reasonable value it will simply chalk up the parameter as having an unknown value. The **nil** flag specifies that the user should never be asked the value of this parameter. It is often used in the case of internal parameters, where the user is never going to be interested in the value of that parameter.

3.2.2.4. Explanation

The **explanation:** line allows the system developer to input some explanatory text about this parameter. Treat this as if you were defining your parameter as a word in a dictionary. It may be useful to decribe what makes this parameter interesting or important. This line is used by the explain function of the Listener window to explain selected pieces of text. The section on using that window will describe this feature in more detail.

3.2.2.5. Remarks

Remarks are very similar to explanations, and are reserved as comments for the parameter. You should note anything unusual about the parameter and how you are using it, so that someone else who reads your knowledge base can understand what you did. No part of the standard HUMBLE window setup uses this line.

3.2.2.6. Altering the Order in which Deducing Rules are Fired

The **deducingRules:** line lists all of the rules which this parameter might fire in order to deduce its own value. They are listed left to right in the order in which they will be evaluated. If you are unsatisfied with the order HUMBLE chooses for them, (and usually the exact ordering makes no difference to the final answer) you can change the order by moving the rules around in the list with copy/cut/paste editing.

If you decide the order yourself, you should be careful of what are technically termed *recursive rules*. A recursive rule is a rule that makes a conclusion about some parameter it checks in its premise. An example of such a rule might be:

```
if: petType = 'dog' & (breed mightBe: 'Labrador')
then: [
    if: furColor = 'black'
    then: [breed is: 'Labrador' withCertainty: 0.2].
    ].
```

Notice that the parameter breed appears in both the premise and in the action block. As with most rules of this sort, this rule is used to follow up a specific piece of information and thereby make it more certain.

These recursive rules should be at the end of the list, rather than the beginning, or they will introduce very odd behavior. HUMBLE does not at present protect you from introducing this sort of error while altering the order of rules, so this feature is strictly *caveat emptor*.

Note – Observant readers may have noticed that the deducingRules: line does not appear in the standard parameter definition template. This is not an oversight. A brand new parameter definition should not have any deducing rules associated with it, so we didn't give anyone the chance to introduce an error during the creation of a parameter definition.

3.2.2.7. The Change Block

The change block is an advanced feature for Smalltalk programmers. If you aren't a Smalltalk programmer, you can either ignore or remove entirely the **changeBlock:** line of the parameter definition. Then just skip the rest of this section.

If you are a Smalltalk programmer, you can use the change block to activate some piece of Smalltalk code whenever the best hypothesis about some parameter has changed. The contents of the change block will be executed whenever this occurs. Your change block should accept one argument, the actual parameter which has changed. You can send it any messages you like, but use only the argument and any global variables from within the change block. You can make mincemeat of your system by trying to modify parameters from within the change block, so be

VERY sure you want to do it before attempting that sort of thing. The change block is useful for a number of things, but is mainly intended to aid with animation or other similar interface problems.

3.3. The Scope of Parameters

In order to avoid confusing the rules, HUMBLE allows them access to parameters only from certain parts of the entity tree. A rule can only access data in the entity it is evaluating in and any above it in the entity tree. This is a very hard concept to explain if you haven't had some experience with it, so don't be dismayed if this section sounds like gibberish the first time you read it.

Think of the example entity tree we discussed earlier, that of an airplane entity and the engine sub-entities it contained. Suppose HUMBLE is trying to execute a rule which will determine if one of the engines is malfunctioning. Obviously, the decision made by HUMBLE should only be based on information about that specific engine, not one of the other engines. This need to isolate data is the reason that a rule can only access data in the entity it is executing against and those above it in the entity tree.

The question which is probably occurring to you right about now is how to determine what type of entity a rule will execute against. The answer is that HUMBLE chooses what sort of entity a rule will execute in by choosing the entity type which:

1. Has one of the parameters you referred to in the rule.
2. Is lowest in the entity tree structure.

Thus, if a given rule in the airplane example refers to parameters in both an engine entity and a rotor entity, it would choose to evaluate the rule against the rotor entity. On the other hand, of the rule were altered so that it no longer referred to any parameters in a rotor entity, the rule would be evaluated against an engine entity instead.

Most of the time, this limitation is not apparent to the user, but occasionally HUMBLE will gripe about a rule you are trying to accept, saying: 'You are attempting to set parameters in separate

branches of an entity tree. This is not allowed.'. This simply means that you are trying to write a rule which would need to be 'executing against two places at once' on the entity tree in order to have access to all the data you specified, and that it cannot accept any rule which would require that. The thing to do here is to rewrite the rule so that it uses only parameters available in a single path of the tree, or to modify your entity tree structure so that it can execute the rule.

4. HUMBLE Rules

4.1. Introduction

The essence of any rule based system is, of course, the set of rules which compose the major part of any knowledge base. These rules are created using the HUMBLE knowledge base editor, and are in many respects rather similar to Smalltalk – 80 code. This manual makes the assumption that you are a minimally competent Smalltalk programmer, and understand the basic elements of creating Smalltalk expressions and statements. If you aren't at least roughly familiar with Smalltalk code, I urge you to take the time to create a few small methods in Smalltalk – 80.

4.2. Basics

This section attempts to describe the basic elements of writing rules in HUMBLE. Be sure you have read the chapter on HUMBLE entity trees before reading this chapter, since it will use concepts explained in that chapter.

4.2.1. HUMBLE Rules and Smalltalk – 80

The rule syntax of HUMBLE, although reasonably easy to use, is most definitely not Smalltalk – 80 code. However, some of the same constraints apply to HUMBLE rules that apply to Smalltalk code. Deep within HUMBLE, the rules you create are converted into actual Smalltalk code which is stored as a block. Therefore, expressions are evaluated in the standard Smalltalk order: unary, then binary, then keyword messages are evaluated left to right. If – then statements and similar constructions can be considered to be keyword messages.

4.2.2. The Structure of a HUMBLE Rule

There are three major parts to any HUMBLE rule. These are the name, the translation, and the statements. The template HUMBLE provides for making a new rule looks like:

RuleName
"translation – this should explain the rule's premise and conclusions, so that the explanation output makes sense"
rule statements

The name appears in bold face on the first line of the rule. This is the name which the rule is stored under in the knowledge base. This name can be used during forward chaining to locate the rule to fire next. It is also the name by which the rule can be selected for editing in the editor window, and is generally a convenient handle for locating the rule. You can do as you like with the rule names: some prefer to use rather bland and uninformative names, ruch as rule001, rule002, etc. These are okay, but I suspect that you will find it preferable to make the names as informative as possible. You can make them as long as you like, but they must be a single long string of characters without spaces. The typical Smalltalk convention on such things is to use capital letters to delineate individual words, such as ruleCheckingTemperature or checkForPhaneriticBiotite. I urge you to use the more informative names, as it will vastly enhance the readability of your rules.

Immediately after the name, the section of text in quotes and appearing in italics is the translation. The translation is primarily used by the explanation mechanism to indicate why particular conclusions have been drawn. Use this section to provide a plain English translation of the rule you are writing. The translation serves the dual purpose of a comment and an element of an explanation.

The statements come after the translation, and are the real substance of the rule. Depending on what statements you place in this section, HUMBLE will make different kinds of inferences. In the next few sections, we will explore the various possibilities.

4.2.3. Rule Statements

4.2.3.1. If – Then – Else Statements

The typical HUMBLE statement follows the form:

```
if: ( <premise> )
then: [ <action> ].
```

The term *premise* is a good one to remember, as it will appear again and again in this document. The premise is the test in the statement which decides whether the action part of the statement should be executed. Similarly, the action of a statement describes what HUMBLE should do whenever the premise is considered to be true.

If desired, an else clause can be added to the statement to specify an action when the premise is considered to be false. These statements follow the form:

```
if: ( <premise> )
then: [ <action> ]
else: [ <alternate action> ].
```

4.2.3.1.1. The Premise

The premise is either the value of a YN parameter or the result of a test on some other sort of parameter. For instance, crystalline is a YN parameter in the example knowledge base ROX. The premise of some statement might very well be:

```
(crystalline)
```

This would mean that you want the statement to take action only when the value of crystalline is considered to be yes (or true). Another sort of premise might well check to see that color, a numeric parameter, is less than 50. This would be expressed as:

(color < 50)

This statement would only take action when the value of color is less than 50. It is just as feasible to test the value of one parameter against another, as long as they are of the same type. The premise:

(color < computedMinimum)

would specify that the statement should only take action when the value of color is less than the value of the parameter named computedMinimum.

4.2.3.1.2. The Action

The action is the part of the statement you want to occur whenever the premise is considered to be true. The actions you can take are quite varied. Conclusions can be drawn, further statements evaluated, other rules fired, and new goals can be set up. Each of the next four sections explains one of these activities.

4.2.3.1.2.1. Making Conclusions

One of the most likely things which you might want to do in the action of a statement is to make some conclusion. A simple conclusion follows the following form:

<parameter> is: <value>.

This tells the system that the value of the specified parameter is certainly the specified value. For instance:

breed is: 'Labrador Retriever'.

sets the value of parameter breed to be the string 'Labrador Retriever'. Since everything in HUMBLE is uncertain, what

this conclusion is actually saying is that "I am as certain that breed is 'Labrador Retriever' as I am that the premise is true".

In some cases, the conclusion being made is not as certain as the premise, and in this case a conclusion can follow the form:

. <parameter> is: <value> withCertainty: <certainty>.

This type of conclusion means that this conclusion does not always hold, even if the premise is certainly true, but that it can be made with a confidence of <certainty>. In standard HUMBLE, these certainties run from −1.0 (absolute disbelief) to 1.0 (absolute certainty). By using a negative certainty, you are telling HUMBLE that the specified parameter probably *doesn't* have the value specified. Negative certainties are often forgotten as options when writing statements, but often make the difference between a very accurate and effective knowledge base and a rather poor one.

4.2.3.1.2.2. Evaluating Other Statements

A *nested statement* is a statement with one or more If statements inserted into one of its action blocks. These statements are useful for doing several sorts of things. The most useful one of these is to control what parameters have values determined for them. In many cases, if no good answer has appeared, a backward chaining system like HUMBLE will check a number of rather useless possibilities. The easiest way to prevent this checking is to nest statements so that particular premises are not evaluated unless some precondition is met. An example of a rule with a nested statement is:

```
phaneriticColor
    "if the rock is phaneritic
        then
            if the color >=50 then the rock is possibly a
gabbro
            if the color < 20 then the rock is possibly a granite
            if the 20 < = color < 50 then the rock may be a
diorite"


    if: aphanitic not
    then: [
        if: color > = 50
        then: [name is: 'gabbro' withCertainty: 0.4].
        if: color < 20
        then: [name is: 'granite' withCertainty: 0.4].
        if: color > = 20 & (color < 50)
        then: [name is: 'diorite' withCertainty: 0.4]]
```

Notice that this is a nested rule which checks aphanitic (a YN parameter), to see whether it is false. If it is, then it will evaluate three nested If–Then statements, taking what action is appropriate.

It is perfectly permissible to use any of the other kinds of statements (described later) in a nested statement.

4.2.3.1.2.3. Firing Other Rules

Another action often taken is the execution of another rule. This sort of thing can be accomplished in two rather different ways in HUMBLE. The standard way in which a rule is executed is as part of HUMBLE's attempt to find out the value of a parameter. This is called *backward chaining*.

The other method of firing a rule is to use a *forward chain specification*. This is a directive to HUMBLE to execute a rule immediately. Forward chain specifications follow the form:

```
{@ruleName}.
```

For example, let's say that you wish to execute a certain rule called **consistencyChecks**, whenever certain conditions are reached. The rule might look something like this:

```
checkConsistencyWhenNeeded
    "If certain conditions are met
        then fire the consistency checks"

    if: certainConditions
    then: [ {@consistencyChecks} ].
```

It is perfectly permissible to mix forward chain specifications, conclusions, and nested If–Thens as needed in the action of a rule.

4.2.3.1.2.4. Setting New Goals

It is very seldom the case that you need to find out the value of a parameter before HUMBLE would normally do it on its own. However, you can specify this as part of an action simply by inserting the parameter name followed by a period, such as:

```
parameter.
```

This can be used to good effect if for some reason you want the rules to explore some indirectly related subject which HUMBLE might normally not even consider at all based on the evidence it has received. This feature is not normally of interest to any but the most advanced expert system builders.

4.2.3.2. If–Any, If–All, and If–None Statements

In the discussion of entity trees, the concept of sub–entities was put forth as a method of breaking a problem down into parts. Obviously, breaking a problem down in this way is of little use if the individual parts cannot be tested and have conclusions made about them. If–Any, If–All, and If–None constructions are used to test the sub–entities of the entity against which a rule is executing. This is a difficult concept, but once you see a few rules

which use this sort of construction it will begin to make sense.

These rules have an extended sort of premise, which must specify two parts. There is still a premise similar to the one used in simple If–Then–Else statements, but there is also a specification of what sorts of entities should be tested this way. The actual rule looks like:

```
ifAny: <entity type name>
has: [ <premise> ]
then: [ <action> ].
```

This statement does the following: *"Search all of my sub–entities for entities of the specified type. Once these are found, test each one against the specified premise, and if the premise is considered to be true then take the action in the context of the sub–entity"*. Notice that during the premise and action parts of the statement the context in which the rule is executing has changed to be that of the sub–entity under consideration. This sort of rule is called a *context switching rule* for this reason. The switch is only temporary, only occurring for the duration of the rule's firing.

There are two other forms of this type:

```
ifAllOf: <entity type name>
have: [ <premise> ]
then: [ <action> ].
```

and

```
ifNoneOf: <entity type name>
have: [ <premise> ]
then: [ <action> ].
```

These statements are a little different, since they only switch the context of execution during the testing of the premise. In other words, the first statement means: *"Search all of my sub–entities for entities of the specified type. Once these are found, test each one against the specified premise, and if the premise is considered to be true for each and every one of the sub–entities then take the action"*.

The second statement can be translated as: *"Search all of my sub–entities for entities of the specified type. Once these are found, test each one against the specified premise, and if the premise is considered to be false for each and every one of the sub–entities then take the action".*

The action blocks of these statements are exactly similar to the action blocks of simple If–Then–Else statements. The premise part is also exactly the same.

4.3. Rule Execution

HUMBLE is one of a general class of expert system tools which use *backward chaining* as the primary method of directing the evaluation of rules. When HUMBLE is asked (by a user) to find out the value of some particular parameter in some entity, it will find what rules can help deduce that value and fire them in order. In most cases, each of these rules will need to know the value of other parameters in order to make a decision, so before it decides about the first rule it will attempt to find out the value of those other parameters. Each of those other parameters has a set of rules which may help deduce it, and so HUMBLE attempts to find out the value by firing those rules. Each of *those* rules may need to find out parameters, and so on. This process is referred to as backward chaining or *goal directed inferencing*. When the user requests the value of some parameter, finding that value becomes the *goal* of the inference engine. In the process of satisfying the goal, the system may realize it needs more information and set up new goals to get that information. Thus, the present goal directs that system to fire the appropriate rules in the correct order.

It means, in simplest terms, that HUMBLE guarantees the correct order of rule firing based on what the rules say. You needn't specify any particular order in advance.

There are ways to alter the order in which HUMBLE would otherwise fire rules. The **deducingRules** line of the parameter definition is a list of what order the rules are to be fired in to find

out that particular parameter. By changing the order of that list, you can change the order of execution.

Placing forward chain specifications is also a way of telling HUMBLE to execute a rule at a certain point.

4.4. Syntax BNF

This section provides a formal specification of HUMBLE's rule language as a reference. It is probably not useful to read this section word for word. Use it as a reference guide, if you need specifics about some point of the rule language.

The representation used here is a variant of Backus–Naur (or Backus–Normal) form (BNF). For those reading this manual unfamiliar with the concept, it is a formal specification language for certain types of grammar. If you can't read it, just ignore this section or get someone to explain it to you (it's well worth knowing). As usual, this sort of representation is precise but almost totally unreadable. Therefore, a number of examples will be given later on.

4.4.1. BNF Conventions

The entities separated by the '|' symbol are one of a number of acceptable alternatives. '<number> | <string>' would mean that either a number or string was acceptable at this point.

Any entity followed by a '*' means that repetition of the entity 0 or more times is acceptable.

Entities appearing in italics between " (double quote) symbols are explanatory comments.

An example knowledge base can be found in the file 'ROX.kb'.

4.4.2. The BNF

<literal> := <number> | <string> | true | false | (<escape>)

<entityType> := <string> *"the name of one of your pre – defined entity types"*

<parameter> := <string> *"the name of one of your parameter definitions"*

<comment> := "<string>"

<value> := <literal> | <parameter>

<op> := + | – | * | / | // *"integer divide"* | \\ *"modulo"* |
, *"comma is a string concatenation operation"* | raisedTo:

<expression> := (<value> <op> <value>) | <value>

<escape> := '<<' <any smalltalk code> '>>'
"example: <<Transcript show: 'Hi There!'>>"

<binaryBooleanOp> := < | > | <= | >= | = | ~= |
"section following explains the following alternatives"
definitelyIs: | definitelyIsNot: |
isNotDefinitely: | isNotDefinitelyNot: |
isEqualTo: | isNotEqualTo: |
isNotKnownToBe: |
mightBe: | mightNotBe:

<unaryBooleanOp> := isKnown *"certainty > 0.2"* |
isNotKnown *"certainty <= 0.2"* |
isDefinite *"certainty = 1.0"* |
isNotDefinite *"certainty < 1.0"*

<booleanConjunction> := & *"logical and"* |
| *"| character is logical or"*

```
<booleanClause> :=
    (<expression> <binaryBooleanOp> <expression>)|
    (<expression> <unaryBooleanOp>)|
    (<specialBooleanExpression>)

<booleanExpression> :=
    (<booleanClause> <booleanConjunction>
<booleanClause>)

<premise> :=
    <booleanExpression> <<booleanConjunction>
<booleanExpression> >*

<specialBooleanExpression> :=
    (anyOf: <entityRType> have: [<premise>])|
    (allOf: <entityRType> have: [<premise>])

<conclusion> :=
    <parameter> is: <expression>. "complete certainty"|
    <parameter> is: <expression> withCertainty:
<number>.

<if-then-rule-form> := if: <premise> then:
[<conclusion>*].

<if-all-rule-form> :=
    ifAllOf: <entityType>
    have: [<premise>]
    then: [<statement>*].
"Explicit context switch during execution of the premise block."

<if-any-rule-form> :=
    ifAnyOf: <entityType>
    have: [<premise>]
    then: [<statement>*].
"Explicit context switch during execution of the permise block and
the action block. The action block statements are applied in the
context of each entity of the specified type which meets the
conditions set in the premise block"
```

<if-none-rule-form> :=
    ifNoneOf: <entityType>
    have: [<premise>]
    then: [<statement>*].
"Explicit context switch during execution of the premise block."

<ruleStatement> :=
    <if-then-rule-form>|
    <if-all-rule-form>|
    <if-any-rule-form>|
    <if-none-rule-form>|
    <escape>

<forwardChain> := {@rulename}.

<statement> :=
    <ruleStatement> |
    <conclusion> |
    <forwardChain>

<rule> := <rulename> <comment> <statement>*

4.4.3. Special Boolean Operations

For purposes of defining the results of these functions, use the following reference clause:

A <booleanOp> B

A and B may be either literals or parameters. In the case of literals, assume each has a best hypothesis equal to the literal's value, and a certainty of 1. The phrase 'best hypothesis of A equal to best hypothesis of B' means more specifically: "find the value of the best hypothesis of B, then find the best hypothesis of A with that same value". These definitions deal with the standard HUMBLE certainty model.

definitelyIs: best hypothesis of A that is equal to best hypothesis of B has certainty = 1.0

definitelyIsNot: best hypothesis of A that is equal to best hypothesis of B has certainty = −1.0

isNotDefinitely: best hypothesis of A that is equal to best hypothesis of B has $0.2 <$ certainty < 1.0

isNotDefinitelyNot: best hypothesis of A that is equal to best hypothesis of B has $-0.2 >$ certainty > -1.0

isEqualTo: best hypothesis of A equal to best hypothesis of B has certainty > 0.2 *Note: isEqualTo and = are distinct operators. = implies that the best hypotheses of the two are equal, while isEqualTo: implies only that some hypothesis is reasonably certain and equal*

isNotEqualTo: best hypothesis of A that is equal to best hypothesis of B has certainty < -0.2

isNotKnownToBe: best hypothesis of A that is equal to best hypothesis of B has $|\text{certainty}| < 0.2$

mightBe: best hypothesis of A that is equal to best hypothesis of B has certainty > -0.2

mightNotBe: best hypothesis of A that is equal to best hypothesis of B has certainty < 0.2

Programmer's note: The certainty figures mentioned here are not hardwired, but are instead related to the certainty constants set up in whatever certainty model HUMBLE is using in this particular knowledge base. See section 10.5: 'Altering a Certainty Model', for more information on these constants and how to alter them.

4.4.4. Smalltalk Escape Sequences

It is often the case that a rule language will simply be inconvenient to use in certain cases. HUMBLE and similar systems have notable problems in the case of complex algebraic calculations. In order to ease the burden of those needing special behavior, HUMBLE implements an 'escape' to Smalltalk, allowing Smalltalk–80 code to be imbedded in any HUMBLE rule.

By placing a '<<' into the source text of any HUMBLE rule, you signal the rule processor to place the text up to a '>>' into the finished rule verbatim, with the exception of parameter references. Parameter references are still translated during escape sequences, since they are a very convenient shorthand for retrieving values out of the fact base.

A rule using a Smalltalk escape sequence might look like this:

```
stabilityAlert
    "if the system shows a very low stability, make an
immediate alert by using the Smalltalk-80 transcript
mechanism"
    if: systemStability < 0.3
    then: [
        << Transcript cr; show: 'Alert: System stability
dangerously low!'.
            Transcript cr; show: systemStability
printString.>>
        ].
```

Calculations made in escape sequences are not subject to the certainty calculations which HUMBLE generally performs automatically, and probably never will be. In many cases, this is precisely why they are used. However, if you are going to try to plug them back into HUMBLE, you should read the programmer's manual (chapter 10) for information about interactions between HUMBLE and Smalltalk–80.

4.5. Example Rules

A number of example rules follow, with some explanation of what should be going on when the rule is fired.

4.5.1. Typical Rules

Here are a few typical rules:

```
systolicPressureCheck
    "if the patient's systolic pressure is high
    then his condition is possibly critical"
    if: (systolicPressure > 120)
    then: [condition is: 'critical' withCertainty:
0.4.].
```

In this rule, the premise checks the parameter systolicPressure against the literal 120. if systolicPressure is greater, then conclude that parameter condition is now 'critical', with the rather low certainty of 0.4.

```
soilAcidity
    "litmus > 50% blue indicates high soil acidity
and poor lime content"
    if: (acidityCutoff < = litmusColor)
    then: [soilAcidity is: 'high'.
            limeContent is: 'poor' withCertainty:
0.8.]
    else: [soilAcidity is: 'acceptable'.
            limeContent is: 'acceptable'
withCertainty: 0.8.].
```

This rule checks the color of litmus paper (parameter litmusColor) against the parameter acidityCutoff. If the test indicated high acidity, the parameter soilAcidity is set as well as the parameter limeContent. Note that this rule uses an else clause.

4.5.2. Searching Rules

Certain rules are searching rules, which make conclusions about any sub – entity which meets a specified criterion. An example:

```
olivineCheck
    "if this rock contains olivine
    then it is very likely a gabbro,
        it is unlikely to be a granite,
        and it is unlikely to be a diorite"
    ifAnyOf: Mineral
    have: [mineralName = 'olivine']
    then: [name is: 'gabbro' withCertainty: 0.8.
           name is: 'diorite' withCertainty: -0.5.
           name is: 'granite' withCertainty:
-0.5.].
```

This rule checks whether olivine is present among a rock's minerals. If any of the sub – entities of type Mineral have parameter mineralName set to 'olivine', then the system makes conclusions about what sort of rock this one is likely to be. During execution, the context is switched to execute in each Mineral entity, both for the premise and the conclusions. Note the negative certainty in one conclusion, indicating that the system believes that the rock is not diorite or gabbro.

4.5.3. Nested Rules

In some cases, for efficiency's sake, it is desirable to nest rules. HUMBLE allows this sort of rule and will take the necessary steps to see that certainty information is properly maintained for conclusions made by the nested rules. Here is an example:

```
siltstoneCheck
    "if the rock is sedimentary, small grained, and
gritty to the tongue
    then it is probably a siltstone"
    if: (class = 'sedimentary' & (grainSize <=
0.1))
    then: [
        if: gritty
        then: [name is: 'siltstone' withCertainty:
0.8.].
        ].
```

If more than one if–then is placed as part of an action block, then the result is an implied forward chain, since the nested statements will be executed in sequence as they appear in the rule.

4.6. Error Messages During Compilation

HUMBLE can catch a number of common errors during compilation of its rules. The error message will appear right in the rule itself, along with an error pointing to the spot where HUMBLE detected that something was wrong. A list of these follows, including some explanations of what may have gone wrong.

A rule cannot forward chain to itself

A forward chain specification specifying the rule being compiled as the rule to be executed is illegal. HUMBLE would not execute the forward chain in any case.

have: expected

HUMBLE was processing a special boolean or searching rule and expected a have: keyword. Check for missing colons.

is: expected

HUMBLE was processing an assertion and expected to see an is: keyword at this point. A missing colon can cause this.

Missing close bracket for condition block

HUMBLE has found what it believes is the end of a condition block, but no closing bracket (']' character). Often, this is caused by a poorly formed expression.

Missing closing parenthesis

HUMBLE has found what it believes is the end of a parenthesized expression, but no closing parenthesis. Often, this is caused by a poorly formed expression.

Missing open bracket for condition block

Some statement with an condition block was being processed, but the open bracket ('[' character) was forgotten at the beginning of the condition block.

Missing open bracket for action block

Some statement with an action block was being processed, but the open bracket ('[' character) was forgotten at the beginning of the action block.

Missing period or close bracket

This can mean many things, but is most often caused by misspelling withCertainty: when making an assertion as part of an action block.

No rule

This one is simple. HUMBLE couldn't detect anything which looked like the beginning of a statement. Often, this means you left a colon off of the first word in the statement.

Nonexistent rule:

A forward chain specification specified a nonexistent rule for execution. A spelling error is the likely cause here.

Only numbers can be negated

A unary negative sign has appeared in front of something besides a number literal. This is not allowed. To negate a parameter's value, use the unary operation negated.

Parameter name, literal value, or escape sequence expected

HUMBLE was looking for a parameter name, literal, or escape sequence and found something else. Check for misspellings.

Statement expected

This is much like the no rule message, meaning that HUMBLE was expecting to see a statement and didn't recognize the beginning of any type of statement it understood. Check for missing colon characters on your first word.

then: expected

Humble was processing an if: statement, believed it had found the end of the condition, and didn't find a then: keyword. A missing colon or a poorly formed condition may have caused the error.

Test expected

HUMBLE was processing a condition and expected to see a test. This error may be the result of misspelling, missing colons, or a poorly formed boolean expression.

Test is valid only after a parameter name

A test was used that is only valid if it is used directy after a parameter name. You will need to rewrite the expression or choose a different test.

Undefined entity type

While processing a searching rule, HUMBLE found something else where it was expecting an Entity Type name. check for misspellings.

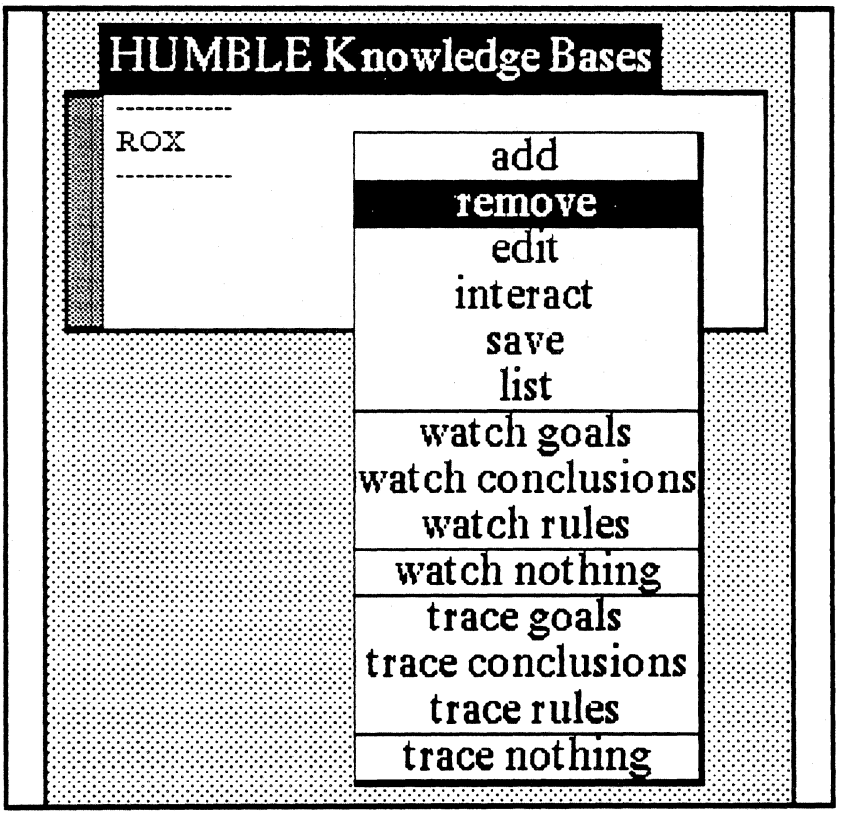
Undefined parameter name

HUMBLE was looking for a parameter name and found something else. Check for misspellings.

5. The HUMBLE Manager

5.1. Introduction

The HUMBLE Manager window is the top level interface to the collection of knowledge bases you have loaded into your image. It serves as the place to load, save, list, remove, and generally control the behavior of your knowledge bases.



5.2. Adding and Removing Knowledge Bases

When the Manager window first appears on screen, it will have an empty list. In order to add a knowledge base to the list, you use the middle–button command **add**. The system will respond to the add command with a prompter asking you for the name of the knowledge base you would like to add. HUMBLE is provided with an example knowledge base called 'ROX'. HUMBLE locates the file containing theknowledge base by finding a file with the knowledge base's name followed by '.kb' (ROX, for instance, can be found in the file named 'ROX.kb'). If no such file exists on the local disk, HUMBLE creates a new empty knowledge base with that name. If there is a file, though, HUMBLE will read the contents and attempt to create a knowledge base. It may take considerable time for a knowledge base to be loaded, at least as long as it would take to file in an equal amount of Smalltalk code.

A knowledge base can be removed from the list by selecting it with the left button, and then using the middle–button **remove** command. This will remove the knowledge base from local memory, but not the stored disk file. It is left to the user to delete any disk files from his disk by using the file list or some similar technique.

5.3. Editing Knowledge Bases

Any knowledge base in the Manager's list may be edited by selecting it with the left button and using the middle button **edit** command. This will open a HUMBLE knowledge base Editor on the selected knowledge base. See the section on using the knowledge base editor for a detailed description of how to use the window.

5.4. Consulting Knowledge Bases

In order to consult a knowledge base, select the base on the Manager's list and then use the **interact** command from the middle button menu. This will open a HUMBLE Listener window on the knowledge base. The section on Listener windows

goes into detail about how to use the Listener to consult with your knowledge base.

There are, however, some kinds of output which are controlled from the Manager window. Information about which rules, goals, and conclusions are being evaluated can be sent directly to the Listener window or to a trace file. These features are of small interest to the casual user, but the builder of an expert system can find them invaluable for debugging his rules. Without a trace or watch running, it is impossible to tell which rules are being executed and making conclusions. The series of commands allowing control of this output are described in the next two sections.

5.4.1. Controlling Output During a Consultation

One of the options available to you is the ability to watch the execution of HUMBLE as a part of your consultation. The commands **watch - - > goals**, **watch - - > rules**, and **watch - - > conclusions** all turn on display of certain information.

Watch - - > goals will tell HUMBLE to notify the Listener whenever HUMBLE has a new parameter which it must find out. Such parameters are often referred to as goals. A notice will appear as part of the consultation that HUMBLE is now attempting to find out the value of some parameter. A second notice will appear when HUMBLE is satisfied that it has found out as much as it can about that value.

Watch - - > rules tells the system to notify the user when a rule is fired. A notice will be sent to the Listener for each rule fired during a consultation.

The **watch - - > conclusions** command informs HUMBLE that it should notify the user of each conclusion made by a rule. A notice is sent to the Listener window that some conclusion has been drawn.

The command **watch - - > nothing** tells the system to stop sending notices about goals, rules, or conclusions. It is the only way presently available to turn off these functions once they have been activated. If you want to turn off just one of the three, you must turn them all off using this command and then reactivate only the desired outputs.

5.4.2. Tracing Execution During a Consultation

Tracing is very similar to the commands for watching, except that the results are written to a file instead of showing up as part of the consultation. The commands **trace - - > goals**, **trace - - > rules**, and **trace - - > conclusions** are all similar to the equivalent watching commands. When activated, the first of these commands activated may prompt for a file name to which it should write the trace information. This file remains open until the **trace - - > nothing** command is given, which automatically closes the file. Just as in the case of **watch - - > nothing**, **trace - - > nothing** is the only way to deactivate the trace outputs once they have been activated.

5.5. Saving Knowledge Bases

Any knowledge base in the Manager list can be saved to the disk by selecting it and using the middle-button **save** command. This will write to the disk a file which HUMBLE can use at a later time or on another machine to recreate your knowledge base.

5.6. Listing Knowledge Bases

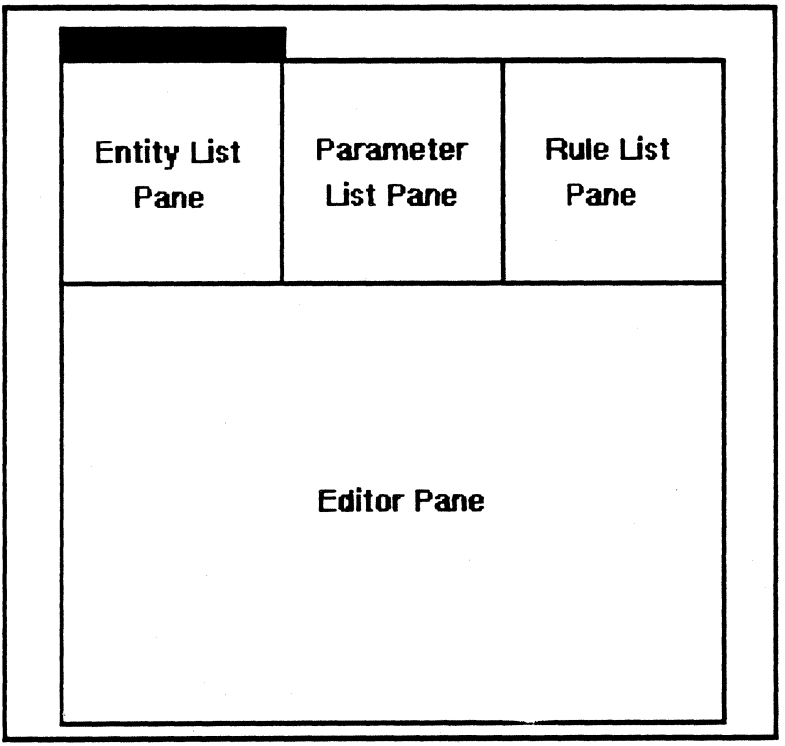
HUMBLE can produce listings of knowledge bases which are on the Manager window's list. To create a listing, select the knowledge base using the left button and then use the middle button **list** command. This will produce a disk file in ASCII called by the knowledge base's name followed by '.listing' (ROX would produce a listing called 'ROX.listing').

The listing provides a human readable file describing the knowledge base. The file lists all entity types, parameter definitions (with rule cross references), rules (with entity type cross references), and a set of metrics about the knowledge base which may or may not be interesting.

6. The HUMBLE Knowledge Base Editor

6.1. Window Description

A HUMBLE Editor window has four smaller areas within it which are traditionally called panes. Each of these panes controls some different aspect of the knowledge base. The diagram below shows the overall layout of the Editor window.



6.1.1. The Entity Type List

The entity type list pane contains a list of all types of entities which the knowledge base knows how to make inferences about.

By selecting one of the entity types in the list, a description is created in the editor pane. The description is directly editable. Read the section on "Entity Type Definitions" for a complete explanation of this description and how to change it. Another side effect of choosing an entity type from the list is that the parameter list now contains the list of parameters describing that entity.

There are two menu functions: **add new** and **remove**. Add new is used to create a template for a new entity type, which you can then fill in and accept. More information on editing is found a few sections further on, in "The Editor Pane". **Remove** removes the selected entity type from the knowledge base.

The **edit** command will bring the definition of the entity type back to the editor pane if it has been replaced by one of the parameters or rules associated with that parameter name.

6.1.2. The Parameter Definition List

The parameter definition list usually contains all of the parameters describing the currently selected entity type from the entity type list. Any particular parameter definition can be displayed by selecting it from the list. This will allow editing of the description. A side effect of selecting a parameter definition is to bring up a list of every rule which tests or makes a conclusion about the selected parameter. As is the case with the entity type pane, the command **add new** will bring up a template to fill in to produce a new parameter definition. The **remove** command will remove a parameter definition from the knowledge base.

The **definition** command can be used to redisplay the parameter definition in the editor pane. This will perform the same operations as selecting that particular parameter definition from the list.

The **show -> all** command circumvents the organization features of the editor window, and allows you to see, select, and edit every parameter definition for every entity type in a single

list. The **show -> ask first** command allows the user to see a list of all parameters of the selected entity type which have the prompt flag askFirst. As one might expect, **show -> ask last** and **show -> ask never** return lists of the parameters with prompt flags askLast and nil, respectively. The standard list (all parameters of the entity type selected) can be obtained by using the **show -> standard** command.

As with the Entity Type List pane, the **edit** command will bring the definition of the selected parameter to the editor pane.

The **graph** command will produce a view in the HUMBLE graphic utility of the backward chaining order of the deducing rules for this parameter.

6.1.3. The Rule List

The rule list usually contains all of the rules referring to the currently selected parameter definition from the parameter definition list. The text of the rule can be obtained by selecting one from the list. This will allow editing of the rule. Just like the entity type pane, the command **add new** will bring up a template to fill in to make a new rule. The **remove** command will remove a rule from the knowledge base.

The **show -> all** command circumvents the organization features of the editor window, and allows you to see, select, and edit every rule for every entity type in a single list. The commands **show -> referring** and **show -> deducing** will display the list of rules referring to the selecting parameter, or deducing it, respectively. **Show -> standard** will display the standard list which appears when a parameter definition is selected, all rules referring to or deducing the value of that parameter.

Two commands are supplied for searching the rules for numbers or strings. They are, quite predictably, **search for -> number** and **search for -> string**. They return the list of ruleseither referring to or deducing the value of the selected parameter definition which contain the number or string being searched for.

6.1.4. The Editor Pane

6.1.4.1. Basic Editing

This manual assumes you have been using Smalltalk for a while, so we won't go into too much detail on the finer points of editing. Suffice it to say that the editor pane of the HUMBLE knowledge base editor is a standard Smalltalk–80 text window, and that recognizable commands will work pretty much as you expect them to. If you have never been exposed to editing in Smalltalk, go and read chapters 1–3 of *Smalltalk–80: The Interactive Programming Environment* by Adele Goldberg. This will give you a basic familiarity with the Smalltalk–80 interface.

The following two sections will describe the specialized commands available in this window which will help you create and edit rules.

6.1.4.2. Rules – – >

One of the features included in HUMBLE is a class called TreeMenu, which allows menus which have sub–menus attached to the side. The middle button menu of the editor window is such a menu. There are two places where the user can slide to the left of his selection to get more selections, both of which end with the '– – >' string.

The **Rules – – >** menu has a list of templates for rule statements. Each of these is a blank form which you can use to fill in the overall form of some sort of statement in a HUMBLE rule. Essentially, they are the skeleton form of the main statements types in HUMBLE. They are mainly self explanatory.

Here is a list of them along with the text they produce

If then produces:

```
if: ( )
    then: [ ].
```

If then else produces:

```
if: ( )
    then: [ ]
    else: [ ].
```

If any then produces:

```
ifAny:
    has: [ ]
    then: [ ].
```

If all then produces:

```
ifAllOf:
    has: [ ]
    then: [ ].
```

If none then produces:

```
ifNoneOf:
    has: [ ]
    then: [ ].
```

6.1.4.3. Actions - - >

The **Actions - - >** submenu is used to fill in the action block of a statement. Each of the choices will produce a skeleton version of the action type specified. You can then easily fill in the skeleton and have a complete action. This section may not make much sense to you if you haven't read the chapter on "HUMBLE Rules". Just keep this section in mind when it comes time to create your own rules.

Conclusion produces:

<parameter> is: <aValue> withCertainty: <0.0>.

You are expected to replace *<parameter>* with a parameter name which exists in your knowledge base, *<aValue>* with either some literal or a parameter name, and *<0.0>* with whatever certainty you care to assign to the conclusion being made.

Forward chain produces:

{@ruleName}.

Simply replace *ruleName* with the name of one of your own rules to specify a forward chain.

Escape produces:

< < some Smalltalk code > >

Replace *some Smalltalk code* with some piece of Smalltalk code which performs some desired action. As noted in the chapter on "HUMBLE Rules", you can use parameter names just as if they were Smalltalk instance variables in these expressions.

7. The HUMBLE Graphic Utility

7.1. The Graphic Display

HUMBLE's graphic display follows a fairly common format. The display consists of a number of rectangular nodes representing rules and parameters in HUMBLE's static knowledge structure. Each node is connected to other nodes via arrows, which point in the direction of backward chaining.

So, as you may have guessed, the graphic display shows the path of traversal through the rules whenever some rule is to be executed or some parameter found out. Rules and parameters appearing toward the top of the graph will be executed/found first, with those lower (vertically, not structurally) on the display being executed later.

If you have a large graph, it may be of interest to spawn a smaller subset of the graph. This is easily accomplished by pointing at any node in the graph and using the middle button **graph** command. This will generate a new graph with the indicated element as the root node.

7.2. Editing Elements

HUMBLE's graphic utility allows editing of any node which appears. Selecting any node and using the **edit** command will generate a window exactly like the Editor Pane of the HUMBLE Knowledge Base Editor. It works in exactly the same way.

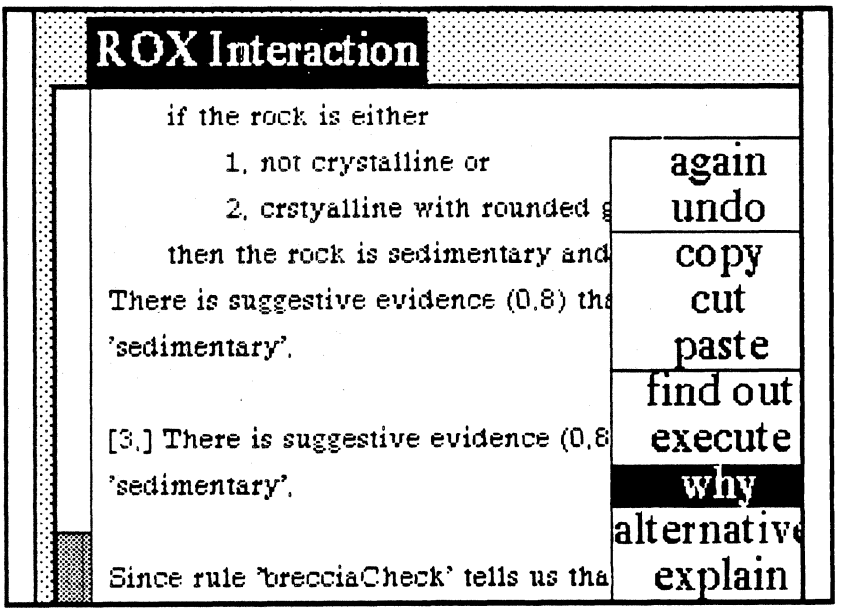
7.3. Testing your Knowledge Base(Find/Execute)

The **find/execute** command in the graphic utility provides a way to test parts of your knowledge base. By selecting a node and using the **find/execute** command, the user can cause HUMBLE to begin a backward chaining session right out of the graphic utility. It is probably a good idea to have a HUMBLE Listener window open at the time, otherwise you might not see much.

8. The HUMBLE Listener Window

8.1. Introduction

The HUMBLE Listener window is the basic interface by which a consultation can be run against a HUMBLE knowledge base. The style of interaction is based roughly on a transcript or listener window style, with the responses of the knowledge base being appended to the end of the transcript. The typical consultation involves little typing, instead relying on mouse selection as much as possible.



8.2. Beginning a Consultation

How a consultation is started depends largely on what sorts of entities and rules compose your knowledge base. If yours is mostly a backward chaining knowledge base, then the way to start a consultation is to type the parameter name you are interested in, select it with the mouse, and then use the middle button **find out** command. This will cause the knowledge base to ask the appropriate questions and return a result.

If the knowledge base is mostly forward chaining, there is probably a rule which starts the entire process. Type the name of this rule, select it with the mouse, and use the middle button **execute** command to fire that rule. The knowledge base will return a printout of the fact base (or working memory, if you prefer) when it is finished.

8.3. Explaining Results

The results of any consultation appear as a series of statements in the listener window. These statements are usually of the form:

There is <<strong, suggestive, poor, etc.>> evidence (<<certainty>>) that the <<parameter>> of <<entity>> is <<a value>>.

For instance, a Statement from ROX might be:

There is suggestive evidence (0.4) that the name of 'ROCK – 1' is 'gabbro'.

You can ask the system to explain any of these statements by selecting it and giving the middle button command **why**. You can select the entire statement or just that part that includes the entity and parameter names. For that matter, you can simply type the entity name (surrounded by ' characters) followed by the parameter name, select what you have typed, and give the **why** command (from the middle button menu, as usual). The phrase:

'ROCK – 1' name

would serve just as well as the entire statement above.

The two parts (entity name and parameter name) are all that is needed to uniquely identify the parameter you want to have explained.

8.4. Explaining Text

The meaning of a piece of text generated by HUMBLE can be partially defined using the **explain** command. This command scans the selected text for parameter names, then writes a table of the parameter names followed by their explanations (explanations are created in the editor window as part of the parameter definition).

8.5. Examining Alternative Conclusions

Alternative conclusions are often as important as the final answer given by a knowledge base. In HUMBLE, the **alternatives** command is used to generate a list of all alternatives for a given statement. To use the command, select a piece of text with the name of an entity and a parameter name, just like the selection needed to use the **explain** command. Then give the middle button **alternatives** command. This command will write to the listener a list of all the hypotheses made about that parameter, along with their certainties, in the usual form. For example, ROX might give the following set of alternatives.

There is suggestive evidence (0.4) that the name of 'ROCK – 1' is 'gabbro'.
There is weakly suggestive evidence (– 0.210526) that the name of 'ROCK – 1' is not 'granite'.
There is poor evidence (0.048) that the name of 'ROCK – 1' is 'diorite'.

Once the alternatives have been written, any of them can be explained by selecting them and using the **explain** command. Be sure to include the value in your selection, otherwise you will

receive an explanation of the most likely alternative (best hypothesis) rather than the one which interests you. A typed string such as

'ROCK – 1' name 'diorite'

would give the same explanation as

There is poor evidence (0.048) that the name of 'ROCK – 1' is 'diorite'.

when selected for use by the **explain** command.

8.6. Examining Entities

At any point you can select the name of any entity and use the **print it** command to get a printout of it. This command is used exactly as in any other Smalltalk window by the Listener, with the exception that is the selection happens to match the name of an entity it will write a printout of that entity to the Listener window. Otherwise, it will act exactly as the standard Smalltalk **printIt**, and attempt to compile and execute the selected text as Smalltalk – 80 code.

9. Example and Tutorial: Building a HUMBLE Knowledge Base

You have just read several chapters devoted to giving you the details of building a HUMBLE knowledge base, so now it's time for a general overview. We will go through the steps necessary to build a simple (very simple!) knowledge base on a rather silly subject, in order to integrate all of the information from the preceeding chapters into an integrated whole.

9.1. The Problem

It's always a problem deciding where to have lunch with a large group of people. Everyone seems to dislike some particular place. With a sufficiently large number of persons and choices for lunch, it can take most of the lunch hour just to decide where to go, let alone actually get there and be served. In response to this problem, one of the programmers here at XSIS suggested we build a small knowledge base to decide where to go to lunch, given who would be going along. This seemed like an ideal example, since the stucture of the rules wouldn't be very complex, but it would use a number of the main features of HUMBLE.

9.2. The Entities and Parameters

As noted earlier, the first thing to do is figure out what sort of entities and parameters are going to be needed for this knowledge base. The rules are probably going to be something like "If Kurt is going along, then we should go to Robin's". This looks simple enough, and already implies a number of things.

First we have the concept of Kurt going along. This means that we have some group going to lunch, which may have more than one person, and which may or may not have Kurt as one of its components. This probably implies that we have at least two sorts of entities: LunchParty entities and Person entities. A LunchParty is composed of Person entities. This is a nice, simple example of an entity tree one level deep.

Person entities, in the example, probably need to have a parameter called Name, since the prototypical rule uses the name to discriminate among persons.

Another concept suggested by the proto–rule above is that the LunchParty is eventually going somewhere. This is probably a case for having a parameter to hold this destination. Let's call it Destination.

Another sort of rule we may want to write is: "If Kurt is going along and it is Friday, then we should go to Spring Garden". This proto–rule suggests that we want to have some idea of what day of the week it is. Clearly, another parameter is useful. We'll call it DayOfTheWeek.

This looks like a good initial starting point for writing some real rules. The first step in creating the knowledge base is to get the Manager window on screen, then **add** a new knowledge base to it. Let's call the new knowledge base LUNCH. After adding the new knowledge base, select it and use the **edit** command to bring up a HUMBLE knowledge base Editor.

Now, let's define the entity types we have just discussed. Use the **add new** command in the entity list pane of the editor. This will bring up the usual template. Fill it in for each of the entity types we decided on, LunchParty and Person. My entity definitions looked like this:

LunchParty
typeAbove: nil
createPrompt: Are there any lunch parties planned?
addPrompt: Are there any other lunch parties?
assumePrompt: I am creating a lunch party
defaultName: PARTY
parameters: #()
mainParameters: #()

Person

typeAbove: LunchParty
createPrompt: Are there any persons going along?
addPrompt: Are there any other persons in the group?
assumePrompt: I am creating a person
defaultName: PERSON
parameters: #()
mainParameters: #()

Now that we have the entity types defined, we should add the parameter definitions for the parameters we discussed. Again, we use the add new command, but this time in the parameter definition list pane. My definitions looked like this:

Destination

describes: LunchParty
type: String
prompt: What is the destination of & ?
promptFlag: askLast
explanation: "The destination is the place to eat I should recommend"
remark: "Standard string parameter used to store the final destination of the lunch party"
changeBlock: [:parameter |]

DayOfTheWeek

describes: LunchParty
type: #('Monday' 'Tuesday' 'Wednesday' 'Thursday' 'Friday' 'Saturday' 'Sunday')
prompt: What day of the week is it?
promptFlag: askFirst
explanation: "DayOfTheWeek describes which day of the week the party is going out to lunch."
remark: "Enumerated string parameter for days of the week"
changeBlock: [:parameter |]

Name
describes: Person
type: String
prompt: What is the name of & ?
promptFlag: askFirst
explanation: "Name is just what you might think, the name of a person"
remark: "Standard string parameter holding person's name"
changeBlock: [:parameter |]

Name is so important to the setup we have here that it should be put on the main parameter list of the Person entity type. This insures that the system will immediately find out the name of the person, which it will certainly always need.

Not too bad so far, eh? Now that we have the entities and parameters defined, we can write some of the actual rules. We needed to set up the entities and parameters first, so that the rules would have something to talk about. The set we have described can represent the problem pretty well.

9.3. The Rules

The rules we considered earlier gave us a good framework on which to base our expert system. Now we can consider what an actual rule should look like. First of all, we should think about how we are going to express "If Kurt is going along" as a premise for a rule, in terms of the entities and parameters we have just defined. Now that we have a description of the entities involved, we would probably phrase our first proto–rule something like "If one of the Persons in the LunchParty has the name Kurt, then the Destination of the LunchParty is probably Robin's". This rule could be expressed in HUMBLE's rule language as:

Kurt
"If Kurt is going along then we should go to Robin's"

```
if: (anyOf: Person have: [Name = 'Kurt'])
  then: [Destination is: 'Robins' withCertainty: 0.6].
```

Now let's consider the second proto–rule, which says that "If Kurt is going along and it is Friday, then we should go to Spring Garden". We could add this consideration to the existing rule, ending up with something like:

Kurt

"If Kurt is going along then we should go to Robin's, or the Spring Garden on Friday"

```
if: (anyOf: Person have: [Name = 'Kurt'])
then: [
  if: DayOfTheWeek = 'Friday'
  then: [Destination is: 'Spring Garden' withCertainty: 0.7]
  else: [Destination is: 'Spring Garden' withCertainty: -0.1].

        Destination is: 'Robins' withCertainty: 0.6].
```

Notice that we have put an if–then–else clause, so that Spring Garden is a preferred destination only on Friday. This was not implied in the original proto–rule, but it seemed appropriate given my tastes. Lastly, we can assume that Kurt may have other preferences as well. You may have noticed that I gave Spring Garden a negative certainty on days which are not Friday. This is a good technique to use in general for these rules, for specifying places that the person the rule is describing does not wish to eat at. In fact, it will serve as the general system for ranking alternatives. Eventually, the rule might look like:

Kurt

"Kurt likes Robin's, Acapulco, Kabuki, Steer&Stein, Panda, or the Spring Garden on Friday. He dislikes Numero Uno, Yamaha, and most especially the Good Earth."

```
if: (anyOf: Person have: [Name = 'Kurt'])
then: [
  if: DayOfTheWeek = 'Friday'
  then: [Destination is: 'Spring Garden' withCertainty: 0.7]
  else: [Destination is: 'Spring Garden' withCertainty: -0.1].

        Destination is: 'Robins' withCertainty: 0.6 .
```

```
                Destination is: 'Numero Uno' withCertainty: -0.5.
                Destination is: 'Yamaha' withCertainty: -0.3.
                Destination is: 'Good Earth' withCertainty: -0.7.
                Destination is: 'Kabuki' withCertainty: 0.5.
                Destination is: 'Acapulco' withCertainty: 0.5.
                Destination is: 'Steer & Stein' withCertainty: 0.5.
                Destination is: 'Panda' withCertainty: 0.5].
```

Once we have developed one of these rules, we can see how virtually all of the others are going to go. We can create a separate rule for each and every person the system will know about. Get some friends together, substitute your own favorites for our restaurants, and then use the listener window to consult your new knowledge base. Here are a couple of other rules which were part of our local LUNCH knowledge base.

Cathy

"Cathy likes Hughes Market, Robin's, Steer & Stein, Acapulco, and Yamaha. She dislikes Spring Garden, Numero Uno, Good Earth, Fuddruckers, and Panda."

```
if: (anyOf: Person have: [Name = 'Cathy'])
  then: [Destination is: 'Spring Garden' withCertainty: -0.1.
         Destination is: 'Hughes Market' withCertainty: 0.75.
         Destination is: 'Numero Uno' withCertainty: -0.5.
         Destination is: 'Yamaha' withCertainty: 0.3.
         Destination is: 'Good Earth' withCertainty: -0.7.
         Destination is: 'Acapulco' withCertainty: 0.3.
         Destination is: 'Steer & Stein' withCertainty: 0.5.
         Destination is: 'Panda' withCertainty: -0.1.
         Destination is: 'Robins' withCertainty: 0.75.
         Destination is: 'Fuddruckers' withCertainty: -0.5.
        ].
```

Rae

"Rae likes Hughes Market, Robin's, Steer & Stein, Acapulco, Spring Garden, Reubens, Kabuki, Numero Uno, and Panda. He dislikes Yamaha, Good Earth, Fuddruckers."

```
if: (anyOf: Person have: [Name = 'Rae'])
  then: [ Destination is: 'Spring Garden' withCertainty: 0.5.
          Destination is: 'Hughes Market' withCertainty: 0.5.
          Destination is: 'Numero Uno' withCertainty: 0.5.
```

```
Destination is: 'Yamaha' withCertainty: -0.3.
Destination is: 'Good Earth' withCertainty: -0.9.
Destination is: 'Kabuki' withCertainty: 0.5.
Destination is: 'Acapulco' withCertainty: 0.7.
Destination is: 'Steer & Stein' withCertainty: 0.6.
Destination is: 'Panda' withCertainty: 0.5.
Destination is: 'Robins' withCertainty: 0.7.
Destination is: 'Reubens' withCertainty: 0.6.
Destination is: 'Fuddruckers' withCertainty: -0.5.
].
```

9.4. A Sample Consultation

Now that we have a knowledge base, let's run a sample consultation. To do this, we need to first get a Listener window open by using the **interact** command in the Manager window. This will produce a Listener window.

Since our knowledge base is geared toward finding out the value of Destination, this is what we should ask it for. Type the word 'Destination' into the Listener window, select it, and then use the **find out** command from the middle button menu. This will start up HUMBLE, which will begin to ask questions. Although these questions will appear as pop-up menus and confirmers, we will display them here in a transcript form. italic text indicates the questions and output from HUMBLE.

I am creating a LunchParty, which we will call PARTY-1
Are there any persons going along?
yes
I am creating a person, which we will call PERSON-1
What is the name of PERSON-1?
Kurt
Are there any other persons in the group?
yes
I am creating a person, which we will call PERSON-2
What is the name of PERSON-2?
Rae
Are there any other persons in the group?
no
Given all the evidence, I can conclude that,
There is strong evidence (0.88) that the Destination of

PARTY-1 'is 'Robins'.

Using the **alternatives** command yields:

There is poor evidence (0.0) that the Destination of PARTY-1 'is not 'Numero Uno'.

There is suggestive evidence (-0.75) that the Destination of PARTY-1 'is not 'Fuddruckers'.

There is strong evidence (-0.97) that the Destination of PARTY-1 'is not 'Good Earth'.

There is suggestive evidence (0.75) that the Destination of PARTY-1 'is 'Panda'.

There is strong evidence (0.85) that the Destination of PARTY-1 'is 'Acapulco'.

There is suggestive evidence (-0.51) that the Destination of PARTY-1 'is not 'Yamaha'.

There is suggestive evidence (0.75) that the Destination of PARTY-1 'is 'Kabuki'.

There is suggestive evidence (0.75) that the Destination of PARTY-1 'is 'Hughes Market'.

There is suggestive evidence (0.444444) that the Destination of PARTY-1 'is 'Spring Garden'.

There is strong evidence (0.88) that the Destination of PARTY-1 'is 'Robins'.

There is suggestive evidence (0.6) that the Destination of PARTY-1 'is 'Reubens'.

There is suggestive evidence (0.8) that the Destination of PARTY-1 'is 'Steer & Stein'.

9.5. A Refinement

If you have gone to the trouble of creating your own knowledge base, you will have noticed that it will always give the same answer for a given group. This is as it should be, given the rules you have already implemented. However, there is a way to refine it a little more.

Suppose we wanted to add a rule saying "If someone has eaten at a given place, he doesn't want to eat there again, so don't go there". We can implement such a rule by doing two things. First,

we must add a new parameter to Person, so that HUMBLE can keep track of where the person last ate. We will probably want to put this parameter on the main parameter list, so that the system always asks about the last place where this person ate. Let's call it LastDestination. My definition looked like this:

LastDestination
describes: Person
type: String
prompt: Where did &Name (&) last eat?
promptFlag: askFirst
explanation: "LastDestination describes where a person last had lunch"
remark:
changeBlock:

Notice that I used the value of Name as part of the prompt, so that the question could be more clear. Once we know that every person has some LastDestination, we can write a rule saying:

DontEatAtSamePlaceTwice
"if someone ate at the selected place before,
then that is a less likely choice"

ifAny: Person has: [Destination isKnown & LastDestination ~= "]
then: [Destination is: LastDestination withCertainty: -0.6].

This is a very powerful rule, since it will go through every person entity and make it less likely that their LastDestination will be today's Destination. This is also a recursive rule, so by definition it will execute only after all of the other evidence is in.

This relatively simple addition will greatly modify the behavior of your expert system, and make it a lot more useable.

10. The HUMBLE Programmer's Interface

10.1. Introduction

This chapter is probably of very little interest to anyone but a Smalltalk programmer. If you don't feel that you fit into this category and don't plan to ever be a Smalltalk programmer, then by all means ignore this chapter. I assure you, I won't even feel hurt. As long as you don't tell me.

It is very seldom the case that a given expert system shell will have a particularly good user interface for any particular problem. In light of this, The design of HUMBLE includes a vast array of messages usable by outside programs, which will allow you to completely replace the standard humble window set with your own custom interface.

"Oh no," you say. "I'd have to be a Smalltalk programmer who understands the inner structure of HUMBLE to do that." Wrong. Sort of. You will need to be a Smalltalk programmer. It would be helpful to understand the internal structure of HUMBLE, but it is not absolutely necessary. HUMBLE was designed to allow programs other than the standard ones to access HUMBLE's logic engine for their own nefarious purposes. This chapter will attempt to provide enough working understanding to enable you to utilize HUMBLE from within your own programs.

It should be noted here that the code of HUMBLE has been copyrighted by Xerox, and that even though it is provided to you for your use, Xerox will jealously guard its rights to it. Therefore, if you use any of the code in any way in any part of your program or programs, you will certainly have to come to some agreement with Xerox about distribution.

10.2. Required Reading

If you are going to attempt to use this chapter, let me recommend that you either read or be familiar with:

A. Goldberg. *Smalltalk – 80: The Interactive Programming Environment.* Addison – Wesley, Reading, MA 1984.

A. Goldberg and D. Robson. *Smalltalk – 80: The Language and its Implementation.* Addison – Wesley, Reading, MA 1983.

Also of interest, though not required:

Buchanan and Shortliffe (eds.). *Rule Based Expert Systems: The MYCIN experiments of the Stanford Heuristic Programming Project.* Addison – Wesley, Reading, MA 1984.

The Goldberg books are essential to doing even simple programs in Smalltalk. The knowledge of how to do simple Smalltalk applications is essential if you want to make any use of this section. The assumption made in this chapter is that you are a fairly accomplished Smalltalk – 80 programmer, capable of producing windows, using files, and have a good overall understanding of the basic Smalltalk system classes.

Since HUMBLE is in many respects similar to MYCIN, the Buchanan and Shortliffe book is fine general reading on this type of expert system.

10.3. The KnowledgeBases Global

KnowledgeBases is a system global. In other words, you can call it by name and send messages to it. There are a number of useful messages it will respond to, and you can find out exactly how they work by looking at class **HumbleManager** in class category **XSIS – HUMBLE – Manager**. However, to save you a lot of effort, I'll explain a few of them right away.

Far and away the most useful of these messages to the programmer is the 'baseNamed:' message. An example of its use might be:

kb ← KnowledgeBases baseNamed: 'ROX'.

In this example, the variable kb is set to be a knowledgebase named 'ROX', which KnowledgeBases has returned in response to the message. Once you have a knowledge base in a variable, you are ready to send it a variety of messages. We'll discuss those messages a little later.

Some other useful messages are:

allBases – return a collection of all known knowledge bases in the System

fileInBase – KnowledgeBases will prompt you for a name, then load the named knowledge base from disk.

edit: – by passing a knowledge base name (string) as the argument, the system will open a standard editor on the knowledge base with that name. i.e. KnowledgeBases edit: 'ROX'.

interactWith: – by passing a knowledge base name (string) as the argument, the system will open a standard listener window on the knowledge base with that name. i.e. KnowledgeBases interactWith: 'ROX'.

remove: – by passing a knowledge base name (string) as the argument, KnowledgeBases will remove the knowledge base with that name from the system. i.e. KnowledgeBases remove: 'ROX'.

As you can see, the KnowledgeBases global is a good general way to interact with knowledge bases by name. It is suggested that any interactions you may perform with knowledge bases as objects be done through the agency of this global, just so that everything remains organized and accessible.

10.4. Individual Knowledge Bases

Each individual knowledge base has a number of tailorable parameters which make it flexible and easy to interact with.

The exact code can be found in class **KnowledgeBase**, in class category **XSIS – HUMBLE – Interpreter**. The complete description of the interactions involved will take several sections. Each one will hopefully cover some area of interest.

10.4.1. Executing the Rules

As HUMBLE is a rule–based system, there is probably some utility in knowing how to execute rules in HUMBLE from outside programs. HUMBLE supports both forward and backward chaining, and the method by which chaining is invoked is different for each chaining direction.

10.4.1.1. Backward Chaining

Backward chaining occurs in a relatively automatic fashion at all times during HUMBLE execution, whenever a reference to a Parameter is made. This process can be initiated by sending the message **findOut: aParameterName** to a knowledge base. The knowledge base will find the parameter definition with the corresponding name, create any needed entities, and then use backward chaining to discover the most likely value for that parameter. It will finally return the actual parameter created for you to examine within whatever object is sending the message.

When the knowledge base object receives this message, it does not automatically reset the knowledge base's fact base. This is to handle the case where an expert system needs to make further inferences about a given set of information. To reset it, send the message **initEntities** to the knowledge base. To reset the counters used to make up default entity names, send the **initEntityTypes** message to the knowledge base.

It is instructive to look at how the listener window interacts with its knowledge base.

10.4.1.2. Forward Chaining

As mentioned in the chapter on HUMBLE rule syntax, HUMBLE rules can forward chain by simply adding a forward chain

construction to some part of a rule. However, it may not be obvious how to start execution on such a knowledge base.

A knowledge base can be sent the message **rules** which will return the collection of rules in the knowledge base. This collection is a dictionary, which will return rules by name to the programmer. That rule can then be sent the message **execute**, which will fire the rule. For example, lets say you have created a knowledge base named 'Rhombus', which can be started by executing the rule named 'startUp'. The code to begin execution of this set of rules might look like:

((KnowledgeBases baseNamed: 'Rhombus') rules at: #startUp) execute.

Notice that #startUp is a Symbol, not a string. The knowledge base uses Symbol instances as keys for efficiency's sake.

Typically, someone interested in starting up a knowledge base in this fashion has his initial rule simply fire off a number of other rules in sequence. Remember if you create a totally forward chained set of rules you still may have an unusual order of execution, since HUMBLE tries to backward chain at all times. Properly written forward chained rules, though, should never have any ambiguities which would induce backward chaining. Use only **askFirst** parameters if you feel compelled to use purely forward chaining, to insure that backward chaining never occurs other than to obtain parameter values.

10.4.2. Inspecting the Entities

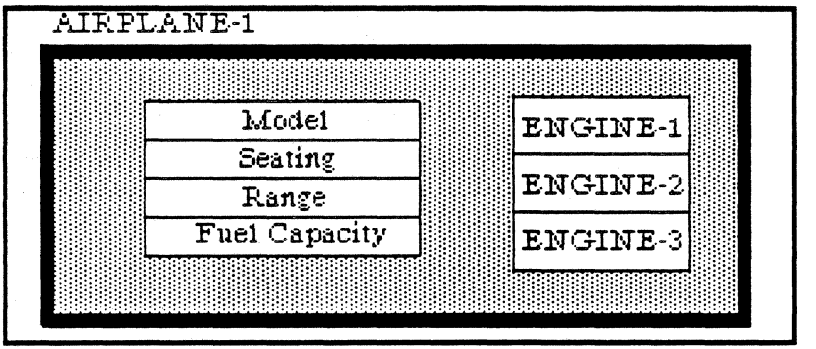
Obviously, if you can't inspect the fact base a knowledge base produces, then it is of very little interest. HUMBLE allows inspection of its fact base at every level. Since the fact base data

structure is quite complex, though, it there are a fairly large number of methods which must be explained.

10.4.2.1. Examining Entities

Entities, as mentioned in earlier sections, are the objects about which HUMBLE reasons. In many respects, they resemble a subroutine execution context as well, and in fact are referred to as contexts in many expert systems of similar architecture. The basic structure of an entity includes two major elements, a list of parameters and a list of sub – entities.

The message **primaryEntity** is understood by members of class KnowledgeBase, and it returns the topmost entity in the fact base. This single entity is the key to all of the other entities in the fact base, since by definition all other entities are a part of it.



10.4.2.1.1. Sub – Entities

Once you have the primary Entity, it is very likely that you will want to 'walk the entity tree' to find the specific entities you are interested in.

One can access the sub – entities of an Entity by a number of techniques. The simplest is to send an entity the message **allSubEntities**. This will return an OrderedCollection containing all of the sub – entities, of whatever type, the receiver contains.

Since every Entity has a type, it may also be of interest to ask for only certain types. **allSubEntitiesOfType: aString** will return an OrderedCollection of only those subEntities with a given type. Note that this message may well return only an empty collection.

Each Entity has a list of parameters, and it may be of interest to search for entities which have a certain parameter name among their parameters. **allSubEntitiesWithParameter: aString** will return an ordered collection with all of the appropriate sub–entities.

10.4.2.1.2. Accessing the Parameters

The primary method of accessing a particular parameter is to send an Entity the message **parameterNamed: aSymbol**. This will return the actual parameter object for scrutiny. See the later section on examining parameters for more information on what information is available in a given parameter.

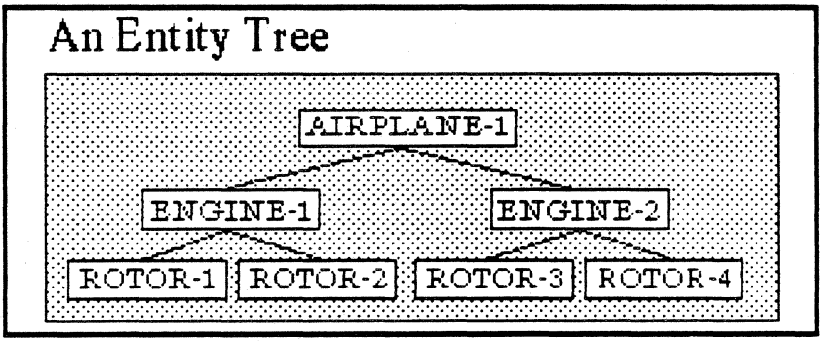
The entire set of parameters can be returned as a dictionary by sending any Entity the message **parameters**. The keys of the dictionary are the names of the parameters contained within.

10.4.2.1.3. Accessing the Entity Type

Every Entity has a pointer to its EntityType contained in the instance variable type. The message **type** will return it for examination. There are numerous pieces of information in the EntityType which might be of interest.

The prompts associated with entities of this type can be returned by the messages **addPrompt**, **assumePrompt**, and **createPrompt**.

The EntityType above this one in the entity tree can be retrieved by sending an EntityType the message **typeAbove**. The types below can be retrieved as an OrderedCollection by the message **typesBelow**.



10.4.2.1.4. Other Useful Information

Some other things which might at some point be of interest are the messages **conclusionSet** and **superEntity**. The **conclusionSet** message will return an OrderedCollection containing the names of all the rules which have executed in the context of this Entity. The other message, **superEntity**, will return the entity above this one in the fact base's entity tree.

10.4.2.2. Examining Parameters

The Parameters of an Entity are where the real information about that entity lies. A parameter is a relatively complex sort of object, certainly far more complex than a simple variable. As mentioned in the previous section on "Looking at Parameters", you can get to an individual parameter by sending the message **parameterNamed: aSymbol** to an Entity. The entire set can be obtained by sending the message **parameters** to an Entity.

Once you have the parameter you are interested in, you are probably interested mainly in the best hypothesis about that parameter. The message **bestHypothesis** will return it to you directly. The **value** message will return the value of the best hypothesis, **certainty** returns its certainty, and **reason** returns its reason.

10.4.2.2.1. Accessing the Hypotheses

Examining the set of hypotheses which has been asserted about a parameter is a relatively complex activity. A member of class Parameter holds all of its hypotheses in a dictionary structure. The keys of this dictionary are the values of the hypotheses asserted. The dictionary values corresponding to those keys are themselves dictionaries, with keys #combined and #all. The value in #combined is the overall level of certainty associated with that value. The values keyed by #all are OrderedCollections, each of which holds all hypotheses with value similar to the dictionary key in chronological order of assertion.

This, of course, makes very little sense when described verbally. Here is a printout of a typical case from the ROX knowledge base. Executing the code –

((KnowledgeBases baseNamed: 'ROX') primaryEntity parameterNamed: #name) hypotheses

might yield:

```
Dictionary (
    'granite'– >Dictionary (
        combined– >'granite' (0.939968)
        all– >OrderedCollection (
            'granite' (0.4)
            'granite' (0.56)
            'granite' (0.56)
            'granite' (0.32)
            'granite' (0.24) ) )
    'diorite'– >Dictionary (
        combined– >'diorite' (0.65728)
        all– >OrderedCollection (
            'diorite' (0.4)
            'diorite' (0.32)
            'diorite' (0.16) ) )
    'gabbro'– >Dictionary (
        combined– >'gabbro' (–0.64)
        all– >OrderedCollection (
            'gabbro' (–0.4)
            'gabbro' (0.56)
            'gabbro' (–0.56)
            'gabbro' (–0.4) ) ) )
```

In this case, three possible values for parameter name have been asserted, and each one has been asserted several times. The Dictionaries contain sets of similar hypotheses, keyed by their values. Each dictionary contains a combined certainty and an OrderedCollection of hypotheses. Each OrderedCollection contains a chronological history of the assertions made. In this example, both 'granite' and 'diorite' became more certain, while 'gabbro' became less certain as rules were executed. See the section on "Examining Hypotheses" to get more information on what can be examined in the individual hypotheses.

10.4.2.2.2. Accessing the Parameter Definition

A ParameterDefinition contains a wealth of information about a specific parameter. By sending any parameter the message **definition**, the programmer can obtain the parameter definition corresponding to that parameter. Once you have the ParameterDefinition object, you can send it a number of different messages:

changeBlock – return the block of code this parameter should execute when its best hypothesis has changed. This is not meant for use by programmers from outside the knowledge base.

deductionRules – return the list of rules this parameter will fire if it must deduce its own value.

entityType – return the entity type of which this parameter is a part.

explanation – return the explanation string associated with this parameter. This is now used by the explain functon of the listener window.

knowledgeBase – return the knowledgeBase of which this parameter definition is a part.

parameterName – return the parameter name, the key under which this type of parameter is stored in an entity.

prompt – return the string used to create a prompt when this parameter must ask its value from the user.

promptFlag – return either #askFirst, #askLast, or nil. This value flags the order in which the inference engine attempts to deduce or ask the value of a parameter.

referringRules – return a list of all rules which test this parameter.

remark – return this parameter definition's remark string. Used as a comment for parameter definitions at present.

type – return what sort of parameter this is, one of: class String, class Number, #YN, #MV.

10.4.2.3. Examining Hypotheses

Each individual hypothesis has a certain amount of limited intelligence about itself which may possibly be of interest to the programmer. It is most likely that the value of the hypothesis is of primary interest, and sending a hypothesis the message **value** will return this value to you.

In some cases, certainty values are also useful information, and the message **certainty** will return whatever certainty the hypothesis has to you.

The last message likely to be of interest is the **reason** method. During execution, HUMBLE rules tell any hypotheses they compute or produce that the reason for the hypothesis is a symbol corresponding to the name of the executing rule. This symbol is used to produce the explanations from the listener window. The section on "Explaining Conclusions" explains what message accomplishes explanations, and it will not be explained here. However, it may be an instructive part of the code to examine for those wishing to create their own explanation facilities.

10.4.3. Resetting a Knowledge Base

In order to handle the case where an expert system needs to make further inferences about a given set of information, the KnowledgeBase will not automatically reset whenever it is queried. If you are going to use a knowledge base with new data, it is your responsibility to reset the knowledge base to clear out old information.

To reset a knowledge base's fact base, send the message **initEntities** to the knowledge base. To reset the counters used to make up default entity names, send the **initEntityTypes** message to it. The standard code used in HUMBLE (using the old standby example 'Rhombus') is:

(KnowledgeBases baseNamed: 'Rhombus') initEntities; initEntityTypes.

10.4.4. Explaining Conclusions

The message **explain: aParameterName in: anEntityName value: value**, when sent to a knowledge base, will return a string containing an explanation of the reasoning behind the assertion that the parameter named **aParameterName** in the entity named **anEntityName** has the value **value**.

The explanation generator chains backwards to explain the reasoning from the simplest data elements involved. This is the message that the Listener window uses to generate its explanations.

This message is also an excellent example of how to go about extracting information from the knowledge base. It may be instructive to look at this message as an example of the techniques discussed in the section on "Inspecting the Entities".

10.4.5. Examining Alternatives

The message **alternativesFor: aParameterName in: anEntityName** will return a string with the printed form of all the hypotheses created for that specific parameter. The Listener window uses this message to generate the alternatives for some parameter selected. Any application can take advantage of the same capacity.

10.4.6. Redirecting Output

Output from a HUMBLE consultation may be directed to any Stream in Smalltalk – 80. To do this, just use the **outputStream:** message. For instance,

```
(KnowledgeBases baseNamed: 'ROX') outputStream:
  (FileStream newFileNamed: 'ROX.output')
```

will direct the output of the knowledge base named 'ROX' into a disk file called 'ROX.output'. There are a number of uses for this facility which come to mind. For instance, a Listener window similar to the standard HUMBLE might display any output directly to the screen. By sending output to a file, it is possible to easily capture explanations or printouts of entities in a knowledge base into disk files for later use or auditing.

The Listener window is a nice example of how to hook the output of a knowledge base to a TextCollector subclass. (TextCollectors, by the way, are the Smalltalk equivalent of a transcript window. The system global Transcript is one.) As is usual, you are urged to go and read the code in the classes in class category **XSIS – HUMBLE – Listener** to see some example code.

10.4.7. Redirecting Input

Input to a HUMBLE knowledge base can also be redirected. When a HUMBLE knowledge base needs to ask the user a question, it uses an intermediary object called an **Interrogator** to ask the question.

On the surface, this simply means that the knowledge engineer can create his own tailored questioning interface, but there are deeper implications. Since there are essentially no constraints on exactly what an Interrogator does, it can even avoid asking the user questions altogether. Instead of asking questions, it could read a database or data structure to glean answers. Some combination is possible. The possibilities are limited solely by the programmer's imagination and skill with Smalltalk.

I suggest making whatever class of interrogator object you need a subclass of class Interrogator. Look carefully at each of the messages implemented in class Interrogator and rewrite them in the subclass to do what you wish. In many cases, you will be able to leave them exactly as they are.

When you have a new type of interrogator ready to be plugged in, use the **interrogator:** message in class KnowledgeBase to tell your knowledge base what object to use to make interrogations.

10.4.8. Tracing Execution

A trace facility exists in HUMBLE, which will write trace information to a stream specified by the **annotationStream:** message. This points the knowledge base to a stream either in memory or a disk file. The system then notes the state of the traceFlags instance variable, and writes to the file based on its state.

There are three possible types of events that can be traced: goal creation, rule firing, and conclusion creation. The messages which turn tracing on or off for these events are:

traceGoals: aBoolean "trace any goal's creation if aBoolean is true"
traceRules: aBoolean "trace any rule's execution if aBoolean is true"
traceConclusions: aBoolean "trace any conclusion if aBoolean is true"

Note that these messages are used both to activate and deactivate their respective types of tracing.

10.4.9. Initializing the Entities

In many cases, it is desirable to initialize the entities in a knowledge base before inferences are made, simply because much of the information is already known. Rather than asking the user, it is much more desirable to have the information ready

and waiting when rules ask for it. This is, unfortunately, not as simple a task as some of the other ones we have discussed.

The difficulty is that the structure of the fact base in HUMBLE is rather complicated. In order to help overcome this, a special message has been provided which will set the parameters of an entity from a dictionary which has been created in some fashion. The following example uses this message, called **setParamsFromDict:** to initialize a knowledge base.

```
| rox rock dict |
"obtain the knowledge base from the
    KnowledgeBases global"
rox ← KnowledgeBases baseNamed: 'ROX'.

"reset the entities, then create a new blank entity.
    The createInstanceForInjection message produces a new

    entity, but does not automatically attempt to find out
    the main parameters."
rox initEntities; initEntityTypes.
rock ← (rox entityTypes at: #Rock)
        createInstanceForInjection.

"build a dictionary of field names (as symbols) and values.
Notice that parameter class is receiving an uncertain value,
while the rest are absolutely certain."
dict ← Dictionary new.
dict at: #class put: (Hypothesis new: 'igneous' certainty: 0.7).
dict at: #color put: 5.
dict at: #fossiliferous put: false.
dict at: #calcareous put: false.
dict at: #porphyritic put: false.

"initialize the blank entity from the dictionary"
rock setParamsFromDict: dict.

"initiate a consultation"
rox findOut: #name in: rock.
```

10.5. Altering a Certainty Model

As advertised, HUMBLE has a modular certainty system which can be altered to suit the user's needs. Essentially, there are eight messages to produce which define how HUMBLE will deal with certainty. The first step is to create a subclass of class Hypothesis, which is the class which defines certainty and other behaviors for data in HUMBLE.

Once you have a subclass produced, you must define several new methods for the subclass, those dealing with certainty. The next two sections will describe exactly what messages need to be defined. One can create any of a number of differing certainty models as needed. The actual implementation of any model is left as an exercise for the reader.

Once the subclass has been completed, it remains to tell the knowledge base that it should use a non-standard certainty model. This can be accomplished by sending a knowledge base the message **hypothesisType: aHypothesisSubclass**. The argument should be that actual class, not an instance of it. From then on, even when it is saved to the disk, the knowledge base will automatically use the new class, with its new certainty model, for all calculations.

The next two sections cover the messages you will need to implement in two groups of four. The first deals with the operations on certainty factors, the second with the constants associated with your model.

10.5.1. Combination Messages

HUMBLE expects instances of your Hypothesis subclass to be able to respond to the following messages:

combineCertaintyWith: aHypothesisCollection
This message should set the certainty of the receiver to be the combined certainty of the receiver and aHypothesis. This is the message used when HUMBLE must combine the results of two or more inference chains which lead to a similar conclusion.

The hypothesis collection which is passed is the standard parameter hypothesis dictionary. (see Section 10.4.2.2.1: Accessing the Hypotheses) You should feel free to add new structure to this dictionary if you desire, but be certain to leave the existing structures intact. The present structure contains information about every hypothesis for a parameter, so it should be possible to add just about any sort of certainty model you desire.

maxCertaintyOf: aHypothesis and: anotherHypothesis
This message should return a value equal to the certainty the most certain hypothesis, aHypothesis or anotherHypothesis.

minCertaintyOf: aHypothesis and: anotherHypothesis
This message should return a value equal to the certainty the least certain hypothesis, aHypothesis or anotherHypothesis.

productCertaintyOf: aHypothesis and: anotherHypothesis
This message should return a value which reflects a certainty of which both aHypothesis and anotherHypothesis partake. In the present certainty model, this is used to determine the certainty of the result of a binary operation. It does this by multiplying the two certainties together.

10.5.2. Certainty Constants

HUMBLE also expects your Hypothesis subclass to be able to respond to the following messages on both the class and instance sides:

maxCertainty
return the value associated with absolute positive certainty

minCertainty
return the value associated with absolute negative certainty

cutoffCertainty
return the certainty value at which it is considered safe to declare that the a parameter has this value. This is the cutoff

point above which a rule will decide to activate its action block.

cutoffNegativeCertainty
return the certainty value at which it is considered safe to declare that the parameter does not have this value.

unknownCertainty
return the certainty value which deontes a total lack of evidence one way or the other.

11. Using HUMBLE in Popular Expert Systems Architectures

11.1. Simulations

Simulations are one of the primary areas in which HUMBLE was designed to be useful. A typical simulation program has areas where rule based behavior could be useful, but these are generally hardcoded into the simulation code since the work necessary to implement a rule based system is often quite extensive. Smalltalk has long been known as a good language in which to write simulations, since an object oriented style is well suited to discrete simulations, and it is then very easy to extend the simulations by subclassing.

HUMBLE includes a tutorial example, MazeMaster, which highlights the methods required to hook rule based behavior into a Smalltalk simulation.

11.1.1. MazeMaster: a Humble Simulation Tutorial

MazeMaster is a tutorial program designed to help you understand how to hook HUMBLE into simulations in Smalltalk–80. The simulation in MazeMaster is a simple mouse and maze simulation, with a mouse trying to run around a small maze, avoid running into the cat, and finding and eating the cheese.

What makes MazeMaster interesting is that the mouse can be attached to a HUMBLE knowledge base, which will serve as the mouse's 'brain'. Altering rules in the knowledge base will alter the behavior of the mouse. Be aware that the 'brain' in the HUMBLE knowledge base is very simple, and in fact using HUMBLE at all for this is a fantastic level of overkill (and overhead). A simple left–hand–wall algorithm for maze navigation could be implemented in the mouse trivially and be many times faster. Please remember, though, that this is a tutorial, and meant to avoid burdening you with a complex

simulation and massively intelligent mouse when the real point is how to get HUMBLE hooked into a simulation.

11.1.2. The MazeMaster Object Classes

MazeMaster defines three new object classes: Maze, Mouse, and MazeMaster. Each defines a limited part of the maze simulation.

Class Maze, as might be expected, describes the shape and contents of the Maze in which the mouse will maneuver. It is essentially a two dimensional array of locations, each of which has some sort of contents. The possible contents are 'wall', 'open', 'cat', and 'cheese'. The maze knows how large it is, and can tell any outside object the contents of some location in the maze. The maze also knows how to display itself.

Class MazeMaster describes the objects which know how to run the simulation. A mazemaster knows which maze and mouse will participate in the simulation, and contains the basic rules of the simulation and a timing clock. The rules are quite simple:

1. If the mouse is in the same location as a wall
   he has slammed into the wall and killed himself.
2. If the mouse is in the same location as a cat
   he has been eaten.
3. If the mouse is in the same location as a cheese
   he can eat the cheese and is declared to have won.
4. If one hundred clicks of the clock have passed without
   the mouse eating the cheese, the mouse starves to death.

The **run** and **runKB** messages in class MazeMaster run the clock and enforce the rules of the simulation.

Class Mouse is the most complex of the three classes. In its simplest form, a mouse knows how to be told where to move, how to look forward, backward, to the left and right. The mouse is quite myopic, and can only see one space in any direction. A mouse knows how to ask a user where it should move next. It also knows how to display itself. Class Mouse also has a set of methods which allow it to be attached to a HUMBLE knowledge base rather than asking a user for instruction.

11.1.3. The MazeMaster Connection

Class Mouse is capable of being hooked into a HUMBLE knowledge base. To do this, the instances of the class all know how to act as a proper HUMBLE Interrogator object. Although not a subclass of class Interrogator, class Mouse has all of the necessary protocols to act just like one. To Smalltalk – 80, this is just as good as being the real thing.

Three messages had to be defined in order to pull off this charade. Each mouse had to know how to respond to the messages **oneOf:**, **moreOf:**, and **request:**. These are the three messages by which a HUMBLE knowledge base interacts with its interrogator object. Notice the difference between these messages in class Mouse and their equivalents in class Interrogator. The message **oneOf:** is how HUMBLE asks if any of a single type of entity exists in the real world. Since we are dealing with a limited case in MazeMaster, where there is always one and only one mouse, this can always return true. Similarly **moreOf:** will always return false, since there will only be one mouse at any time.

The message **request:** was a little more difficult. The **request:** message is used by HUMBLE to ask for the value of a parameter, which is passed as an argument to **request:**. This parameter always has a definition, which contains the parameter's name. Therefore, the easiest method of returning all needed parameters was to make a message for each parameter name, and then tell the mouse to execute each message whenever the knowledge base asked for its value. Thus, there are a set of methods implemented in class Mouse which correspond exactly to the names of the parameters for the Mouse entity type in the knowledge base named 'Mouse'. The code of the **request:** message therefore just looks up the parameter name, turns it into a symbol, and then tells itself to perform the method with that symbol as its selector. This is all a lot simpler than it sounds right now, so don't be alarmed.

11.1.4. What We Were Trying to Say

The point of providing this example was to make clear how to connect HUMBLE's knowledge base objects to other working systems in Smalltalk–80. The idea was *not* to suggest that HUMBLE knowledge bases are the perfect way to control all aspects of a simulator. Instead, they are probably most useful when a simulator has 'fuzzy' aspects to it, that is to say that elements of the simulation have limited or perhaps incorrect data being passed to them. Only in such cases will the overhead of adding a HUMBLE rule based system to the simulation be justified.

11.2. Blackboard Systems

A blackboard system is a way of coordinating a number of seperate knowledge bases and other sources of information, thus using them all to help solve a particular problem. The blackboard serves as a central repository of information, which many different sorts of problem solving units can access. HUMBLE knowledge bases can be easily attached to a blackboard system built in Smalltalk–80.

11.2.1. The Blackboard Concept

A blackboard system is generally composed of a number of parts, which can be devided into three main categories: the 'big cheese', the blackboard, and the knowledge sources. Each category complements the others, allowing a modular and convenient approach to solving complex problems.

The 'big cheese' is the master control section of the blackboard system. The cheese selects which knowledge source will now attempt to act on some information on the blackboard. Often, it will query the knowledge sources about whether they think they can help produce a solution given the knowledge present on the blackboard. The cheese declares the problem solved or unsolvable, and therefore always ends the problem solving session.

The blackboard itself is simply a clearinghouse for information. All of the knowledge sources have some access to the blackboard, it serves as a source of inputs and outputs to the various knowledge sources. The blackboard is often arranged in a hierarchy, although the reasons for this are not clear in most papers published about such systems.

The 'knowledge sources' can be literally anything. One knowledge source might be a human operator, another a large database, yet another a knowledge base, and still another a simple computation routine. Anything that can provide input into the solution process is a knowledge source.

This structure has proven highly successful in cases which require varied approaches in different parts of the problem. It has the disadvantage of not being as profficient at explaining itself as a pure rule based system, but is far more computationally flexible. As may be obvious, object–oriented systems are almost ideal for building blackboard systems. HUMBLE knowledge bases can easily serve as knowledge sources in a Smalltalk based blackboard system.

11.2.2. HUMBLE Knowledge Bases as Knowledge Sources

HUMBLE knowledge bases can be attached to a blackboard system in much the same way as they can be attached to a simulation (In fact, a simulation can be viewed as a special case of the blackboard idea). The steps to accomplish this attachment are as follows:

1. Build an interrogator which can read the blackboard. This should pose no particular problem, being almost exactly the same sort of procedure as was used in the MazeMaster system.

2. Build an interface which can take the output of a HUMBLE knowledge base and insert it into the blackboard. This is more interesting. The case in MazeMaster was simple, since we were going to immediately apply the decision the brain knowledge base reached in the simulation. This will not always

be the case in a general blackboard system. The best way to get the information back to the blackboard is to have the big cheese take the results, perform the insertion, and then use the fact that it has just received new data to select the next knowledge source.

Further, the ability of each HUMBLE knowledge base to have a separate certainty model, unique to the problem it is trying to solve, make HUMBLE knowledge bases a particularly flexible addition to a blackboard system.

11.3. Smalltalk–80 and Frame Based Systems

Frame based systems are a popular method of building large and complex expert systems. They provide a number of advantages which strict rule–based systems have been unable to provide. HUMBLE knowledge bases can be added to a Smalltalk–80 based frame system with relative ease, using techniques similar to those used to attach them to simulators and blackboard systems.

11.3.1. The Frame Concept

Now, the biggest surprise of all. Smalltalk–80 is a frame based system already! Really.

If you take a look at the concept of frames and slots, you'll find that they very much resemble objects and instance variables. Building a frame based system for Smalltalk–80 is somewhat like building a basic arithmetic package to run in Fortran–77. You simply don't need to do it. Objects and frames are literally almost the same idea. The minor differences in the systems are usually matters of emphasis. Smalltalk–80 tends to emphasize general programming, rather than convenient shorthands for expert system creation.

While the emphases are different, it is relatively easy to provide a number of the most important conveniences for yourself. One of the ones that is most useful is a global name table for objects you are using as frames. Smalltalk provides a convenient global name table, the object named Smalltalk (as opposed to the entire system, which is also generically called Smalltalk). This object,

which is a member of class SystemDictionary, is where the system looks up the names of classes and certain special objects in the system. By adding a dictionary to Smalltalk, we can get a special name table for frames which we can clear out as needed without disturbing the rest of the system. The code might look like this:

Smalltalk at: #Frames put: Dictionary new.

Whenever a new frame is created, simply execute code like the following to add it to the name table:

Frames at: frameName put: newFrame

where newFrame and frameName are the new frame and its name, respectively. This sort of dictionary, the Smalltalk system dictionary, and pool variables can all provide fast access to frame objects in any frame based system you are building.

Other aspects of the system can be convenient as well. A typical activity in a frame system involves changing the class of a frame as more information is gained. This is most easily performed in Smalltalk using the **become:** message. Be careful, though, since the **become:** message cuts both ways. If you tell something to **become:** nil or tell nil to **become:** some other object, you will almost certainly crash your system. This is generally considered undesirable.

11.3.2. Humble Knowledge Bases within Frames

Since HUMBLE knowledge bases are objects just like anything else in Smalltalk, they can easily occupy slots in any frame system you create. Thus, it is easy for a particular frame to keep a rule–based system available for its internal use. Also, many frames could share the same knowledge base. HUMBLE knowledge bases could then be used:

1. As a decision mechanism for deciding the next frame to examine.

2. As a way of filling in slots in the frame. Results of HUMBLE knowledge bases could be used to fill in slots in frames as needed.

Further, various parts of HUMBLE, such as class Hypothesis, are perfectly capable of being used as part of a frame mechanism, since they don't directly depend on being part of a HUMBLE knowledge base for their operation. This could allow the easy creation of a system to store uncertain values within frames, given a little ingenuity.

Appendix A. Glossary

Term	Definition
Knowledge Base	The place where the expertise which composes an expert system is stored by HUMBLE.
Fact Base	The part of a knowledge base which stores temporary information. Information which is useful only during the present consultation resides in the fact base. Sometimes called a working memory.
Rule	A description of some action to be taken or conclusion to be drawn when certain conditions are to be met. Usually describes some conclusion which can be made if specified conditions are met.
Entity	A component of the fact base. An entity is a formal representation of some thing or idea in the real world, about which rules can be written.
Entity Type	A description of how a certain class of entity should be constructed and behave. To a Smalltalk programmer, this is much like a class definition for an object.
Parameter	This is a value which describes some aspect of an entity. Parameters are the means by which entities of the same type can be distinguished from one another.

Premise	The part of a rule statement which HUMBLE must test for truth in order to decide whether the action is to be taken.
Action	The part of a rule statement which makes conclusions, fires other rules, or evaluates other statements. The action occurs only when the premise is true.
Uncertainty	The system of values attached to every data element indicating HUMBLE's level of belief in that datum.
Conclusion	A possible action in a rule statement. A conclusion is an assertion of new information based on existing information in the fact base.
YN Parameter	A YN parameter is a parameter which supports only yes or no (true/false) values. It has slightly more specialized behavior compared to a normal parameter.
MV Parameter	An MV parameter is a parameter which supports any number of simultaneous values. Unlike a standard parameter, it does not assume that there is only one correct value for itself.
BNF	Backus – Naur Form. A widely used notation scheme used to precisely describe the syntax of a computer language.
Forward Chaining	A method of directing the order of rule execution. Forward chaining leaves control of the order strictly in the hands of the user.

Backward Chaining	A method of directing the order of rule execution. Backward chaining places control of the order of execution with the inference engine. The structure of the rules themselves dictates the order in which they will be fired.
Change Block	The Change Block is a block of Smalltalk – 80 code associated with a parameter. It is automatically executed by HUMBLE whenever the best hypothesis of that parameter has changed.
Entity Tree	An Entity Tree is a collection of related entities. Each entity is either at the top of the tree or is a part of another entity. The entity tree has implications for how rules work and what data they can access.
Escape	An Escape is a small section of ordinary Smalltalk – 80 code imbedded in a rule. This allws the system implementor to add functionality not originally designed into HUMBLE.
Hypothesis	A potential value for a parameter. A hypothesis has a suggested value, a level of certainty, and some information used to produce explanations.
Main Parameters	Every entity type has a list of parameters considered so essential that the system should attempt to find out their value immediately upon creation. This list is alled the main parameter list.

Nested Statements	HUMBLE allows statements in a rule to evaluate more statements as part of their action. Such statements are called nested, since they resemble nested if – thens common to ordinary programming.
Recursive Rules	Some rules test values in the premise that the later make conclusions about in the action. These are called recursive rules. HUMBLE permits them, with certain restrictions on when they should be fired.
Statement	A part of a HUMBLE rule which tests some premise and then takes some action. Similar to a statement in an ordinary programming language.
Working Memory	The part of a knowledge base which stores temporary information. Information which is useful only during the present consultation resides in the fact base. Sometimes called a fact base.
Rule Base	The collection of rules, entity types, and parameter definitions in a knowledge base.

Appendix B. ROX Listing

ROX, a HUMBLE knowledge base

as of 10 June 1986 9:38:10 am

Entity Types

Rock
    typeAbove: nil
    createPrompt: Are there any Rocks to consider?
    addPrompt: Are there any other Rocks?
    assumePrompt: I am creating a Rock
    defaultName: ROCK
    parameters: #(class name texture fabric color grainSize aphanitic grainShape crystalline gritty fossiliferous soft porphyritic calcareous attributes)
    mainParameters: #(name)

Mineral
    typeAbove: Rock
    createPrompt: Are there any identifiable minerals in the rock?
    addPrompt: Are there any other identifiable minerals in the rock?
    assumePrompt: I am creating one mineral of the rock
    defaultName: MINERAL
    parameters: #(mineralName amount)
    mainParameters: #(mineralName amount)

Parameter Definitions

amount
describes: Mineral
type: Number
prompt: What percentage (1 – 100) of &mineralName is there?
promptFlag: askFirst
explanation: "The percentage of this mineral which exists in the rock"
remark: ""
deducingRules: #()
changeBlock:

Associated Rules:
rules which infer amount – *None!*
rules which test amount –
phaneriticKSpar
phaneriticPlag
phaneriticPyroxene
phaneriticQuartz

aphanitic
describes: Rock
type: YN
prompt:
promptFlag: nil
explanation: "a rock classification indicating an igneous rock with large grains"
remark: ""
deducingRules: #(phaneriticCheck aphaniticCheck)
changeBlock:

Associated Rules:
rules which infer aphanitic –
aphaniticCheck
phaneriticCheck
rules which test aphanitic –
aphaniticColor

phaneriticBiotite
phaneriticColor
phaneriticKSpar
phaneriticOlivine
phaneriticPlag
phaneriticPyroxene
phaneriticQuartz

attributes
describes: Rock
type: MV
prompt: –
promptFlag: nil
explanation: "the various attributes of the rock"
remark: ""
deducingRules: #()
changeBlock:

Associated Rules:
rules which infer attributes – *None!*
rules which test attributes –
calcareousCheck
fossiliferousCheck

calcareous
describes: Rock
type: YN
prompt: Does & bubble with dilute HCL?
promptFlag: askFirst
explanation: "a rock classification indicating the presence of calcium carbonate in the rock"
remark: ""
deducingRules: #()
changeBlock:

Associated Rules:
rules which infer calcareous – *None!*
rules which test calcareous –
calcareousCheck
chemicalSedCheck

class
describes: Rock

type: #('igneous' 'metamorphic' 'sedimentary')
prompt: What is the class of & ?
promptFlag: askLast
explanation: "the rock's general classification; igneous, sedimentary, or metamorphic"
remark: "The ordering of rules is important here, for the sake of a more natural flow"
deducingRules: #(sedimentaryCheck calcareousCheck fossiliferousCheck metamorphicCheck igneousCheck weldedCheck stretchedCheck)
changeBlock:

Associated Rules:

rules which infer class –

calcareousCheck
fossiliferousCheck
igneousCheck
metamorphicCheck
sedimentaryCheck
stretchedCheck
weldedCheck

rules which test class –

aphaniticCheck
brecciaCheck
chemicalSedCheck
conglomerateCheck
gneissCheck
metaSedCheck
phaneriticCheck
porphyriticCheck
sandstoneCheck
schistCheck
shaleCheck
siltstoneCheck
stretchedCheck
weldedCheck

color
describes: Rock
type: Number
prompt: What is the percent black (1 – 100) of & ?
promptFlag: askFirst
explanation: "what percentage of the rock is black material"
remark: ""

deducingRules: #()
changeBlock:

Associated Rules:
rules which infer color – *None!*
rules which test color –
aphaniticColor
phaneriticColor

crystalline
describes: Rock
type: YN
prompt: Do you think & is crystalline?
promptFlag: askFirst
explanation: "whether the rock is composed of noticeable crystals"
remark: ""
deducingRules: #()
changeBlock:

Associated Rules:
rules which infer crystalline – *None!*
rules which test crystalline –
igneousCheck
metamorphicCheck
sedimentaryCheck

fabric
describes: Rock
type: #('fine' 'medium' 'coarse' 'none')
prompt: What sort or layering is in & ?
promptFlag: askFirst
explanation: "a classification indicating coarseness of layering"
remark: ""
deducingRules: #()
changeBlock:

Associated Rules:
rules which infer fabric – *None!*
rules which test fabric –
gneissCheck
igneousCheck
metamorphicCheck
schistCheck

fossiliferous
describes: Rock
type: YN
prompt: Does & contain any fossils?
promptFlag: askFirst
explanation: "whether the rock contains any fossils"
remark: ""
deducingRules: #()
changeBlock:

Associated Rules:
rules which infer fossiliferous – *None!*
rules which test fossiliferous –
fossiliferousCheck

grainShape
describes: Rock
type: #('rounded' 'angular' 'stretched' 'fused')
prompt: Which best describes the grain shape for & ?
promptFlag: askFirst
explanation: "the shape of a the grains the rock is composed of"
remark: ""
deducingRules: #()
changeBlock:

Associated Rules:
rules which infer grainShape – *None!*
rules which test grainShape –
brecciaCheck
conglomerateCheck
metaSedCheck
sedimentaryCheck
stretchedCheck
weldedCheck

grainSize
describes: Rock
type: Number
prompt: How big are the grains/crystals (in mm.) of & ?
promptFlag: askFirst
explanation: "the size in millimeters of individual grains in the rock (see also porphyritic)"

remark: ""
deducingRules: #()
changeBlock:

Associated Rules:
rules which infer grainSize – *None!*
rules which test grainSize –
aphaniticCheck
brecciaCheck
conglomerateCheck
metaSedCheck
phaneriticCheck
sandstoneCheck
shaleCheck
siltstoneCheck

gritty
describes: Rock
type: YN
prompt: Is & gritty on the tongue?
promptFlag: askFirst
explanation: "A rather odd classification used to distinguish sandstone from siltstone. The user tests this by placing the rock against his tongue and reporting the resulting feel"
remark: ""
deducingRules: #()
changeBlock:

Associated Rules:
rules which infer gritty – *None!*
rules which test gritty –
shaleCheck
siltstoneCheck

mineralName
describes: Mineral
type: #('quartz' 'plagioclase' 'potassium feldspar' 'biotite' 'hornblende' 'pyroxene' 'olivine')
prompt: What is the name of & ?
promptFlag: askFirst
explanation: "What sort of mineral this is"
remark: ""
deducingRules: #()

changeBlock:

Associated Rules:
rules which infer mineralName – *None!*
rules which test mineralName –
phaneriticBiotite
phaneriticKSpar
phaneriticOlivine
phaneriticPlag
phaneriticPyroxene
phaneriticQuartz

name
describes: Rock
type: String
prompt: What is the name of & ?
promptFlag: nil
explanation: "the scientific name for the type of rock this entity is "
remark: ""
deducingRules: #(brecciaCheck schistCheck phaneriticKSpar phaneriticQuartz chemicalSedCheck phaneriticOlivine conglomerateCheck shaleCheck metaSedCheck sandstoneCheck phaneriticPyroxene phaneriticBiotite phaneriticColor siltstoneCheck aphaniticColor phaneriticPlag weldedCheck gneissCheck)
changeBlock:

Associated Rules:
rules which infer name –
aphaniticColor
brecciaCheck
chemicalSedCheck
conglomerateCheck
gneissCheck
metaSedCheck
phaneriticBiotite
phaneriticColor
phaneriticKSpar
phaneriticOlivine
phaneriticPlag
phaneriticPyroxene
phaneriticQuartz
sandstoneCheck
schistCheck

shaleCheck
siltstoneCheck
weldedCheck
rules which test name –
porphyriticCheck

porphyritic
describes: Rock
type: YN
prompt: Does & contain any significantly larger crystals?
promptFlag: askFirst
explanation: "whether the rock has any particularly large crystals"
remark: ""
deducingRules: #()
changeBlock:

Associated Rules:
rules which infer porphyritic – *None!*
rules which test porphyritic –
porphyriticCheck

soft
describes: Rock
type: YN
prompt: Can you easily scratch & with a penny?
promptFlag: askFirst
explanation: "can the rock be scratched by a penny or soft metal easily"
remark: ""
deducingRules: #()
changeBlock:

Associated Rules:
rules which infer soft – *None!*
rules which test soft –
igneousCheck
metamorphicCheck

texture
describes: Rock
type: #('crystaline' 'clastic' 'chemical' 'glassy')
prompt: What is the texture of & ?
promptFlag: nil

explanation: "an internal parameter; crystaline, clastic, chemical, glassy"
remark: ""
deducingRules: #(sedimentaryCheck metamorphicCheck igneousCheck)
changeBlock:

Associated Rules:

rules which infer texture –

igneousCheck
metamorphicCheck
sedimentaryCheck

rules which test texture –

chemicalSedCheck

Rules

aphaniticCheck

"if the rock is igneous and very fine grained
then it is aphanitic"

```
if: (class = 'igneous' & (grainSize <= 0.5))
then: [aphanitic is: true]
"Executes in the context of Rock entities"
```

---

aphaniticColor

"if the rock is aphanitic
then
if color > 50 then the rock is a basalt,
if color < 30 then the rock is a dacite
if 50 > color >=30 then the rock is andesite"

```
if: aphanitic
then: [if: color >= 50
    then: [name is: 'basalt' withCertainty: 0.8].
    if: color < 30
    then: [name is: 'dacite' withCertainty: 0.8].
    if: color >= 30 & (color < 50)
    then: [name is: 'andesite' withCertainty: 0.8]]
"Executes in the context of Rock entities"
```

---

brecciaCheck

"if the rock is sedimentary and has large angular clasts
then the rock is a breccia"

```
if: (class = 'sedimentary' & (grainSize >= 2) & (grainShape =
'angular'))
then: [name is: 'breccia' withCertainty: 0.8]
"Executes in the context of Rock entities"
```

---

calcareousCheck

"if the rock contains calcite then it is a calcareous rock,
and almost always sedimentary"

```
if: calcareous
then:[class is: 'sedimentary' withCertainty: 0.8.
```

```
            attributes has: 'calcium carbonate component']
"Executes in the context of Rock entities"
```

---

chemicalSedCheck

```
    "if the rock is sedimentary and chemically deposited (texture =
'chemical')
        then
            if it is calcareous
                then rock is a limestone,
                otherwise it is evaporite"
    if: class = 'sedimentary'
    then: [
        if: (texture = 'chemical' & calcareous)
        then: [name is: 'limestone' withCertainty: 0.9].
        if: (texture = 'chemical' & calcareous not)
        then: [name is: 'evaporite' withCertainty: 0.9]]
"Executes in the context of Rock entities"
```

---

conglomerateCheck

```
    "if the rock is sedimentary and has large rounded clasts
        then it is a conglomerate"
    if: (class = 'sedimentary' & (grainSize >= 2) & (grainShape =
'rounded'))
    then: [name is: 'conglomerate' withCertainty: 0.8]
"Executes in the context of Rock entities"
```

---

fossiliferousCheck

```
    "any rock which contains some fossils is fossiliferous,
        and almost always either sedimentary or metamorphic"
    if: fossiliferous
        then: [class is: 'sedimentary' withCertainty: 0.4.
            class is: 'metamorphic' withCertainty: 0.4.
            attributes has: 'contains fossils']
"Executes in the context of Rock entities"
```

---

gneissCheck

```
    "the rock is metamorphic,
        and has either medium or coarse layering
        then it is a gneiss"
```

```
    if: (class = 'metamorphic' & (fabric = 'medium' | (fabric = 'coarse')))
    then: [name is: 'gneiss' withCertainty: 0.8]
"Executes in the context of Rock entities"
```

---

igneousCheck

```
    "if the rock is crystaline with no layering (fabric is 'none')
        then
            if the rock is soft the class of rock is sedimentary
                with a chemical texture
            otherwise it is an igneous rock of crystalline texture"
    if: crystalline & (fabric = 'none')
    then: [
        if: soft
        then: [class is: 'sedimentary' withCertainty: 0.8.
            texture is: 'chemical']
        else: [class is: 'igneous' withCertainty: 0.8.
            texture is: 'crystalline']]
"Executes in the context of Rock entities"
```

---

metamorphicCheck

```
    "if the rock is crystaline, with some sort of layering or foliation
        then a soft rock is sedimentary with a chemical texture,
            but a hard rock is metamorphic"
    if: (crystalline & (fabric ~= 'none') & (soft not ))
    then: [class is: 'metamorphic' withCertainty: 0.8].
    if: (crystalline & (fabric ~= 'none') & (soft))
    then: [class is: 'sedimentary' withCertainty: 0.8.
            texture is: 'chemical' ]
"Executes in the context of Rock entities"
```

---

metaSedCheck

```
    "if the rock is a metamorphic rock with stretched grain
        then
            if the grains are larger than 4mm
                then the rock is probably a meta-conglomerate
            if the grains are smaller than 0.5mm
                then the rock is probably a slate
            if the grains are larger than 0.5mm but smaller than 4mm
                then the rock is probably a meta-sandstone"
    if: (class = 'metamorphic' & (grainShape = 'stretched'))
    then: [
```

```
        if: (grainSize >= 4.0)
        then: [name is: 'meta-conglomerate' withCertainty: 0.8].
        if: (grainSize < 0.5)
        then: [name is: 'slate' withCertainty: 0.8].
        if: (grainSize >= 0.5 & (grainSize < 4.0))
        then: [name is: 'meta-sandstone' withCertainty: 0.8]]
"Executes in the context of Rock entities"
```

---

phaneriticBiotite

```
    "the rock is phaneritic and biotite is present
        then the rock may be either a granite or diorite
            but is probably not a gabbro"
    if: aphanitic not
    then: [
        if: (anyOf: Mineral have: [mineralName = 'biotite'])
        then: [name is: 'granite' withCertainty: 0.3.
            name is: 'diorite' withCertainty: 0.2.
            name is: 'gabbro' withCertainty: -0.5]]
"Executes in the context of Rock entities"
```

---

phaneriticCheck

```
    "if the rock is igneous and rather large grained
        then it is phaneritic (not aphanitic)"
    if: (class = 'igneous' & (grainSize > 0.5))
    then: [aphanitic is: false]
"Executes in the context of Rock entities"
```

---

phaneriticColor

```
    "if the rock is phaneritic
        then
            if the color >=50 then the rock is possibly a gabbro
            if the color <20 then the rock is possibly a granite
            if the 20 <= color <50 then the rock may be a diorite"
    if: aphanitic not
    then: [
        if: color >= 50
        then: [name is: 'gabbro' withCertainty: 0.4].
        if: color < 20
        then: [name is: 'granite' withCertainty: 0.4].
        if: color >= 20 & (color < 50)
        then: [name is: 'diorite' withCertainty: 0.4]]
```

```
"Executes in the context of Rock entities"


- - - - - - -
phaneriticKSpar
    "if the rock is phaneritic
        then if potassium feldspar is present in quantities > 15%
            the rock is probably a granite or less probably a diorite
        and any amount of pottasium feldspar means
            that it is probably not a gabbro"
    if: aphanitic not
    then: [
        if: (anyOf: Mineral
            have: [mineralName = 'potassium feldspar' & ( amount >
15)])
        then: [name is: 'granite' withCertainty: 0.7].


        if: (anyOf: Mineral
            have: [mineralName = 'potassium feldspar' & ( amount > =
15)])
        then: [name is: 'diorite' withCertainty: 0.4].


        if: (anyOf: Mineral
            have: [mineralName = 'potassium feldspar'])
        then: [name is: 'gabbro' withCertainty: -0.7]]
"Executes in the context of Rock entities"


- - - - - - -
phaneriticOlivine
    "the rock is phaneritic and olivine is present
        then the rock is probably a gabbro,
            and is probably not a diorite or a granite"
    if: aphanitic not
    then: [
        if: (anyOf: Mineral have: [mineralName = 'olivine'])
        then: [name is: 'gabbro' withCertainty: 0.8.
            name is: 'diorite' withCertainty: -0.5.
            name is: 'granite' withCertainty: -0.5]]
"Executes in the context of Rock entities"


- - - - - - -
phaneriticPlag
    "the rock is phaneritic and the amount of plagoiclase is
        < 30%
```

then the rock may be a granite
between 30 and 70%
then the rock may be a diorite
< 60%
then the rock may be a gabbro"

```
    if: aphanitic not
    then: [
        if: (anyOf: Mineral have: [mineralName = 'plagioclase' &
(amount < 30)])
        then: [name is: 'granite' withCertainty: 0.7].

        if: (anyOf: Mineral have: [mineralName = 'plagioclase' &
(amount >= 30) & (amount < 70) ])
        then: [name is: 'diorite' withCertainty: 0.7].

        if: (anyOf: Mineral have: [mineralName = 'plagioclase' &
(amount < 60)])
        then: [name is: 'gabbro' withCertainty: 0.7].
    ]
"Executes in the context of Rock entities"
```

---

phaneriticPyroxene

```
    "if the rock is phaneritic
        and the percentage of pyroxene > 15%
            then the rock is probably a gabbro
        if the percentage is <=10%
            then the rock is probably a diorite
        but any amount of pyroxene means that it is probably not a
granite"
    if: aphanitic not
    then: [
        if: (anyOf: Mineral have: [mineralName = 'pyroxene' & (amount
> 15)])
        then: [name is: 'gabbro' withCertainty: 0.8].
        if: (anyOf: Mineral have: [mineralName = 'pyroxene' & (amount
<=10)])
        then: [name is: 'diorite' withCertainty: 0.5].
        if: (anyOf: Mineral have: [mineralName = 'pyroxene'])
        then: [name is: 'granite' withCertainty: -0.5].]
"Executes in the context of Rock entities"
```

---

phaneriticQuartz

```
    "if the rock is phaneritic
        then if the percentage of quartz is > 10%
            then the rock is possibly a granite or diorite
        but if any quartz at all is present then it is probably not a gabbro"
    if: aphanitic not
    then: [
        if: (anyOf: Mineral have: [mineralName = 'quartz' & (amount
>= 10)])
        then: [name is: 'granite' withCertainty: 0.5.
            name is: 'diorite' withCertainty: 0.5].
        if: (anyOf: Mineral have: [mineralName = 'quartz'])
        then: [name is: 'gabbro' withCertainty: -0.5]]
"Executes in the context of Rock entities"


- - - - - - - -
porphyriticCheck
    "if the rock is igneous and has some large crystals,
        then it is a porphyritic rock"
    if: (class = 'igneous')
    then: [
        if: name isKnown & porphyritic
        then: [attrinutes includes: 'porphyritic']]
"Executes in the context of Rock entities"


- - - - - - - -
sandstoneCheck
    "if the rock is sedimentary, and only somewhat small grained,
        then it is a sandstone"
    if: (class = 'sedimentary' & (grainSize < 2) & (grainSize > 0.1))
    then: [name is: 'sandstone' withCertainty: 0.8]
"Executes in the context of Rock entities"


- - - - - - - -
schistCheck
    "if the rock is metamorphic and finely layered
        then it is a schist"
    if: (class = 'metamorphic' & (fabric = 'fine'))
    then: [name is: 'schist' withCertainty: 0.8]
"Executes in the context of Rock entities"


- - - - - - - -
sedimentaryCheck
    "if the rock is either
```

1. not crystalline or
2. crstyalline with rounded grains
then the rock is sedimentary and clastic"
if: (crystalline not)
then: [class is: 'sedimentary' withCertainty: 0.8.
texture is: 'clastic'].
if: (crystalline & (grainShape = 'rounded'))
then: [class is: 'sedimentary' withCertainty: 0.9.
texture is: 'clastic']
"Executes in the context of Rock entities"

---

shaleCheck
"if the rock is sedimentary, small grained, and not gritty
then it is a shale"
if: (class = 'sedimentary' & (grainSize < = 0.1))
then: [
if: gritty not
then: [name is: 'shale' withCertainty: 0.8]]
"Executes in the context of Rock entities"

---

siltstoneCheck
"any rock that is sedimentary, small grained, and gritty to the tongue is a siltstone"
if: (class = 'sedimentary' & (grainSize < = 0.1))
then: [
if: gritty
then: [name is: 'siltstone' withCertainty: 0.8]]
"Executes in the context of Rock entities"

---

stretchedCheck
"the rock appears sedimentary but has stretched grains
then it is not sedimentary but is instead metamorphic"
if: (class = 'sedimentary' & (grainShape ='stretched'))
then: [class is: 'sedimentary' withCertainty: −0.9.
class is: 'metamorphic' withCertainty: 0.9.]
"Executes in the context of Rock entities"

---

weldedCheck
"if the rock appears sedimentary but has fused grains

```
        then it is not sedimentary but is instead igneous
            and is probably a welded tuff'"
    if: (class = 'sedimentary' & (grainShape = 'fused'))
    then: [class is: 'sedimentary' withCertainty: -0.9.
        class is: 'igneous' withCertainty: 0.95.
        name is: 'welded tuff' withCertainty: 0.8]
"Executes in the context of Rock entities"
```

Rule Metrics

Total Number of Entity Types: 2
Total Number of Rules: 27
An Average Rule Tests 2.48148 Parameters
An Average Rules Makes Conclusions About 1.11111 Parameters
Total Number of Parameter Definitions: 17
2 parameters with 1 associated rules, 12%
5 parameters with 2 associated rules, 29%
1 parameters with 3 associated rules, 6%
3 parameters with 4 associated rules, 18%
2 parameters with 6 associated rules, 12%
1 parameters with 8 associated rules, 6%
1 parameters with 10 associated rules, 6%
1 parameters with 19 associated rules, 6%
1 parameters with 21 associated rules, 6%

Index

addPrompt: 3.1.1, 3.1.1.2, 9.2, 10.4.2.1.3
allBases: 10.3
allSubEntities: 10.4.2.1.1
allSubEntitiesOfType: 10.4.2.1.1
allSubEntitiesWithParameter: 10.4.2.1.1
alternative: 4.4.1, 4.4.2, 8.5, 9.3, 9.4, 10.4.5
annotationStream: 10.4.8
assumePrompt: 3.1.1, 3.1.1.2, 9.2, 10.4.2.1.3
backward chaining: 4.2.3.1.2.2, 4.2.3.1.2.3, 4.3, 8.2, 10.4.1, 10.4.1.1, 10.4.1.2
bestHypothesis: 10.4.2.2
BNF: 4.4
certainty: 1.1, 3.2.1, 3.2.2.2.3, 4.2.3.1.2.1, 4.4.2, 4.4.3, 4.4.4, 4.5.1, 4.5.2, 4.5.3, 6.1.4.3, 8.3, 9.3, 10.4.2.2, 10.4.2.2.1, 10.4.2.3, 10.4.9, 10.5, 10.5.1, 10.5.2
changeBlock: 3.2.2, 3.2.2.7, 9.2, 9.5, 10.4.2.2.2
consultation: 3.1.1.2, 3.2.1, 5.4.1, 5.4.2, 8.1, 8.2, 8.3, 9.4, 10.4.6, 10.4.9
context: 3, 4.2.3.2, 4.4.2, 4.5.2, 10.4.2.1, 10.4.2.1.4
createPrompt: 3.1.1, 3.1.1.2, 9.2, 10.4.2.1.3
deducing rules: 3.2.2.6
deductionRules: 10.4.2.2.2
defaultName: 3.1.1, 3.1.1.2, 9.2
definitelyIs: 4.4.2, 4.4.3
definitelyIsNot: 4.4.2, 4.4.3
describes: 2, 3.2.2, 4.2.3.1, 9.2, 9.5
edit: 1.2, 3.1.1.3, 3.2.2.6, 4.1, 4.2.2, 5.3, 6.1.1, 6.1.2, 6.1.3, 6.1.4.1, 6.1.4.2, 8.4, 9.2, 10.3
Editor: 3.1.1, 3.2.2, 5.3, 6.1, 6.1.1, 9.2
entity: 2, 3, 3.1, 3.1.1, 3.1.1.1, 3.1.1.2, 3.1.1.3, 3.1.1.4, 3.2, 3.2.2.2.1, 3.2.2.2.4, 3.2.2.3.1, 3.3, 4.2, 4.2.3.2, 4.3, 4.4.1, 4.4.2, 4.5.2, 5.6, 6.1.1, 6.1.2, 6.1.3, 8.3, 8.5, 8.6, 9.2, 9.5, 10.4.1.1, 10.4.2.1, 10.4.2.1.1, 10.4.2.1.3, 10.4.2.1.4, 10.4.2.2, 10.4.2.2.2, 10.4.3, 10.4.4, 10.4.9
entity tree: 3.1, 3.1.1.1, 3.3, 4.2, 4.2.3.2, 9.2, 10.4.2.1.1, 10.4.2.1.3, 10.4.2.1.4
entity type: 2, 3, 3.1, 3.1.1, 3.1.1.1, 3.1.1.3, 3.2, 3.3, 4.2.3.2, 4.4.2, 5.6, 6.1.1, 6.1.2, 6.1.3, 9.2, 10.4.2.2.2
entityType: 4.4.2, 10.4.2.2.2, 10.4.9
enumerated: 3.2.2.2.5

escape: 4.4.4
example: 1.3, 3.1, 3.1.1, 3.1.1.1, 3.2.2.2.1, 3.2.2.2.2, 3.2.2.2.3, 3.2.2.2.4, 3.2.2.2.5, 3.2.2.3.1, 3.2.2.6, 3.3, 4.2.3.1.1, 4.2.3.1.2.2, 4.2.3.1.2.3, 4.4, 4.4.1, 4.5, 4.5.2, 4.5.3, 5.2, 8.5, 9.1, 9.2, 10.3, 10.4.1.2, 10.4.2.2.1, 10.4.3, 10.4.4, 10.4.6, 10.4.9
execute: 3.1, 3.2.2.7, 3.3, 4.2.3.1, 4.2.3.1.2.3, 4.3, 4.5.2, 4.5.3, 5.4, 8.2, 8.6, 9.5, 10.4.1, 10.4.1.2, 10.4.2.1.4, 10.4.2.2.1, 10.4.2.2.2
explain: 1.1, 3.2.2, 3.2.2.2.3, 3.2.2.4, 3.3, 4.2, 4.2.2, 4.2.3.1.2, 4.4, 4.4.2, 8.3, 8.4, 8.5, 10.3, 10.4.2, 10.4.2.2.2, 10.4.2.3, 10.4.4
fact base: 1.1, 2, 3.2.1, 4.4.4, 8.2, 10.4.1.1, 10.4.2, 10.4.2.1, 10.4.2.1.4, 10.4.3, 10.4.9
fileInBase: 10.3
findOut: 10.4.1.1, 10.4.9
forward chaining: 4.2.2, 8.2, 10.4.1.2
goal: 4.2.3.1.2, 4.3, 5.4, 5.4.1, 5.4.2, 10.4.8
hypotheses: 4.4.3, 8.5, 10.4.2.2.1, 10.4.2.3, 10.4.5
hypothesisType: 10.5
if: 1.1, 1.2, 3, 3.1, 3.1.1, 3.1.1.1, 3.1.1.2, 3.1.1.4, 3.2, 3.2.1, 3.2.2.2, 3.2.2.2.1, 3.2.2.2.3, 3.2.2.2.4, 3.2.2.2.5, 3.2.2.3.2, 3.2.2.4, 3.2.2.6, 3.2.2.7, 3.3, 4.2.2, 4.2.3.1, 4.2.3.1.1, 4.2.3.1.2.1, 4.2.3.1.2.2, 4.2.3.1.2.3, 4.2.3.1.2.4, 4.2.3.2, 4.3, 4.4, 4.4.2, 4.4.3, 4.4.4, 4.5.1, 4.5.2, 4.5.3, 5.4.1, 6.1, 6.1.4.2, 6.1.4.3, 8.2, 8.3, 9.3, 9.5, 10.1, 10.2, 10.4.1, 10.4.1.2, 10.4.2, 10.4.2.1.1, 10.4.2.2.2, 10.4.5, 10.4.8, 10.4.9, 10.5, 10.5.1
ifAll: 4.2.3.2, 4.4.2, 6.1.4.2
ifAny: 4.2.3.2, 4.4.2, 4.5.2, 6.1.4.2, 9.5
ifNone: 4.2.3.2, 4.4.2, 6.1.4.2
initEntities: 10.4.1.1, 10.4.3, 10.4.9
initEntityTypes: 10.4.1.1, 10.4.3, 10.4.9
interactWith: 10.3
interrogator: 10.4.7
isEqual: 4.4.2, 4.4.3
isNotDefinitely: 4.4.2, 4.4.3
isNotDefinitelyNot: 4.4.2, 4.4.3
isNotEqualTo: 4.4.2, 4.4.3
isNotKnownToBe: 4.4.2, 4.4.3
knowledge base: 1.1, 1.3, 2, 3, 3.1.1.3, 3.1.1.4, 3.2.1, 3.2.2.2.1, 3.2.2.2.2, 3.2.2.2.3, 3.2.2.2.4, 3.2.2.2.5, 3.2.2.3, 3.2.2.5, 4.1, 4.2.2, 4.2.3.1.1, 4.2.3.1.2.1, 4.4.1, 4.4.3, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 6.1, 6.1.1, 6.1.2, 6.1.3, 6.1.4.1, 6.1.4.3, 8.1, 8.2, 8.5, 9, 9.1, 9.2, 9.3, 9.4, 9.5, 10.3, 10.4, 10.4.1.1, 10.4.1.2, 10.4.2, 10.4.2.2.1, 10.4.2.2.2, 10.4.3, 10.4.4, 10.4.6, 10.4.7, 10.4.8, 10.4.9, 10.5
knowledgeBase: 10.4.2.2.2
KnowledgeBases: 10.3, 10.4.1.2, 10.4.2.2.1, 10.4.3, 10.4.6, 10.4.9

Listener: 3.2.2.4, 5.4, 5.4.1, 8.1, 8.6, 9.4, 10.4.4, 10.4.5, 10.4.6
listing: 1.3, 5.6
lunch: 1.1, 9.1, 9.2, 9.5
mainParameters: 3.1.1, 3.1.1.4, 9.2
Manager: 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 9.2, 9.4, 10.3
mightBe: 3.2.2.6, 4.4.2, 4.4.3
mightNotBe: 4.4.2, 4.4.3
MV: 3.2.2, 3.2.2.2, 3.2.2.2.2, 10.4.2.2.2
nested: 4.2.3.1.2.2, 4.2.3.1.2.3, 4.5.3
number: 2, 3.1.1, 3.1.1.2, 3.2.1, 3.2.2.2.3, 3.2.2.2.5, 3.2.2.7, 4.2.3.1.2.2, 4.4, 4.4.1, 4.4.2, 4.5, 9.1, 9.2, 10.3, 10.4, 10.4.1.2, 10.4.2, 10.4.2.1.1, 10.4.2.2.2, 10.4.6, 10.5
outputStream: 10.4.6
parameter: 3.1, 3.1.1, 3.1.1.3, 3.1.1.4, 3.2, 3.2.1, 3.2.2, 3.2.2.1, 3.2.2.2, 3.2.2.2.1, 3.2.2.2.2, 3.2.2.2.3, 3.2.2.2.4, 3.2.2.2.5, 3.2.2.3, 3.2.2.3.1, 3.2.2.3.2, 3.2.2.4, 3.2.2.5, 3.2.2.6, 3.2.2.7, 3.3, 4.2.3.1.1, 4.2.3.1.2.1, 4.2.3.1.2.2, 4.2.3.1.2.3, 4.2.3.1.2.4, 4.3, 4.4.2, 4.4.3, 4.4.4, 4.5.1, 4.5.2, 5.4.1, 5.6, 6.1.1, 6.1.2, 6.1.3, 6.1.4.3, 8.2, 8.3, 8.4, 8.5, 9.2, 9.3, 9.5, 10.4, 10.4.1.1, 10.4.1.2, 10.4.2.1, 10.4.2.1.1, 10.4.2.1.2, 10.4.2.2, 10.4.2.2.1, 10.4.2.2.2, 10.4.4, 10.4.5, 10.4.9, 10.5.1, 10.5.2
parameter definition: 3.1.1.3, 3.2.2, 3.2.2.2, 3.2.2.3, 3.2.2.3.1, 3.2.2.6, 3.2.2.7, 4.3, 4.4.2, 5.6, 6.1.2, 6.1.3, 8.4, 9.2, 10.4.1.1, 10.4.2.2.2
parameterName: 10.4.2.1.2, 10.4.2.2, 10.4.2.2.1, 10.4.2.2.2
parameterNamed: 10.4.2.1.2, 10.4.2.2, 10.4.2.2.1
primaryEntity: 10.4.2.1, 10.4.2.2.1
prompt: 3.1.1.2, 3.2.2, 3.2.2.3, 3.2.2.3.1, 5.2, 5.4.2, 9.2, 9.5, 10.3, 10.4.2.1.3, 10.4.2.2.2
promptFlag: 3.2.2, 3.2.2.3, 9.2, 9.5, 10.4.2.2.2
reason: 1.1, 3, 3.1, 3.2.1, 3.2.2.2.3, 3.2.2.3.2, 3.3, 4.2.1, 4.2.3.1.2.4, 4.2.3.2, 4.4.3, 10.4.2.1, 10.4.2.2, 10.4.2.3, 10.4.4
referringRules: 10.4.2.2.2
remark: 3.2.2, 9.2, 9.5, 10.4.2.2.2
remove: 3.2.2.7, 5.1, 5.2, 6.1.1, 6.1.2, 6.1.3, 10.3
ROX: 1.3, 3.1.1, 3.1.1.2, 3.2.2.2.1, 3.2.2.2.2, 3.2.2.2.3, 3.2.2.2.4, 3.2.2.2.5, 4.2.3.1.1, 4.4.1, 5.2, 5.6, 8.3, 8.5, 10.3, 10.4.2.2.1, 10.4.6, 10.4.9
rule: 1.1, 2, 3, 3.1, 3.1.1.2, 3.2, 3.2.1, 3.2.2.2.4, 3.2.2.6, 3.3, 4.1, 4.2, 4.2.1, 4.2.2, 4.2.3.1.2, 4.2.3.1.2.2, 4.2.3.1.2.3, 4.2.3.1.2.4, 4.2.3.2, 4.3, 4.4, 4.4.2, 4.4.4, 4.5, 4.5.1, 4.5.2, 4.5.3, 5.4, 5.4.1, 5.4.2, 5.6, 6.1.2, 6.1.3, 6.1.4.1, 6.1.4.2, 6.1.4.3, 8.2, 9.1, 9.2, 9.3, 9.5, 10.4.1, 10.4.1.2, 10.4.2.1.4, 10.4.2.2.1, 10.4.2.2.2, 10.4.2.3, 10.4.8, 10.4.9, 10.5.2
rule base: 2, 4.1
setParamsFromDict: 10.4.9

Smalltalk: 1.2, 3.2.2.2.3, 3.2.2.7, 4.1, 4.2.1, 4.2.2, 4.4.4, 5.2, 6.1.4.1, 6.1.4.3, 8.6, 10.1, 10.2, 10.4.6, 10.4.7
statement: 4.1, 4.2.1, 4.2.2, 4.2.3.1, 4.2.3.1.1, 4.2.3.1.2, 4.2.3.1.2.1, 4.2.3.1.2.2, 4.2.3.2, 4.4.2, 4.5.3, 6.1.4.2, 6.1.4.3, 8.3, 8.5
string: 3.2.2.2.4, 3.2.2.2.5, 3.2.2.3.1, 4.2.2, 4.2.3.1.2.1, 4.4.1, 4.4.2, 6.1.4.2, 8.5, 9.2, 10.3, 10.4.1.2, 10.4.2.2.2, 10.4.4, 10.4.5
superEntity: 10.4.2.1.4
trace: 5.4, 5.4.2, 10.4.8
type: 2, 3, 3.1, 3.1.1, 3.1.1.1, 3.1.1.2, 3.1.1.3, 3.2, 3.2.2, 3.2.2.1, 3.2.2.2, 3.2.2.2.5, 3.3, 4.2.3.1.1, 4.2.3.1.2.1, 4.2.3.2, 4.4, 4.4.2, 4.5.2, 5.6, 6.1.1, 6.1.2, 6.1.3, 6.1.4.2, 6.1.4.3, 8.2, 8.3, 8.5, 9.2, 9.5, 10.2, 10.4.2.1.1, 10.4.2.1.3, 10.4.2.2.2, 10.4.7, 10.4.8
typeAbove: 3.1.1, 3.1.1.1, 9.2, 10.4.2.1.3
typesBelow: 10.4.2.1.3
watch: 5.4, 5.4.1, 5.4.2
working memory: 2, 8.2
YN: 3.2.2, 3.2.2.2, 3.2.2.2.1, 4.2.3.1.1, 4.2.3.1.2.2, 10.4.2.2.2

XEROX

Xerox Special Information Systems
Vista Laboratory
250 North Halstead Street
P.O. Box 5608
Pasadena, CA 91107-0608