KRF

Knowledge Representation Framework Project
Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Knowledge Representation Framework: Overview of Languages and Mechanisms

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).
The present report, PM-krf-009, can persistently be accessed as follows:

Project Memo URL:	`www.ida.liu.se/ext/caisor/pm-archive/krf/009/`
AIP (Article Index Page):	`http://aip.name/se/Sandewall.Erik.-/2010/006/`
Date of manuscript:	2010-06-09

Related information can also be obtained through the following www sites:

KRFwebsite:	`http://www.ida.liu.se/ext/krf/`
AIP naming scheme:	`http://aip.name/info/`
The author:	`http://www.ida.liu.se/~erisa/`

Introduction

The Knowledge Representation Framework (KRF) contains a notational system that combines and extends the following traditional notations:

- Set theory notation for sets and sequences.
- The syntax of predicate logic.
- The notation for recursive functions of symbolic expressions, along the lines introduced by the Lisp programming language.

This framework also provides a simple notation for assigning attribute values to entities, and conventions for organizing definitions and other knowledge-base contents in a modular way as structured textfiles.

Among other notational approaches with similar goals, there is in particular the family of *S-expression-based languages* and the family of *SGML-based languages.* S-expressions are the recursively parenthesized expressions that were introduced by the Lisp language, and this family includes i.a. the Knowledge Interchange Format (KIF), the FIPA-ACL Agent Communication Language, Agent Planning Languages such as the Reactive Action Packages (RAPS), and finally the Planning Domain Definition Language (PDDL). SGML was the ancestor of HTML and XML, and SGML-based languages include the Resource Definition Framework (RDF), the Web Ontology Language (OWL) which is built on RDF, and a large number of application-specific languages. The relations between these languages and those in KRF will be discussed in several of the chapters in this report.

Although the KRF notation is based on the mathematical languages mentioned initially, its graphical appearance differs from them in order to make it more suitable for use in communication with computers. It is restricted to the Latin-1 character set, it does not use lowered or raised text (as for exponentials), and most importantly, it makes it convenient to use identifiers consisting of several characters, like e.g. in programming languages. This differs from the usual notation in mathematics and formal logic where formulas can be very compact, but at the expense of strong limitations on the size of the vocabulary being used.

It is intended that KRF notation shall be used both in printed text, for example textbooks and research articles, and for input to, and output from software systems. We want to make it as easy as possible to take a method that is described in a textbook and implement it in a computer; we also want to facilitate the step from computer output to published report, so that the experience from implemented software can be reported as transparently as possible.

The KRF notational system is *extensible* in the sense that it contains strict definitions of some notational domains and it also provides ways of extending the notation. We shall therefore describe it as a collection of *several interdependent languages.* These are interdependent in the sense that an expression in one language can occur as a sub-expression in another one of the languages, which means that when a new language is added, it can freely build on and incorporate the syntax of previously defined languages.

The KRF contains notation both for *modelling* (i.e. for describing the application or environment where a KRF-based system will be used) and for expressing *commands* to such a system. Both these aspects require clear definitions of the notation's semantics, i.e. what is the meaning of expressions in the respective notation. The KRF framework therefore introduces the notion of *mechanisms* in order to define the meaning of commands in a principled way. These mechanisms have only informal definitions at the present time.

Besides its basic notational system consisting of a few interrelated languages and the associated definitions of mechanisms, the Knowledge Representation Framework also contains the following parts:

- A *type system* and an *ontology* for use in organizing knowledgebases of nontrivial size.
- An *Agent Messaging Framework* (AMF) including an *Agent Message Language* (AML) for defining command-passing and other message-passing between software agents.
- A *Document Preparation Framework* (DPF) including a markup language for documents and for defining static and dynamic webpages.

A small part of the type system and ontology is used for defining the structure of the KRF mechanisms, but apart from that it is optional and need not be used. The Agent Messaging Framework is only needed in applications where agents send or receive messages, although it can also be used for defining web servers. Type system, ontology, AMF and DDF will not be addressed in the present report.

The `Leonardo` software system is an implementation of the KRF framework and contains concrete implementations of these facilities together with a knowledgebase for storing and operating on application models.

The present report introduces the core part of the KRF notational system. It is cited as required introductory reading for the following four groups of publications:

- For our textbook "Knowledge Representation for Intelligent Autonomous Agents," and in due time also for some research articles in the same area.
- For further reports on the Knowledge Representation Framework, in particular its type system, ontology, AMF and DDF.
- For the documentation of the Leonardo software system and its applications, and indeed also for the software of that system itself.
- For publications and systems in the area of *information analysis.*

The material in the present report has therefore been selected as a kind of least common denominator for what is required in those four topic areas, and the continued publications will build on it but proceed in different directions.

Additional publications and other information from work that is based on the Knowledge Representation Framework can be obtained from our website at `http://www.ida.liu.se/ext/krf/.`

Chapter 1

Overview

1.1 Language Structure

The Knowledge Representation Framework is based on four intertwined syntaxes or 'languages' which are as follows:

- The syntax for *Knowledge Representation Expressions,* or KR expressions, which is used for expressing structured information in textual form.
- The *Entityfiles syntax* which defines the structure of textfiles containing assignments of *attribute values* to individual symbols called *entities.* The values are written as KR expressions.
- The *Common Expression Language,* or CE language, which defines functions and predicates for use in recursively formed expressions. Expressions in the CE language are KR expressions; they can be evaluated and their values are also KR expressions.
- The *Session Command Language,* SCL, which is a language for commands that can be given to a software system that implements the Knowledge Representation Framework. Each SCL command consists of a *verb* and its sequence of arguments. The arguments are expressed in the CE language and the entire command is a KR expression.

A knowledgebase is therefore essentially a collection of entityfiles; operations on it and operations using it are expressed in the Session Command Language. The latter is the first one in a suite of command languages that are specialized for various purposes. They share a common syntax, and most of them use arguments written in the Common Expression Language.

Each command language allows the formation of *scripts* consisting of commands that are to be executed sequentially or concurrently. Some of the verbs in a command language are *elementary;* other verbs are defined using scripts, and ultimately in terms of elementary verbs. Entityfiles are used for storing the scripts that define non-elementary verbs.

The four languages shown above are interdependent so all of them are needed in order to have a working system. It is common however that

a system consists of several cooperating 'agents', and in this case a fifth language is needed:

- The *Agent Message Language,* AML, specifies the structure of messages that can be sent from one agent to another, including *requests* whereby an agent asks to obtain some information or some service, and *responses* that agents return after they have honored a request. The Agent Message Language is expressed as KR expressions, much like the CE language is.

The present chapter will give a few simple examples from each of the four languages listed initially, in order to give a general idea of how they are organized and how they combine. Later chapters define each language in more detail and in the following order:

- Chapter 2: Knowledge Representation Expressions
- Chapter 3: The Common Expression Language
- Chapter 4: The Session Command Language
- Chapter 5: Entityfiles and Processors
- Chapter 6: Some Knowledge Representation Issues
- Appendix: Notational Details in KRE

Agent cooperation and the Agent Message Language will not be addressed in the present report; please refer to a separate document in the same series.

An earlier language, the Knowledge Interchange Format (KIF) was introduced in the U.S. by DARPA in the early 1990's (reference) and with the same goals as the Common Expression Language. The CEL has many similarities with KIF; the major differences will be described in Chapter 3 (the chapter on CEL).

The Knowledge Query and Manipulation Language (KQML) (reference) was introduced at the same time as KIF, and as a complement to it. KQML was later replaced by FIPA-ACL, an industry-standard Agent Communication Language (reference). The Agent Message Language is essentially a subset of FIPA-ACL which is expressed using KR expressions for the sake of uniformity.

Other formats for the representation of structured information in text files, such as XML, OWL and YAML will be discussed in Chapter 5 (the chapter on the syntax of entityfiles).

1.2 Languages

The following very simple example is a fragment from an entityfile where values are assigned to attributes for two entities called `yellow` and `blue`. These entities are of course intended to represent the two colors whose English-language names they carry.

```
------------------------------------------------------------
-- yellow

[: type color]
[: has-examples {dandelion sun amber}]
[: translations {[: french jaune][: german gelb]}]

------------------------------------------------------------
-- blue

[: type color]
[: has-examples {bluebell forgetmenot sky turqoise}]
[: translations {[: french bleu][: german blau]}]

------------------------------------------------------------
```

Between two successive dash-lines one finds first a header line containing the entity that is to be defined, for example the entity `blue,` and then a sequence of *maplets* each of which consists of an *attribute* and the corresponding *value.* The values for the `has-examples` and `translations` attributes are simple examples of KR expressions.

Besides sets and mappings which occur in this example, the KRE notation also allows for a few other constructs, such as for sequences. It differs from conventional mathematical notation by being restricted to the character-set of a conventional computer keyboard, and its use of a fixed-width font, like from a typewriter. The latter convention is because large KR expressions must be written on several lines, which means systematic indentation must be used, and this looks more natural in typewriter style.

The representation of the color-related information that was shown above can be maintained and preserved in conventional files in a computer's directory structure. Such files are called entityfiles. In order to be used computationally they are read into the working memory of a *session* with a software system, such as Leonardo for example, and one may expect that such a system will convert this information into a datastructure where entities are represented as nodes in a network-like structure, and where the attributes values are represented in a computationally efficient form. These datastructures constitute the *dynamical manifestation* of the information, as opposed to the original *textual* manifestation. The two manifestations are interdependent since each can be converted to the other one.

Commands in the Session Command Language are intended to be addressed to sessions with software systems that implement the Knowledge Representation Framework, and their effect shall be to extract information from, or to modify the information in the current state of that session. The following is a simple example of an SCL command:

```
[addmap yellow translations [: swedish gul]]
```

This command would be intended to add the translation item `[: swedish gul]` to the `translations` attribute-value of the entity `yellow.` Actual implementations may receive such commands either by command-line input from the user, or as the result of an interaction on a webpage where the implemented system is the webserver, or by receiving a command-message from another software process. The textual representation of the command

serves as a normal form for these various ways of communicating it.

In this particular example, the three arguments in the command are KR expressions. However, in many cases one wishes to use commands where the arguments are stated as CE expressions that are to be evaluated and their respective values are to be used by the definition of the command verb. The following example of a Leonardo system command-line will illustrate the reason for this.

```
[put amber has-examples (get yellow has-examples)]
```

The idea here is that the system shall retrieve the value of the `has-examples` attribute of the entity `yellow` and assign it to `amber,` so that these two concepts will have the same set of examples. The expression `(get yellow examples)` is evaluated and its present value in the current session is used as an argument for the `put` command.

1.3 Mechanisms

Besides the languages that have now been described in gross outline, the Knowledge Representation Framework also specifies the use of a few *mechanisms,* that is, computational artifacts that operate on expressions in these languages. The Leonardo system is an actual implementation of the KR Framework, so it is a concrete KRF mechanism. However, a more abstract and implementation-independent description of KRF mechanisms is useful in order to clarify the meaning and the use of the languages.

KRF mechanisms include *processors* and *engines.* A processor is a free-standing software system; an engine in this context is a program that can be used as a subsystem within an processor. An instance of the Leonardo system is an example of a processor. In general, a KRF processor must be able to create and perform computational *sessions* during which it interacts with a user or with a physical environment, or with other sessions, or with several of these.

A KRF processor can relate in the following ways to the four types of notations and languages that were described above.

- KR Expressions: A session with a processor maintains an internal representation, in the form of datastructures, of information that can also be expressed textually using KR expressions. More specifically, it maintains a mapping that can assign a KR expression in internal form, to any combination of an entity and an attribute.
- Entityfiles: A session can *load* and *save* entityfiles. Loading an entityfile means parsing the textual contents of the file and converting them to the internal representation in the session. Doing so may overwrite information previously contained in the session. Saving an entityfile means producing a textual file containing a serialization of a designated part of the session's information contents. Loading and saving are reciprocal operations.
- Common Expression Language: A session is able to evaluate expressions in the Common Expression Language. Some expressions in the CEL are context-dependent in the sense that their value is obtained from the current contents of the session where they are evaluated.

- Session Command Language: A session is able to execute single commands and scripts in the Session Command Language. The scripts that define non-elementary verbs are stored in the session's information state, which means that they are usually also stored and preserved in entityfiles and entered into the session using a load operation.

Examples of engine types for use in KRF processors include automatic deduction engines, engines for decision-trees and causal nets, and engines for planning a sequence of actions. The lecture notes "Knowledge Representation for Autonomous Intelligent Agents" describe major such engine types using the framework that is described in the present text.

Each engine must be associated with one or a few elementary verbs whereby the engine can be invoked and controlled from a command in the Session Command Language.

The execution of a command (elementary or not) during a session is called an *action* in that session. Processors may differ with respect to how actions are invoked. The following are the major cases.

- *Command-driven processors* operate a command-line loop whereby a user may enter SCL commands for execution by the processor.
- *GUI-driven processors* maintain a graphic user interface where a user can invoke commands by clicking buttons or selecting from menues.
- *Servers* continuously receive commands from a communication network and execute incoming commands.
- *Interactors* exchange commands with other interactors in a network, so they can both execute incoming commands and send commands to other interactors for execution there.
- *Reactive processors* operate on computers or in computer networks that are equipped with sensors for their environment. They contain one or more *sensory engines* that interpret incoming sensor information; they also contain a *decision mechanism* that is able to invoke actions according to information obtained from the sensory engine.
- A *deliberative processor* is a variant of any of the above which is able to use idle time (in the absense of incoming commands or decisions about additional actions) for analyzing its own current 'state of mind,' including its current model of the state of the world and its history of past events, in order to modify and improve its current state. Some methods that are used in artificial intelligence, in particular case-based methods are well suited for being used in this way.

A processor of any of these types is called an *agent* iff it maintains an explicit representation of its own past and future actions, and uses it for some purpose. A processor that merely receives commands and executes them, without keeping any trace of them is not an agent.

Reactive and deliberative agents are often called *autonomous agents,* emphasizing their ability to perform actions "according to their own decisions," and more specifically, their capability for other behavior than what is dictated step by step and by user commands.

1.4 Usage

Processors that are organized according to the Knowledge Representation Framework and using its languages are intended to have significant advantages when implementing knowledgebased systems, such as the cognitive part of an intelligent autonomous robot system, as well as for more mundane tasks such as author-assistance systems for document preparation, or systems for information analysis. (Should write more about this here).

Chapter 2

Knowledge Representation Expressions

This chapter shall define the notation for *Knowledge Representation Expressions*, or KR expressions for short. We first define the basic notation and then a few cosmetic extensions

2.1 Basic KR Expressions

KR expressions are constructed from three kinds of elements:

- *numbers*, which are written in the usual way, for example as `128` or as `3.14`
- *strings*, which are written enclosed by double quotes, for example as `"This is a string"`
- *untyped symbols*, which are written as a sequence of characters that does not contain a space or a double quote, and that can not be interpreted as a number.

For untyped symbols containing interpunction characters (i.e. those that are neither letters nor digits) there are some configurations that are reserved for special purposes, as described in Chapter 4 and Appendix 1. The emerging notation for typed symbols is also discussed in Chapter 4.

Argument Lists and Composite Entities

An *argument list of arity* `n`, where `n` is a non-negative integer, is a sequence of exactly `n` KR expressions, preceded and separated, if necessary ([1]), by at least one whitespace character (space, tab or new line) between successive elements.

Symbols beginning with a colon character (`:`) are called *tags*. An *extended argument list of arity* `n` is an argument list of arity `n`, immediately and

[1] See the Leonardo system documentation for the exact whitespace requirements.

optionally followed by one or more pairs consisting of a tag and a KR expression called the *parameter for* the tag that precedes it. The following is an example of an extended argument list:

```
red big :size 44 :caption "Dynamo"
```

Command verbs and other operators are used together with argument lists, and each operator imposes its particular restrictions on the structure of argument lists that are to be used with it. A *signature* is a collection of information that expresses these restrictions for a particular language and in a particular context. We shall later define how signatures are expressed, and here we just observe that any context where KR expressions are used shall contain a *current signature.*

Symbols are sometimes used as *composers.* The status of a symbol in that respect is represented in the current signature. A tag can not be used as a composer. Each composer is either an *entity composer* or a *record composer.* By convention, entity composers will often be written as symbols whose last character is a colon.

A *composite entity* consists abstractly of an entity composer and an argument list. The current signature specifies what is the correct number of arguments (the "arity" of the entity composer). The signature may also impose restrictions on the arguments themselves, for example, that a particular argument must be a symbol, or must be a number. A composite entity is written as a left parenthesis, an entity composer, whitespace, the argument list, and a right parenthesis.

An *entity* is either an untyped symbol, a typed symbol, or a composite entity.

Composite KR Expressions

There are several other kinds of composite KR expressions, besides composite entities, namely records, sequences, sets, and mappings.

A *record* is similar to a composite entity, but it is formed using a *record composer* rather than an entity composer, it is formed using an extended argument list, and in its textual manifestation it is enclosed in square brackets rather than parentheses.

The main formal difference between composite entities and records is that composite entities can have attributes assigned to them, whereas the same does not hold for records.

A *form* is composed of a *formant* and a (non-extended) argument list, where the formant is an entity and its status as a formant is specified by the current signature. The concrete manifestation of a form encloses the formant and the argument list by round parentheses. Notice that forms and composite entities are different kinds of things. The distinguishing property of forms will be described in the next subsection.

A *sequence* and a *set* is defined as usual in set theory, and are represented by enumerating all the elements. The concrete representations of sequences

and sets use angle brackets, ⟨⟩ ([2]) and curly brackets, {}, respectively, to enclose the representations for the elements, separated by whitespace. It is permitted to separate the elements using commas, but it is not required. We shall sometimes use commas for separating successive atomic elements, but not when one or both of them is a composite expression.

A *mapping* is a set of *maplets* each of which is a twotuple consisting of an entity and a corresponding value, and is written like in the following example:

```
⟦age 46⟧
```

or equivalently as

```
[: age 46]
```

The first notation is for use in publications like the present one; the second one is for computer input where the dotted bracket is of course not available.

The mapping can not contain two maplets ⟦e v⟧ and ⟦e w⟧ having the same first element and different second elements.

An entity e is said to be *defined in* a mapping M iff there is a maplet ⟦e v⟧ in M for some v.

Two mappings are *separate* iff no entity is defined in both of them.

A *KR expression* is either of a string, a number, an entity, a record, a form, a set, a sequence, or a maplet.

2.2 Cosmetic Extensions of KRE

Besides the basic notation that was described above, there are some extensions to the notation that may help to make it even more readable. Some of these will be introduced in later chapters, in contexts where we can see the reasons for them, but some can be introduced here.

The elements of a set or sequence may be separated by commas. This is in accordance with conventional set theory notation, but notice that here it is not necessary, just an option. The use of commas is often natural when the members of the set or sequence are elementary objects, but not when they are composite expressions.

An expression for a mapping consisting of explicitly written maplets, for example

```
{[: johan 23][: berit 25]}
```

can alternatively be written as follows

```
{johan = 23, berit = 25}
```

The latter notation is however not sufficient in situations where one wishes to refer to individual maplets, like in the command example in Chapter 1. The bracket-colon notation is more general.

[2] The software system variant of the KRE notation uses less-than and greater-than characters as angle brackets, in order to remain with the characters on the standard computer keyboards. The publication variant of the notation, which is used here, uses the ⟨ and ⟩ characters instead.

2.3 Embedding of Other Languages

Some applications require the embedding of other languages within KR Expressions. This can sometimes be done by representing expressions in the other language as strings, but doing so is impractical e.g. when the embedded expressions contain double quotes. There is a general notation whereby one writes

```
[$altlan (... expression in the altlan notation) $]
```

This notation allows the embedding of a variety of languages each of which is characterized by its own code to be used as [t altlan], so for example

```
[$tex one may embed text in {\em LaTeX} markup $]
```

(Replace dollar character by paragraph character here) The only restriction on the expression in the embedded language is that it may not contain a paragraph character and a right square bracket in immediate succession. The explicit tagging of the choice of language is helpful for organizing the processing of such data.

Chapter 3

The Common Expression Language

The Common Expression Language contains constructs for *retrieving* information from the current knowledgebase in the session where it is used, and also for *composing and decomposing* KR expressions. To this end it uses two major types of constructs, namely *conditions* and *terms.* Terms have KR expressions as values, conditions have truth-values such as `[true]` or `[false]` as their values. The present chapter defines the syntax for terms and conditions in CEL.

3.1 Terms

Terms are formed using a *term composer* followed by *arguments* and are enclosed in ordinary, round parentheses. *Notice that the term composer is also located inside the parentheses,* which is different from standard mathematical notation.

We mention the following operators as introduction and examples. The separate compendium "Compendium of Programming Techniques for Knowledge-Based Autonomous Systems" contains a list of the term composers that constitute basic CEL.

`(get` ***c a***`)`

Obtains the value of the ***a*** attribute for the 'carrier' entity ***c.*** Ω

`(filemembers` ***ef ty***`)`

Obtains the set of those members of the entityfile ***ef*** whose type is ***ty.*** Type is only defined directly as the value of the `type` attribute and does not allow for subsumption from a supertype. Ω

`(concat` ***s1 s2 ...***`)`

All the arguments shall evaluate to strings; this function obtains the concatenation of the values of the respective arguments. Ω

Use of Infix Notation for Operators

As an additional cosmetic measure, some composers are specified to be *infix composers* and these can *optionally* be placed between the first and the second argument, provided that there are exactly two arguments. It is always possible to let the composer be the first element and thereby to precede the arguments. For example, the term `(a + b)` is the same as `(+ a b)` given that `+` has been declared as infix. The infix specification is part of the signature.

Moreover, if an infix composer admits more than two arguments as well, as is the case for example for `+` then it is possible to use the natural notation where the composer is placed between all arguments, so that e.g. `(a + b + c)` is the same as `(+ a b c)`. Notice that it is obligatory to repeat the infix operator, so `(a + b c)` is not correct.

Notice also that it is not allowed to mix several different operators, prefix or not, within the same parenthesis level. Thus it is not permitted to write `(a + b * c)` and one must write this as `(a + (b * c))`.

The following are some of the infix operators: `+ - * union`. All except `-` allow n arguments in the way just described. The `-` composer can be used with one or two arguments, but not more.

3.2 Conditions

Elementary conditions are called *literals* and are defined as follows.

3.2.1 Positive Literals

A *positive literal* consists of a *predicate* and its *arguments* and is enclosed in square brackets. Usually the predicate precedes the arguments, but like for terms there are some predicates of two arguments that have been declared to be infix operators. For example, one can write `[a = b]` equivalently with `[= a b]` The following predicates are defined in basic CEL.

`[true ]`

This predicate of no arguments always has the value true. Ω

`[false ]`

This predicate of no arguments always has the value false. Ω

`[equal` ***a b***`]`

The literal is true iff the two arguments are equal. Equality between sets is defined so that two set expressions containing the same members but in different order are still considered as equal. Ω

`[member` ***a b***`]`

Membership relation in a set. Ω

`[subseteq` ***a b***`]`

Subset relation between sets. Ω

`[singleton` ***a***`]`

The literal is true iff ***a*** is a set or sequence with exactly one member. Ω

`[hastype` ***a ty***`]`

This literal is true iff the type of the entity ***a*** i.e. the value of its `type` attribute is either equal to ***ty*** or is subsumed by ***ty*** according to the entity subsumption predicate `sub.` Ω

`[sub` ***a b***`]`

The two arguments shall be entities representing types or other things for which a subsumption relationship is defined. The literal is true iff there is a subsumption chain from ***a*** to ***b,*** including the case where the two arguments are equal. Ω

3.2.2 Negative Literals

A *negative literal* is the negation of a positive literal. This is represented by preceding the predicate with a dash character (-). For example, the negative literal `[-sub a b]` is the negation of the positive literal `[sub a b]` A negated literal can be written in infix if this is allowed for the corresponding positive literal.

3.2.3 Composite Conditions

A *boolean composite condition* is formed from positive and negative literals by combining them using the operators `not, and, or` or `imp` and surrounding the expression with round parentheses. The operator `not` has one argument, `imp` has two arguments, and `and` and `or` can have two or more arguments. All except `not` can be written in infix mode.

The negation obtained using the operator `not` is the same as the one obtained by using a negative literal. For example, `[-sub a b]` is entirely equivalent to `(not [sub a b] )` A *quantified composite condition* is formed recursively from the previous types of conditions also using quantified expressions formed as in the following examples:

```
[all :x a :y b ^ ([.x sub .y] imp [.y sub .x])]
[exists :x a :y b ^ ([.x sub .y] and [.x -equal .y])]
```

In general, the quantifiers `all` and `exists` are used as record formers for records with exactly one argument, and with one or more parameters. The tag in each parameter serves as a quantified variable; the corresponding parameter value shall be an entity representing a type that the variable in question will vary over. Notice that the variable symbols are preceded by a *colon* when the variable is bound in the quantifier, and by a *fullstop* (i.e. point) when the variable is used in the quantified expression.

The first example shall thus be read "for all x whose type is `a` and for all y whose type is `b` , `[`x `sub` y`]` implies `[`y `sub` x`]` ".

3.3 The Relationship between KRE and CEL

If we compare the KRE notation of Chapter 1 with the CEL notation of the present chapter, it is clear at once that they use the same syntax. The KRE specifies a few simple rules for the use of round parentheses, square brackets and a few other delimiters, and it turns out that the CEL conforms to all of these conventions. The difference is instead that KRE is *mere syntax* since it only specifies how expressions are to be written, whereas CEL is more specific about the *meaning* of its syntactic constructs as it distinguishes between terms, positive and negative literals, and so forth.

Therefore, although the introduction in the first part of Chapter 1 characterized KRE as one language and CEL as another one, this must be understood as a difference of meaning, or lack thereof, and not as a difference of appearance. The best way to look at this is to think of KRE as a syntax that can be used for a variety of purposes. Representing information in a system's knowledgebase is one purpose, and serving as the syntax for CEL expressions is another such purpose.

This means also that CEL expressions can be stored in the knowledgebase, like any other KR expression. For example, whereas a query like in the example at the beginning of this chapter can be input to a system in the command-line dialog, it can also be stored away as an attribute value for some entity that is a name for the query, for example for the purpose of re-executing it at regular intervals of time.

3.4 Relaxation of Syntax for Publication Use

The syntax for KRE, also used by CEL was defined with two major goals in mind: it should be a systematic and easily readable notation for structured information, and it should be directly usable for input and output to computers using the standard keyboard and character set. In particular it should be possible and convenient to use this as the standard notation in textbooks and articles.

When the notation is used for publication purposes, it may actually be convenient to allow the commonly used symbols for a number of functions and predicates, although they are not available in the computer keyboard. For example, the `subseteq` predicate can be replaced by the commonly used symbol $\subseteq$, and the term composer `union` can be replaced by the symbol $\cup$ as usual.

In publication usage it is not necessary to always insist on full parenthesization of terms. A certain latitude in the use of the notation helps readability and it is insignificant as an obstacle to the easy transfer of definitions and information collections between publications and their use in a computer system. In any case it is natural to require that the computational variety of KRE and CEL shall be defined with full precision, and only to allow commonsense-based exceptions in the publication variety.

Chapter 4

The Session Command Language

4.1 Commands

A *simple command* is written as an expression that is surrounded by square brackets, and consists of a *command verb* followed by *arguments.* The current signature specifies what is its correct number of arguments for each verb, and there are also some verbs that shall not have any argument. The verb `put` shall have three arguments; the verb `creobj` shall have two arguments, for example.

A *composite command* is also written as an expression surrounded by square brackets, but here the first element shall be a *command composer.* The following arguments may be commands, terms or conditions according to the specific rules for each command composer.

4.1.1 Some Simple Commands

A separate memo, "Facilities in Leonardo" contains a list of the command verbs that are defined in the core part of the Leonardo system. The following are some of them to serve as introduction and examples.

`[show` ***a***`]`

Evaluates its single argument and displays it on standard output, which usually is the terminal screen. The use of a single point character (.) has the same effect. This is used in those cases where one wishes to give a form to the command-line dialog and obtain the corresponding value. Ω

`[put` ***c a v***`]`

For the 'carrier' entity ***c*** and attribute ***a*** assign the value ***v*** Ω

`[putsub` ***c a i v***`]`

The carrier ***c*** has a value for the attribute ***a*** that is a mapping. This operation modifies that mapping so that it maps ***i*** to ***v*** Ω

`[addmember` ***s e***`]`

Add ***e*** destructively to the set ***s.*** If it is already a member then no action is taken. Ω

`[output ]`

The arguments are successively evaluated and produced to standard output, with a space character between each argument. Normally, each argument value is written to standard output as it is. By exception, if the value of some ***xi*** is a symbol representing a System Output Phrase (as described in the memo "Facilities in Leonardo") then the phrase for that symbol in the currently selected language is produced to output instead. Ω

4.1.2 Command Composers

The following command composers are defined in the core part of the Session Command Language.

`[soact` ***cmd1 cmd2 ...***`]`

The composer is an acronym for "sequence of actions." The arguments are supposed to be commands and are executed in sequence. Ω

`[coact` ***cmd1 cmd2 ...***`]`

The composer is an acronym for "concurrent actions." The arguments are supposed to be commands and are executed, but in no particular order. Ω

`[if` ***cond cmd1 cmd2***`]`

The ***cond*** argument must be given as a condition expression; the other two shall be commands. The third argument may be omitted. Such a command is executed by first evaluating ***cond.*** If its value is `[true]` then ***cmd1*** is executed, otherwise ***cmd2*** is executed if it is present. Ω

`[repeat` ***i r cmd1 cmd2 ...***`]`

The first argument must be given as a variable; the second argument shall be a set or a sequence. The command executes the command sequence ***cmd1, cmd2,*** etc once for each member of ***r.*** The parameter ***i*** that is given as the first argument is bound to the current sequence member in each cycle. The first argument is used as is and is not evaluated. Ω

`[let` ***cmd1 cmd2 ...***`]`

The arguments of this composer are evaluated in succession. However, the command record may also have parameters, and these parameters are available to the ***cmdi*** during their execution. Ω

`[set-outcome` ***v***`]`

This operation may be used inside a `let` -expression that binds the parameter `:outcome` and it has the effect of assigning the value ***v*** to that parameter. The intention is that different steps during a sequential execution shall be able to assign a value to the eventual outcome from that sequence. Ω

`[with ]`

The first argument shall evaluate to a record; the following arguments are like a case-expression over the composer of the record from the first argument. Each of the later arguments shall have the form `[on` ***c cmd1 cmd2***

etc] The `with` operation identfies the one of the later arguments whose ***c*** component equals the record former. It then executes ***cmd1, cmd2,*** etc. in sequence. The parameters in the record from the first argument are available in the execution of the ***cmdi.*** This construction is useful e.g. when the first argument is the outcome of executing a command, and where one wishes to take different action depending on what kind of outcome record is obtained from that command. Ω

4.2 Definitions of Command Verbs

Command verbs such as those described above are intended to be used for giving commands to a software system, and there must therefore be a definition of the effects for each of these verbs. Such definitions can be made in a few different ways, as follows

- *Elementary verbs* are defined using a piece of program in the system's *host programming language,* i.e. the language that the system is written in. The present implementation is written in CommonLisp.
- *Command composers* are also defined using a piece of program in the host programming language, but they differ from elementary verbs in an important aspect. When the system executes a command with an elementary verb, it will first evaluate the arguments and parameters in the command, and then give control to the verb's definition. For a command composer, on the other hand, the unevaluated arguments and parameters are given to the definition of the composer.
- *Composite verbs* are defined using an an expression in the Session Command Language, i.e. using the constructs that have been defined above. For example, to define the operation `symput` so that `[symput jesper lina spouse]` assigns `lina` as the value of `(get jesper spouse)` and vice versa, one could write the following definition.

```
[def [equal [symput .x .y .a]
            [soact [put .x .a .y][put .y .a .x]] ]]
```

Arguments are referred to using variables in the same sense as the variables that are used in condition expressions, i.e. a symbol preceded by a full stop. A following chapter says more about the naming and binding of arguments.

4.3 The Generic Scripting Language

Several kinds of software systems need to make computation about *actions* and *events,* that is, things that have happened or are expected or planned to happen. This includes intelligent robotic systems that need to anticipate future events in their environment before they actually happen in order to plan their own actions; it also includes discrete-event simulation systems, advanced computer games with an artificial intelligence component, and many others. In fact, it also includes various facilities in the computer's internal operation, for example, for maintaining a log of past user commands and for extracting information from it.

The notation that has been described in this chapter is therefore applicable for a broader range of purposes besides for the command-line dialogue in an interactive computer system. We define the *Generic Scripting Language* as the language described above, but without the particular reportoire of command verbs that we had in a few examples. Several varieties of scripting language can be obtained by changing the reportoire of command verbs, and by extending the sets of term composers and of predicates. The following are some important instantiations of the Generic Scripting Language:

- The *Session Command Language,* SCL, described here.
- The *Robot Scripting Language,* RSL, for representing the actions of a robotic system that operates in a physical environment.
- The *Document Scripting Language,* DSL, for representing the contents of publications and web pages. In this case the 'commands' are formatting commands, for example "produce the text in the arguments using italic font."
- The *Operations Scripting Language,* OSL, for representing commands to a computer on the operating-system level. Thus the OSL is a variety of shellscript language, but using the uniform notational format of the GSL.
- The *Vector Graphics Scripting Language,* VGSL, for defining diagrams defined in terms of line segments and closed areas defined by a set of line segments.

The syntax for the Generic Scripting Language and all its instances is entirely consistent with the notation for Knowledge Representation Expressions that was defined in Chapter 2. This means that the same notation is used for expressing simple, static knowledgebase contents of the kinds that we saw in the examples in Chapter 2, and for the more complex contents that arise when actions, events, and dynamic processes are to be represented.

Command composers such as `if` and `repeat` are considered as part of the Generic Scripting Language, which means that they can be used in all of the language instances.

4.4 Example: Document Preparation

The production of the present report is one example of how this uniformity of representation can be used. The running text in the report is represented using a variety of the Document Scripting Language. However, those pieces of text that define a particular command verb or predicate are located in the source file of the Leonardo software system that implements our approach. All the information about a particular command verb is located together, including its procedural definition which is written in Common-Lisp, the description of the command verb that is included in the present report, comments about the procedural definition which are important in the maintenance of the software, and so forth. The formatting of the report compiles the contributions from the various sources and creates the finished document in `.pdf` form. The implementation of this information architecture is greatly facilitated by the use of a uniform notation and one single, coherent software system.

For this kind of system usage it is essential that same notation, viz. the Common Expression Language can be used for the command arguments in all the command languages in the family. In the example it means that the source text for the present document can contain 'queries' that are evaluated in the session at hand in order to obtain information for inclusion in the document produced.

As another example, if one wishes to produce a well-formatted report of some of the knowledgebase contents, one will use CEL expressions as 'queries' against the knowledgebase, and embed them in a script for the overall document that is written in the Document Scripting Language. Conversely, constituent programs in the Leonardo implementation sometimes use DSL expressions in order to specify explanations of system "errors" or other non-intended outcomes of commands.

t are evaluated in the session at hand in order to obtain information for inclusion in the document produced.

As another example, if one wishes to produce a well-formatted report of some of the knowledgebase contents, one will use CEL expressions as 'queries' against the knowledgebase, and embed them in a script for the overall document that is written in the Document Scripting Language. Conversely, constituent programs in the Leonardo implementation sometimes use DSL expressions in order to specify explanations of system "errors" or other non-intended outcomes of commands.

use CEL expressions as 'queries' against the knowledgebase, and embed them in a script for the overall document that is written in the Document Scripting Language. Conversely, constituent programs in the Leonardo implementation sometimes use DSL expressions in order to specify explanations of system "errors" or other nonintended outcomes of commands.

Chapter 5

Entityfiles and Processors

Each KRF processor maintains a knowledgebase, and this knowledgebase can be understood as an assignment of attribute values to a number of entity-attribute pairs. Such a mapping is called an *information state.* In practice there are two separate manifestations for the processor's information state. There is a *textual manifestation* using files in the computer's file system where each file consists of KR expressions for the attribute-values, together with some surrounding notation. There is also the *dynamic manifestation* as datastructures in the executing program during a session with the processor.

The notation for *entityfiles* is used for the textual manifestation; it has been designed with two considerations in mind: ease of reading for the human user, and ease of parsing and processing in the computer system. The design of the dynamic representation is an issue for each implementation, the only constraint being that it must make it effectively possible to implement the verbs in the Session Command Language efficiently and reliably.

The basic model for the exchange between the textual and the dynamic manifestation is very simple: A session can read entityfiles and construct a dynamic representation with the same contents; it can also produce entityfiles that express a part of the information in its dynamic information state; and finally it can execute commands that change the contents of its information state i.e. the dynamic representation. In one mode of use, entityfiles are loaded at the beginning of a session, the information state is modified using commands, and entityfiles are saved (i.e. written back as entityfiles) at the end of the session. In another mode, the user text-edits the contents of entityfiles and loads them into the session after each edit, and the important commands to the session are those that *use* the information state for some purpose. Other models are also possible.

The basic model is modified in a number of ways, however, and one should think of the load-change-save model as a procedural skeleton that can be modified using a variety of "plug-ins." The present chapter will first describe the syntax that is used for entityfiles for the purpose of the basic model, then describe the structure within entityfiles and methods for organizing large numbers of them, and finally describe some important plug-in facilities.

5.1 Entityfile Syntax

An entityfile consists of a sequence of *entity descriptions* separated by *separator lines.* A separator line is a line consisting of at least four characters of the same kind, which must be a dash (`----`), an equal sign (`====`) or the small letter o (`oooo`), and no other characters. When separator lines are produced in output from the Leonardo system they consist of 57 characters, but this is unimportant; the parser in a KRF processour should accept any line consisting of at least four of these characters as a separator line.

A separator line consisting of small letters o is called an *ending line* and marks the end of the contents of the entityfile, and any material after it is ignored. Separator lines consisting of dashes or of equality signs are treated equally by the parser and the latter are only used for readability purposes as they separate sections within the entityfile.

The first line of the entityfile shall be a separator line, then follows an entity description (unless the first line is an ending line), another separator line, and so forth.

An entity description consists of up to four parts: a *head,* an *attribute part,* a *definition part* and a *property part.* The last two are optional. There must be at least one blank line between each part. (A blank line is a line that does not contain any printing character; it may contain whitespace characters such as an ordinary space or a tab).

The head is a single line consisting of exactly two dash characters, followed by a single space (not tab) character, followed by an entity which may be either a symbol or a composite entity. The entity in the head is the one that is assigned attribute values and property values by the parts that follow.

The attribute part is a sequence of maplets. Each maplet must start on a new line, but maplets may extend over several lines. There is no need to have blank lines between successive maplets.

The optional definition part is used in entityfiles containing programs in the host programming language. In the case of the Leonardo system this is CommonLisp, and the definition shall then be an S-expression that can be evaluated using the standard `eval` function.

The optional property part consists of a sequence of *property assignments* that are separated by at least one blank line between successive property assignments. Each property assignment consists of a *property line* consisting of an at-sign immediately followed by a symbol, and then the *property value* which is an arbitrary number of lines of text (including blank lines), except of course that none of these lines may be a separator line or a line beginning with an at-sign.

Consider an entity description containing the following head

```
-- banana
```

where the following is one of the maplets in the attribute part

```
[: color yellow]
```

and the following is one of the property assignments:

```
@Shape
```

```
Most bananas are banana-shaped.
```

This entity description *corresponds to* an information state in a KRF processor where the value of the CEL expression `(get banana color)` is the entity `yellow` , and where the value of the CEL expression `(get banana Shape)` is the string `"Most bananas are banana-shaped."` There is a convention that attributes are written with small letters, as in `color,` and properties are written with a capital first letter, as in `Shape,` but this is not formally required. A few basic attributes in the type system are also written with a capital first letter.

The meaning of "corresponds to" involves definitions of a number of facilities that are associated with the loading and saving of entityfiles. The basic idea is that when an entityfile is loaded into a session with a KRF processor, assignments are made so that subsequent evaluation of `get` expressions will be as just described. Also, when an entityfile is saved in such a session, it will contain entity descriptions containing maplets and properties with values obtained using the `get` function. Additional aspects of this will be added throughout the present chapter.

If the definition part is present then it will be evaluated when the entityfile is loaded, using the function `eval` in the case of the Leonardo system. The unevaluated definition part will also be preserved in the session so that it can be produced in the resulting file when the same entityfile is saved.

5.2 Entityfile Self-description

The textual manifestation of a knowledgebase consists of a large number of entityfiles, whereas the dynamic manifestation consists of one single datastructure in the executing session. For example the basic `remus` agent, which is often used for clones, consists of around 125 entityfiles. Loading an entityfile means integrating its contents into the dynamic manifestation, therefore, overwriting previous attribute values and property values if necessary, and saving an entityfile means making an excerpt from the dynamic manifestation and converting it to textual form.

How will the save operation know which entity descriptions to include in a particular entityfile? Normally one wishes each entityfile to have an identity of its own and a well-defined set of members, so that it can be saved containing descriptions of the same entities as when it was previously loaded. It must also be possible to add and to remove members during a session, for example if a new entity is defined during the session and one wishes to save it in a particular entityfile.

The following design is used for making this possible. Each entityfile is represented by its own entity, and entities representing an entityfile shall in particular have an attribute called `contents` whose value is a sequence of entities, namely, those entities that are to be included when the entityfile is written. The save operation will therefore take an entityfile entity (i.e. an entity representing an entityfile) *`ef`* as its argument; it will evaluate `(get` *`ef`* `contents),` whose value must be a sequence, and for each member of that sequence it will write its entity description to the file being produced.

Where shall the entityfile entity be saved? Recall that the collection of entityfiles is the only representation of an agent (or other processor) be-

tween sessions, so the entity descriptions for entityfile entities must also be preserved in some entityfiles. The basic convention is that *an entity that represents an entityfile has itself as the first element in the list of contents,* so that it will be saved together with its contents.

For example, consider an entityfile called `colorfile` that shall contain entity descriptions for the two entities `yellow` and `blue` that were mentioned in Chapter 1. The first entity description in such a file should look as follows (with considerable simplification):

```
------------------------------------------------------------
-- colorfile

[: contents <colorfile yellow blue>]

------------------------------------------------------------
```

followed by the entity descriptions shown in Chapter 1. In this way the description of entityfiles is represented like any other information in the knowledgebase, and can be operated on using the same software tools.

Another possible design for this could have been to consider entityfile membership as system information that is not accessible to the user, except through functions whose natural names may be e.g. `add-entityfile-member` and `remove-entityfile-member.` Such a design would be in line with the "encapsulation" style of programming. The KRF does not subscribe to that design philosophy, for good reasons that merit a report of their own.

5.3 Entity Types

In Chapter 1 we showed the entity descriptions for `yellow` and `blue` as follows:

```
------------------------------------------------------------
-- yellow

[: type color]
[: has-examples {dandelion sun amber}]
[: translations {[: french jaune][: german gelb]}]

------------------------------------------------------------
-- blue

[: type color]
[: has-examples {bluebell forgetmenot sky turqoise}]
[: translations {[: french bleu][: german blau]}]

------------------------------------------------------------
```

where in particular each entity has a `type` attribute. The use of a type concept is a standard one and makes no surprise. It has a number of uses, and in particular in the context of entityfiles, type information is used in order to determine which attributes are to be written for each entity. In the present example, the entity `color` should have the following entity description:

```
------------------------------------------------------------
-- color

[: type qualitytype]
[: attributes {has-examples translations}]

------------------------------------------------------------
```

When the save operation writes an entity description for a particular entity (which occurs in the value of the `contents` attribute of the entityfile being written), it therefore looks up the value of its `type` attribute, and then in turn the value of the `attributes` attribute of the type. This specifies what attributes are to be looked up and written for the given entity.

The same applies for entities representing entityfiles, of course, so the first entity description in this file should actually be

```
------------------------------------------------------------
-- colorfile

[: type entityfile]
[: contents <colorfile yellow blue>]

------------------------------------------------------------
```

The definitions of type entities, such as `color` and `entityfile` must also be present in the session information state when the save operation is performed, which means that they must also be located in some entityfile. In some cases it may be convenient to include the type entity in the same entityfile as the instances of the type, in particular if all those instances are located in the same file. This may be the case for `color.` In other cases it is better to collect a number of type-defining entities into a separate "ontology" entityfile.

Entities representing types must themselves have a type, for example the entity `qualitytype` which was used above, `thingtype,` and so forth. The resulting ascending chain is not very long, since one soon arrives at a type that has itself as its type, or to a cycle of two types each of which has the other as its type.

Types are used in several other, important ways; this is a topic for the next chapter.

5.4 Agent Directory Structure

The Knowledge Representation Framework specifies the notation to be used within entityfiles. Although each entityfile may contain a large number of entities, the number of entityfiles is also substantial in a practical system, since already the `remus` agent contains around 125 entityfiles, and nontrivial applications require several times that. It is therefore necessary to assign some structure to the collection of entityfiles, at least in order to facilitate for the developer that is working with them and needs to keep track of them. There are several methods for doing this, additional methods emerge, and different implementations may do it in different ways. However, some

specific methods will be briefly mentioned in order to give a first idea of this topic.

The organization of entityfiles has to take into account that there is both a textual and a dynamic manifestation of a knowledgebase. The most obvious organization of the textual manifestation is to partition the entityfiles into groups and to assign each group into its own subdirectory. The textual manifestation of the entire processor is then a directory structure, with one top-level directory that represents the processor as a whole, and with one or more levels of subdirectories that ultimately contain the entityfiles that make up the processor.

It is not entirely obvious whether the dynamic manifestation of the knowledgebase needs to assign structure to the collection of entityfiles, except to the extent that is needed in order to load and save entityfiles in their proper subdirectories. However, it seems natural to have such a structure, and to design it in such a way that it is coordinated with the directory structure in the textual manifestation. The Leonardo system therefore uses a concept of a *knowledgeblock* that has entityfiles as members, just as entityfiles contain entity descriptions.

Another method, which complements the use of knowledgeblocks, is to organize an application in terms of several agents each of which has its own entityfiles for both "programs" and "data." If a particular task requires the combined activity of several agents then they will communicate by message-passing. If different tasks are done at different times then the application may be held together simply by the use of shared data.

5.5 Compiled Entityfiles

Interpretive progamming languages such as Lisp, Perl or Python are attractive choices for implementing a KRF processor since by their design it is easy to integrate programs and data. The present section applies to KRF implementations in such languages, including of course the Leonardo system.

In interpretive languages it is usual that both programs and data can be expressed as text files that essentially contain a sequence of commands in the language in question, including commands to store particular data in the system's database, and commands that assign a definition to a particular function or command word. Programs and data are text-edited by the user and entered into a session using a load operation. The KRF processor design uses the same method, but adds to it by also having a uniform procedure whereby such files can be saved from a session. This additional facility was introduced by the Interlisp system and is not present in CommonLisp.

When a KRF processor is implemented in an interpretive language, each session must start by reading a few files that are expressed in the host language, including a file containing a parser for KRF entityfiles. Only when the parser and other basic facilities are in place is it possible to start reading entityfiles using the syntax described above.

At the same time it is a basic design goal that all information in the system shall be organized in the same way, so that it can be processed using the

same routine. Then it is undesirable to have certain startup files that differ from all other files in the system.

The solution to this problem is obtained using a method that also serves some other purposes at the same time, namely, the use of *compiled entityfiles*. A compiled entityfile is a file that contains the same information as an ordinary entityfile, but expressed using commands in the host programming language, so that it can be loaded directly by its interpreter. Compiled entityfiles are much more cumbersome to read than standard entityfiles, so they are not suitable for human use, but they serve an important purpose during startup.

Compiled entityfiles are produced by the entityfile save operation, so that when it writes an entityfile it may do so both in standard form and in compiled form. These are then two separate files with the same file name but different extensions: `.leo` for the standard form and `.leos` for the Lisp-compiled form. However, the compiled form is only needed for a minority of the entityfiles, so it would not be meaningful to produce them for all files. Instead, entityfiles have an attribute `has-savestyle` which specifies the saving policy. If the value of that attribute is `leo-savestyle` then only the standard file is produced, if it is empty then both standard and compiled files are produced. There are also other options.

The startup sequence for a session is defined so that it first loads a number of compiled entityfiles, and then proceeds to loading ordinary entityfiles. The `has-savestyle` attribute controls this choice. However, the `loadfil` command which the user can issue during a session will load the standard representation and not the compiled one. Therefore, if one wishes to change something in the session startup procedures, this is done simply as follows: (1) text-edit the entityfile in question to obtain the desired changes, (2) load that file, (3) save it, thereby producing a revised version of the compiled file.

Another aspect of compiled entityfiles is that they load much faster than the corresponding standard form. In those cases where one has to deal with very large entityfiles that change infrequently, it is therefore convenient to store them in both standard and compiled form, usually load the compiled form into sessions, and only use the standard form when some of the contents have to be changed by text-editing. Furthermore, if changes are done using a graphic user interface to the session, so that the dynamic manifestation of the data is modified directly, then one can dispense with the standard form altogether and use only the compiled form.

To give some idea about what data volumes can be supported with this approach, the largest application of the Leonardo system is the Common Knowledge Library which contains more than 60.000 entities, and where the largest entityfile alone contains around 8.000 entity descriptions, each with a number of attributes. The compiled form of this entityfile loads in 5 - 10 seconds depending on the computer used.

5.6 Entityfile Attached Facilities

The basic structure for entityfiles is designed for, and defined by the interplay between the textual and the dynamic manifestation of a knowledge-

base, as implemented by a processor in the KRF sense. In addition there is a large and growing reportoire of additional services that are attached to the entityfile machinery. The following are some examples in order to give a first idea of what can be done there. For details, please refer to the report "Facilities in Leonardo."

- Each entityfile entity has an attribute `latest-written` whose value is a timestamp that is reassigned each time the file is saved.
- An entityfile may contain entities whose type is `section.` The effect of doing this is that the following entities, up to the next `section` entity or the end of the file are grouped together as a section within the file. Sections can be referred to, and for ease of reading the save operation shows them separated by lines made of equality signs rather than dash signs, for increased emphasis.
- There is an operation that sorts the entities in an entityfile alphabetically, but if the file has sections then the sorting is applied separately to each section.
- Each entity may have an attribute `in-categories` whose value is a set of symbols. These symbols may be assigned freely and can be used for putting "flags" on the entity.
- The entityfile load operation checks the value of the `in-categories` attribute of the entity representing the entityfile, and for each element in that set it checks if that element has a particular attached procedure, and executes it if so. This makes it possible to specify operations that are to be done after a particular entityfile has been loaded.
- It is possible to attach a loading procedure to property names, for example a procedure that parses the corresponding property value according to a particular syntax.
- A property assignment beginning with the a line containing `Log` but preceded by *two* at-signs has the effect that when the entityfile is loaded, the corresponding property is accumulated to a separate *log file* for changes in that particular entity, and then the property is removed. This is convenient for making notes about software changes: one can write the note near to where the change was made, but one does not have to be bothered by long change logs in the files one is working with.
- The *phrase facility* arranges that phrases that are to be printed at a point in the program that is defined by some entity, can be written as properties of that entity, rather than in-line in the code itself. This facility gives support for defining each phrase in several natural languages, so that the entire system can be switched easily from one language to another.

Chapter 6

Some Knowledge Representation Issues

This chapter will briefly discuss a few issues relating to the representation of knowledge in the KRF framework. A more extensive coverage of this subject is provided in the lecture notes "Knowledge Representation for Intelligent Autonomous Agents" which is one of the continuations of the present report, as well as other and more specialized documents.

6.1 Ontologies and Types

As defined in Chapter 5, each entity in an information state must have a value for the attribute `type` . The value of this attribute shall in turn be an entity, which means that it also must have a type, and so on. The Knowledge Representation Framework does not allow any exceptions to the rule that every entity shall have a type, but the resulting chain consisting of types of types is never very long since one arrives quickly at a supertype that has itself as its type.

The purpose of the type information is syntactic rather than semantic: it is used for specifying restrictions on the choice of and the values of attributes, but it does not say very much about the meaning of the entity. However, each type entity (and some others) may also have an attribute `subsumed-by` whose value is an entity for another, more general concept. For example, a simple knowledgebase might contain one entity for the type `swedish-person` and another one for `scandinavian-person.` Both of these types may in turn have the type `person-type,` but in addition there is a subsumption relation between the two person types. This subsumption relation is expressed by assigning `scandinavian-person` as the value of the `subsumed-by` attribute of `swedish-person.` Subsumption chains can be significantly longer than subtype chains, and are likely to change more often in the lifetime of a system, for example by insertion of intermediate nodes in the chain from time to time.

The information in the subtype and subsumption hierarchies represent the beginnings of an *ontology.* Each Leonardo system contains and maintains an ontology that consists of at least two parts: a *self-description ontology* for

organizing the types of entities that make up the software of the Leonardo system itself, and a *cognitive ontology* which defines abstract and general concepts on the top level of the subsumption and subtype chains. In addition there is an on-line resource called the *Common Knowledge Library* which provides a library of ontology modules that can be downloaded to, and imported into a Leonardo system in order to provide it with domain-specific information.

The subsumption structure is the more expressive one from the application point of view, but the type structure serves an important purpose as a way of defining and checking the formal correctness of information sets of nontrivial size. In particular, the Common Knowledge Library is organized as a collection of *knowledge modules* that are checked for formal correctness by special software. Just like each entity has a type, so also each knowledge module has a supermodule which specifies the correct form of the contents in the given module, and the supermodule has *its* supermodule in turn, until arriving at a module that is its own supermodule. The software checks that knowledge modules that are published in the CKL are consistent with the requirements of their respective supermodules.

A separate report, "The Leonardo Ontology" describes these matters in more depth.

6.2 Uses of Logic Expressions

Chapter 3 defined the syntax for condition expressions which are formed starting with positive and negative literals, and combining them using propositional ("boolean") connectives and quantifiers. Condition expressions can be used for the condition part of `if` expressions and `those` expressions, for example, but they have also other uses. In particular, although information about an application can in principle always be encoded using entities and attribute-values, it is sometimes more convenient to describe an application as a set of *propositions,* that is, expressions formed using literals and combining them just like condition expressions. In fact, we shall use the term "proposition" as the general term and we will consider "condition" simply as a synonym that is preferred when the proposition in question occurs in an expression that is used computationally.

A further use of propositions, or conditions, is for expressing the *preconditions* of command verbs and other verbs. Please recall that the definition of a command verb will often be an `if-then-else-` expression where the condition (the `if` part) checks whether the desired operation is possible and appropriate, the `then` part performs the desired operation, and the `else` part explains to the user (or to surrounding code) why the operation could not be performed.

The separate handling of conditions/propositions in the KRF framework makes it possible to define command verbs in such a way that the `if` part and the `then` part are defined separately from each other, and the `else` part is omitted or greatly reduced. The `if` part is called a precondition and is expressed as a proposition, with exactly the same syntax as was described for conditions in Chapter 3.

The Leonardo processor uses this technique in some parts of the system and

the intention is to introduce it throughout. When it receives a command whose verb has been defined in this way, it first checks whether the precondition is satisfied for the given arguments and in the session's current information state. If this is the case then the separate `then` part is executed, otherwise the system will take appropriate action based on its knowledge of which part of the precondition was not satisfied. If the appropriate action is to explain the problem to the user, then the evaluation of the precondition and information that is associated with each predicate makes it possible for the system to give reasonable "error" information.

Another possible response from the system, if the precondition is not satisfied, is to try to fix the problem by taking autonomous action in order that the precondition *shall be* satisfied. Yet another response may be to look for a substitute action, by first trying to identify what was the user's goal when requesting the action question, and then finding another action that would achieve that goal instead. These are classical topics in the field of Artificial Intelligence. Our point here is that by organizing a software system in the way described here, it becomes possible to bring down these A.I. techniques to the level of common and mundane computer use.

6.3 Using Logic for Knowledge Representation

The use of predicate logic for knowledge representation purposes is a large topic, and it is a major topic in the textbook "Knowledge Representation for Intelligent Autonomous Agents." Here we shall only say a few words about how the use of logic fits into the various aspects of the Knowledge Representation Framework.

The basic idea in the Framework is to organize the knowledgebase in terms of entities and their attributes and properties. The basic idea in Logic is to organize known facts as one single set of propositions, each of which makes a true statement about the world. However, in practical use it is anyway convenient to organize large numbers of propositions into groups, partly to facilitate their management, but sometimes also because one wishes to only use a subset of the known facts or "axioms" for the purpose of drawing conclusions or answering questions.

The natural way to place logic-expressed information in a KRF framework is therefore to make natural groupings of the given information, to associate each such grouping with a suitable entity, and to write it either as an attribute value or as a property value of the chosen entity.

The use of logic also calls for procedures or 'mechanisms' (in the sense of the word from Chapter 1) for drawing conclusions, finding the answers to queries, and other logic-based tasks. The development and incorporation of such mechanisms is an ongoing task in the Leonardo system.

6.4 Using the Expressiveness of KRE and CEL

A few notes about how Knowledge Representation Expressions and the Common Expression Language can be used.

6.4.1 Forms vs Composite Entities

The KRE syntax makes a distinction between forms and composite entities which is useful both for use in attribute-value representations and logic-based representations of a domain. Suppose for example that `father:` is an entity composer, while `father` is a formant that is associated with an evaluation rule mapping the entity for a person to the entity for that person's father, and suppose also that `lars` represents the father of `per` . Then the data expressions to the left in the following table *evaluate to* the data expressions to the right:

```
(father: per)                  =>   (father: per)
(father per)                   =>   lars
(father: (father per))         =>   (father: lars)
(father (father: per))              can not be evaluated
<(father: per) (father: maria)>     evaluates to itself
<(father: per) (father per)>   =>   <(father: per) lars>
```

It is intended that an expression such as `(father: per)` can be used to represent the *concept of* Per's father, and that it can be used for example as a component when expressing "Gunnar knows who is Per's father". That is why an expression such as `(father (father: per))` does not make sense. The introduction of entities such as `(father: per)` is known as *reification.*

6.4.2 Parameters in Command Expressions

Since commands are expressed as KRE records, they may contain both arguments and parameters. The use of parameters has already been exemplified in some of the command composers in Chapter 3. However, parameters may also be used for a variety of purposes in simple commands. Sometimes it is useful to use parameters instead of arguments, for example in order to have a mnemonic tag on the argument, or if an argument is optional.

Additional uses occur when parameters are used for indicating *how* or *for what reason* an action is performed, similar to the use of adverbials in natural language. Consider the following example.

```
[to-achieve [insert key-4 lock-12]
            [turn-clockwise key-4]
            [remove key-4]
       :goal (not [locked lock-12]) ]
```

This expression is intended to specify a goal – the lock called `lock-12` should be in an unlocked state – and a method for achieving the goal. Notice that the goal is a proposition; this is yet another example of the use of propositions in the system. In many cases it is natural to write the parameters before the arguments, and in such cases they are by convention separated using the uparrow or circumflex character, as follows:

```
[to-achieve :goal (not [locked lock-12]) ^
            [insert key-4 lock-12]
            [turn-clockwise key-4]
            [remove key-4] ]
```

Appendix: Notational Details

This Appendix describes notational details that have practical and technical relevance, and is intended for reference if specific questions should come up in these respects.

6.5 Reserved Morphologies for Entity Names

The basic rule for entity names is that they may consist of any (finite and non-empty) sequence of printable characters except those that are used as delimiters in composite expressions (the double quote, the parentheses, the square and curly brackets, and the less-than and greater-than characters) and also excepting the colon, the comma, and the backquote. The last three are excluded due to the special treatment of these characters in Common-Lisp. This is an effect of having used it as the host language in the first implementation. An exception to this rule is made for having a colon as the first character in a tag.

The phrase "printable character" excludes of course the space character, the tab, the carriage-return and new-line characters, and others that are even more obviously excluded.

Some additional restrictions must also be imposed. Any sequence that can naturally be interpreted as a number can not be used for an entity, for example `3.14` or `666` can not be used for entities.

The use of CommonLisp as an implementation language means that there may be occasions where the system is requested to print out a datastructure that is not a correct encoding of a Leonardo structure, although it is still correct from the Lisp point of view. In such cases the system will write, for example

```
[$ (this is a lisp list) $]
```

Insert paragraph signs instead of dollar signs here. The KRE parser will accept such an input and will produce the equivalent Lisp list structure again. If the paragraph character is used in an application, it should make sure not coming in conflict with this convention.

All constructs that satisfy these restrictions should be able to pass the Leonardo parser for KRE expressions. Some additional conventions are implemented in the parsing process and therefore become obligatory. Any

entityname that begins with a full stop, and that has not already been classified as a number and not an entity, will be considered as a variable by the KRE parser, and should only be used in that way, for example in GSL scripts and in quantified propositions.

The symbol consisting only of the circumflex character ^ should not be used in applications, since it is used in the extended syntax for records as a separator between parameters and arguments in those cases where one wishes to write the parameters before the arguments.

A notation for typed entitynames is described below.

Besides these restrictions that are imposed by the KRE parser in the present system, there are also a few conventions that are imposed by various applications on the internal representation that they receive after parsing. The following is the current list of such special conventions.

- The special meaning of preceding a predicate symbol with a dash in order to indicate negation has already been described. It is not implemented in the KRE parser, but is applied afterwards by the facility for evaluating and reasoning with propositions.
- An implementation of decision-trees assigns a special meaning to entitynames having the question-mark as their last character.
- Symbols of the form dddd-dd-dd, such as 2010-02-18 are reserved for being used to represent dates in the Christian Era (Anno Domini) chronology, Gregorian variant. Other chronologies may be represented as e.g. BE-2542-02-18 for the Buddhist Era.
- Symbols such as `w.Frankfurt` having a short prefix that is attached using a full stop are used in information analysis facilities for marking a symbol as used by another party. For example, the prefix `w.` expresses that it is the entity that is called `Frankfurt` in the Wikipedia.
- Symbols such as `Andersson.Sven` or variants thereof are used for representing persons. The variants include using a postfix for distinguishing between several persons having the same name, for example `Andersson.Sven.3` and having a prefix separated using a slash character for indicating a domain of some kind where the naming and the numbering is applied, for example `se/Andersson.Sven.3` showing that the numbering is done in the Swedish domain.

It is advisable to keep morphological rules such as these to a minimum, and to use composite entities instead if at all possible. However, these examples show that the use of composite entities is not always convenient.

This list may be extended at later times.

6.5.1 Typed Entitynames

There is often a need to distinguish in a systematic way between different meanings of the same word in natural language. Typical examples are for `orange` as a fruit or as a color, or for `china` as a country or a material.

None of the possible approaches to this problem is ideal, and the present Leonardo system does not contain a definite solution. One possible solution

is to specify the existence of separate *namespaces* and to prefix every symbol with an identifier for the namespace where it belongs. For each namespace there must be a coordinator that takes responsibility for the assignment of symbols in that space. In particular, the XML community uses Internet URL's as namespace identifiers, as a way of assuring the uniqueness of each such identifier. This approach has advantages in the construction of very large systems that include many stakeholders, but it has the disadvantage that expressions tend to be large and difficult to read, and we prefer not to use this approach.

We make instead the assumption that Leonardo knowledgebases will be developed in distinct, small communities and that each such community can organize its naming conventions for itself although within the Knowledge Representation Framework with suitable adaptations. Communication of knowledgebase contents between such communities willl then sometimes require some systematic transformations, for example with respect to entitynames, but we believe this can be managed easily.

Even within such a user community there is the problem of how to select entitynames when the English-language word has several distinct meanings all of which are of interest for the knowledgebase, like in the examples above. This calls for a choice between two approaches: one can either require that distinct names are to be introduced and used in all textual data, for example in entityfiles, or one can arrange that the same name is used for all the meanings and the disambiguation is done by the parser of the textual data.

The proposed textual representation in the first case is to use entities of the form

```
orange[color]
orange[fruit]
```

as part of the basic syntax. These will then be distinct entities both internally and externally, and the type assignment to each of them shall be implicit in the entityname. However, the present Leonardo parser does not support this convention.

If the disambiguation is to be done by the parser then there must be some surrounding information that makes this possible. An obvious way of doing this, although it will not work for all cases, is to use type information for entities and attributes. Consider for example the following entity description

```
-------------------------------------------------------------
-- orange

[: type fruit]
[: has-colors {red orange yellow}]

-------------------------------------------------------------
```

in a system where it is previously known that the name `orange` can refer both to a fruit and to a color. If this entity description is loaded into that system, for example because it is included in an entityfile that is reloaded in its entirety after some editing, then it is clear that the first occurrence of `orange` can immediately be disambiguated. Furthermore, if there is type information to the effect that the value of the attribute `has-colors` shall be a set of entities whose type is `color` then the second occurrence can

immediately be disambiguated as well.

The current Leonardo parser contains a facility for disambiguating entity names in the way that is shown in this example. It uses an internal representation where the disambiguated entities are called `orange#fruit` and `orange#color` in the example. The corresponding printing (serialization) facility produces the untyped name. This facility is in experimental use, in particular in the system for the Common Knowledge Library, and it is still too early to evaluate the experience from using it.

As long as neither of these approaches is in regular use, the remaining possibility is for each application to introduce its own conventions for situations such as these. An obvious choice is to use symbols such as `orange.fruit` or `orange/fruit` while recognizing that these are unrelated symbols from the system point of view, so that it is the responsibility of the application to implement any synonym management and any disambiguation that may be needed.

6.6 Whitespace Requirements and Restrictions

Whitespace characters are defined as the ordinary space character, the comma, and the carriage-return and newline "characters" The tab character is not a whitespace character, and it *may not be used anywhere* in input or in entityfiles. Whitespace characters are by definition ignored in input, except inside strings and properties. They must however be used in some particular positions.

The first line of an entity-description, immediately following the line of dashes, shall consist of exactly two dashes, followed by exactly one space character, followed by the entity being defined there. There must not be any character, whitespace or otherwise, before the two dashes.

Within KRE expressions, and except for inside strings, it is necessary to have a least one whitespace character for separating two atomic items that appear in immediate succession. Atomic entities, numbers and strings count as atomic entities.

Furthermore, in maplets there must be at least one whitespace character between the introductory `:` and the following entity.