CASL

Cognitive Autonomous Systems Laboratory **Leonardo**
Department of Computer and Information Science
Linköping University, Linköping, Sweden

Erik Sandewall

The Leonardo Representation Language Part I: Entity Descriptions

Core Design, Version 0.2

This project memo pertains to the development of the Leonardo system. Identified as PM-leonardo-001, it is disseminated through the CAISOR website and has URL http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/001/

Related information can be obtained via the following www sites:

CAISOR website:	http://www.ida.liu.se/ext/caisor/
CASL website:	http://www.ida.liu.se/ext/casl/
Leonardo system infosite:	http://www.ida.liu.se/ext/leonardo/
The author:	http://www.ida.liu.se/~erisa/
Date of manuscript:	2007-01-22

Abstract

The Leonardo representation language is a textual representation of information structures based on *entities* that have attributes and properties. Attributes are structured objects formed using composition of sets, sequences, records, and a few other constructs. Properties are textual objects that can be used to harbor whatever representation language is needed by an application. A distinguishing feature of this language is that entities can be composite objects, and are not restricted to be atomic symbols. The representation language is used for a wide range of purposes, ranging from low-level operational needs in the Leonardo system, to ontologies and other knowledge representation tasks. The present report defines the core part of the Leonardo representation language.

Note on Methodology

Along with the work on the Leonardo system, its languages and applications, we have also reflected on the choice of research methodology for projects such as this, that is, projects where a relatively complex system design is developed through several iterations of system development, and where the design itself is the primary outcome of the project. A concurrent report([1]) describes our view of methodology and its implications for the structure of system documentation and of publications that is needed in order to pursue and to communicate the research.

The documentation for Leonardo is a collection of interrelated documents that follow the approach that is proposed in those methodology reports. In particular, the purpose of the present report is to define the central part of Leonardo's representation language. A discussion and analysis of various choices in its design, and a comparison with other, more or less related work is to be made in a separate report. Likewise, it is intended to have another separate report for various "features" of the representation language that have a marginal character from a structural point of view. The implementation status of software supporting the representation language is of course changing over time, and is indicated on the webpage for the present report([2]).

Examples of use

The Leonardo Representation Language is designed to be used in text files that can be relatively large, and that should be suitable both for a human reader and for computer processing. Some examples of such files are referenced from the webpage of the Leonardo project ([3]). In order to make full use of these examples, it is recommended to also study the report on the Leonardo Computation System ([4]) which describes the larger context within which the representation language is used.

[1] `http://www.ida.liu.se/ext/caisor/pm-archive/morador/001/`
[2] `http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/001/`
[3] `http://www.ida.liu.se/ext/leonardo/`
[4] `http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/002/`

1 Introduction

This section provides an introduction to the Leonardo representation language and its use in the Leonardo system, as well as to the character of the system and to the overall goal of the project where the language and the system are developed.

1.1 The Representation Language

The Leonardo representation language allows the user to express information that is associated with *entities.* The following is a very simple example of an entity-description in Leonardo, for an entity called `acm`:

```
-----------------------------------------------------
-- acm

[: type organization]
[: sections {sigplan sigact sigsim}]
[: fullname "Association for Computing Machinery"]

@Headline
The First Society in Computing

@Autodescription
Founded in 1947, ACM is a major force in advancing
the skills of information technology professionals
and students worldwide. Today, our 80,000 members
and the public turn to ACM for the industry's leading
Portal to Computing Literature, authoritative
publications and pioneering conferences, providing
leadership for the 21st century.

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

In this example, whose textual contents are from the ACM website, the `type`, `sections`, and `fullname` elements are called *attribute assignments*, and the `Autodescription` and `Headline` elements are called *property assignments.* Properties can be used for storing plain text, like in this example, but they can also be used for storing definitions e.g. in a programming language or a grammar language.

Information structures that are similar to this simple example have been developed and used for a long time and in several branches of computer science. The following are some additional properties that are specific to the Leonardo representation language:

- Attribute values can be formed recursively by composition using set formation ({ ... }), sequence formation (< ... >), and several other, similar constructs. Our representation language stays close to standard set-theory notation. The elements of these expressions can be entities, strings, numbers, of formatted text.
- Entities may be expressed as single symbols, such as `acm` in the example, but they can also be composite expressions. For example, the

information about the membership in `acm` by a particular person, denoted `erik`, could be expressed as shown below, where `(membership: erik acm)` is a composite entity having three attributes. The values of the latter two attributes are *records*, which is an additional construct in the Leonardo language, besides sets, sequences, and formation of composite entities.

```
----------------------------------------------------
-- (membership: erik acm)

[: type membership-info]
[: annual-due [amount: USD 120]]
[: paid-until [date: 2006 09 30]]

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

Types are first-class citizens, so symbols such as `membership-info` and `organization` in these examples are also entities that can be associated with attributes and properties.

The Leonardo representation language is closely connected to several software facilities of widely different character:

- It is the framework and the representation for all programs in the Leonardo system, including both its own implementation and applications that are built on it;
- It is used for expressing an ontology that serves as the cohesive framework for all information in the Leonardo system and its applications;
- The representation language contains a sublanguage for markup of text; the Leonardo system contains subsystems for text formatting.

The close integration of these aspects of the system provides particular power and simplicity, for example in the following ways:

- The text formatter can be used for generating typographic quality presentations of all software in the system
- Expressions in the Leonardo evaluation language ("programming language") can be embedded in texts, and are evaluated by the formatter
- The ontology is available for program code on all levels, including for the implementation of Leonardo itself
- Support software that operates on the Leonardo knowledge structure (which is expressed using entities and their associated information) can be used for all kinds of data, including programs
- Every piece of software in the system is an object in the system ontology, providing control information for the various general-purpose service functions that operate on the knowledge structure.

For example, an entity-level version management system provides version control services when several developers cooperate on a joint project based on the Leonardo platform. The version management system is part of the system core, and uses itself the Leonardo representation language for keeping track of the administrative information it needs.

1.2 Language Architecture

One single notation can not serve all purposes, and like in many other comprehensive systems we work with several sublanguages that can be embedded inside each other, or interconnected in various ways. The global structure of these languages is referred to as the *language architecture*.

One of the sublanguages in Leonardo has a primary status, namely the *Leonardo Data Expression Language*, LDX, which is used for the attribute sections of entity descriptions. There is a number of *associated languages* which can be used in attribute values of the kinds that were shown in the examples above, or in subexpressions inside attribute values. The conventions for how to "wrap" expressions in associated languages will be specified in Section 5. HTML, XML, and LaTeX markup notations are associated languages in this sense. It is intended that the Leonardo system shall provide parsed representations of those embedded expressions, at least as an option, and not merely represent them as plain text.

Other languages, for which wrapping conventions and structured representations have not been defined, will be referred to as *foreign languages*. They can be used freely in properties of entities, similar to the `Autodescription` property in our first example above, but the Leonardo system does not provide any support for them.

Some of the associated languages may be *reciprocally associated* with LDX, in the sense that not only can LDX expressions contain expressions in those languages, but they can also contain expressions in LDX, even recursively, and such multi-language aggregates are supported by the Leonardo implementation. The structure consisting of LDX and its reciprocally associated languages is referred to as the Leonardo Representation Language, LRL. Reciprocally associated languages have particular requirements since a reasonable implementation of Leonardo will have to treat all of them as well as LDX in an integrated way. Foreign languages, and languages that are not reciprocally associated can much more easily be treated as separate implementations. For these reasons it is natural to refer to LRL is *the language of* the main Leonardo system. LDX and the reciprocally associated languages are called the *sublanguages of* LRL.

To explain the concept of reciprocally associated languages we quote the example of Latex which has general Latex language and the reciprocally associated language for markup of mathematical formulas. Each of them can embed expressions in the other one, recursively.

For all associated languages, even if they are not reciprocally associated, the Leonardo system must be able to transport variable bindings inwards through their wrapping for the purpose of substitution.

The implementation of Leonardo at the time of writing does not actually give full support for any sublanguage besides LRX, since the others are still in the stage of language design. We are working on two such languages, namely ST for *structured text* and IF for *informatic formulas*, including expressions in logic. ST can be compared with Latex and with the textual-markup aspect of XML. IF is used for representing formulas in logic, for database queries, and a few similar purposes. IF actually contains LDX as a subset so no wrapping is needed for embedding an LDX expression in IF, but full IF inside LDX needs to be explicitly wrapped. This will be further described in Section 5.

1.3 The Leonardo System

Although LDX can be parsed and used in any computational context, our emphasis is on the use of LDX in a *Leonardo system* which is an environment for the development and use of Leonardo based applications. The main activity in the Leonardo system is to *execute actions.* In this respect it differs from other incremental computing systems, such as in Lisp, Prolog, or Perl, where the main activity is to evaluate expressions. The emphasis on the execution of actions means that the use of agendas and plans is built into the basic system design, as well as facilities for remembering and referring to past actions.

Each implementation of Leonardo will of course be made in a *host language.* The host language of the present major implementation is CommonLisp, but the definition of LDX is intended to be neutral with respect to the choice of host programming language. Partial implementations have been made for PHP and Python in order to obtain some experience of language independence.

Although implementations in e.g. Java and Perl would also be of interest, our longer-range goal is to implement Leonardo on a much more elementary platform, so that many services that are traditionally provided by operating systems and database systems shall be performed within Leonardo itself. Leonardo has been designed with that goal in mind. A very minimal, Scheme-like language would be an appropriate basis for Leonardo.

1.4 Implementing Applications on the Leonardo Platform

Applications on a given software platform typically require the implementor to define both data structures and procedures. For applications built on Leonardo, both declarations of data structures and actual data contents are expressed in LDX to the largest possible extent. Procedures are written as Leonardo entities, with the full text of the procedure as one of the properties of the entity naming the procedure. Information that is intermediate between 'declarative' and 'procedural', such as grammar rules or robotic behavior rules, can also be stored using the properties of entities.

Documentation of the various aspects of an implementation are naturally integrated with the implementation, using additional properties. It is intended that utilities in the Leonardo system shall allow documentation to be generated by composition of documentation properties of all entities in a given structure, using the built-in text formatting services.

1.5 The Goal of Software System Consolidation

The development of the Leonardo representation language as well as the Leonardo system is part of a broad research effort whose overall goal is to identify a *consolidated* software architecture, that is, an architecture that eliminates the considerable conceptual redundancy that is characteristic of contemporary software systems. The redundancy that we have in mind is the one where similar constructs are used in programming languages, operating systems, database systems, text formatters, www servers, webpage scripting languages such as Javascript, modelling systems such as UML, knowledge representation languages, and so on.

We believe that it should be possible to design the total system in such a way that the same concepts and constructs are used for all the facilities that are today offered by separate software systems such as those mentioned, with considerable gains in cost of development, ease of learning and of use, and cross-utilization of services. We propose to use the term *consolidation* for this review and revision of contemporary software technology, and for which Leonardo is the experimental vehicle.

2 Data Expressions

The initial example showed how the description of entities has an attribute part using structured expressions and a property part allowing for free text. We begin by defining the representation that is used in the attribute part. Expressions there are called *data expressions.* The definitions in this section are recursive, so by necessity some definitions use terms that are defined later on in the section.

2.1 Entities and Argument Lists

Data expressions are formed recursively from a small number of element types, namely symbols, strings, and numbers, and from wrapped expressions in associated languages, for example for marked-up text.

A *string* is written enclosed by double quotes, for example `"This is a string"`. It must not contain a double quote and not any non-printing character other than space and newline. In particular, tab characters are not allowed. There is also no escape character, for example for including a double quote within a string. For such purposes one can use one of the text markup languages that are associated with LRL. A string always means the sequence of characters between the double quote characters: `"There are 38 characters in this string"`.

A *number* is written as a positive or negative integer in the obvious way, or as a decimal number e.g. as `3.14159` for π. Other kinds of numbers, such as fractions, are expressed using records which will be defined below.

An *untyped symbol* is written as a sequence of characters in the standard, 8-bit ascii alphabet, not containing any of the following: the double quote, whitespace characters, control characters, other non-printing characters, or the following *bracket characters* which are reserved for the formation of composite expressions:

```
( ) [ ] { } < >
```

Also, a full stop (`.`) can not be used as the first character of a symbol, since this is how variables are formed in the informatic formula languages (see section 5.2). Finally, a sequence of characters that can be interpreted as a number can not be used as a symbol.

Additional restrictions on the choice of characters within symbols may occur in a particular implementation. A sequence of characters between ascii 32 and 127 that conforms to the above restrictions must always be allowed, however.

A *typed symbol* consists of an untyped symbol, a left square bracket, an entity, and a right square bracket. It is intended that the entity within brackets shall indicate what type is intended for the symbol. For example, one may write `china[material]` and `china[country]` to distinguish the two major meanings of the English-language word "china".

An *argument list of arity* n, where n is a non-negative integer, is a sequence of exactly n data expressions, separated if necessary ([5]) by at least one whitespace character (space, tab, or new line) between successive elements. A *tag* is a symbol whose first characer is the colon (`:`). A *parameter* is a tag followed by a data expression, with at least one whitespace between them. An *extended argument list of arity* n consists of an argument list of arity n, optionally followed by at least one whitespace and one or more parameters using different tags, again separated by one or more whitespace.

Some symbols are used as *composers*. The status of a symbol in that respect is represented in type describing information states, which will be introduced in section 4. By convention, entity composers are written as symbols whose last character is a colon (`:`). Each composer is either an *entity composer* or a *record composer*.

A *composite entity* consists abstractly of an entity composer and an extended argument list whose arity is the arity that the entity composer has according to the describing information state. Specific entity composers may impose restrictions on their argument lists, for example, that a particular argument must be a symbol, or must be a number. In the concrete syntax, a composite entity is written as a left parenthesis, an entity composer, whitespace (except if the argument list is empty), the extended argument list, and a right parenthesis.

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

2.2 Composite Expressions in LDX

There are also several other kinds of composite expressions in LDX, 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, and in its concrete manifestation it is enclosed in square brackets rather than parentheses.

The main difference between composite entities and records is motivated by implementation and is analogous to the difference between symbols and strings. In the internal representation of data expressions in an implementation there should be a unique data object representing a composite entity with a given composer and a given extended argument list, but there can be several instances of a record with given elements. Composite entities can have attributes and properties assigned to them, but records can not.

[5]See section 2.3 for the whitespace requirements.

Examples of the use of records have been shown above, although the records in those examples did not have any parameters.

A *form* is composed of a *formant* and an extended argument list, where the formant is an entity and its status as a formant is specified in the describing information state. The concrete manifestation of a form encloses the formant and the argument list by round parentheses. Forms play a special role for the *evaluation* of data expressions, described below.

A *sequence* and a *set* is defined as usual in set theory. The LDX language allows one to define sequences and sets by enumerating all the elements. The concrete representations of sequences and sets use angle brackets ([6]) and curly brackets, respectively, to enclose the representations for the elements, separated by whitespace. Commas are not used between the elements.

A *mapping* is a set of *maplets* each of which is a twotuple consisting of an entity and a corresponding value. The current implementation considers maplets as records of two elements with `:` as the record composer, for example

```
[: age 46]
```

It is foreseen that future implementations may wish to view maplets as a type in its own right, separately from records, while however retaining the same syntax.

The *value* of a data expression is defined as follows. If an expression is not a form and does not contain any form as a subexpression, then it evaluates to itself. The value of a form is obtained using an *evaluation rule* that is associated with the form's formant. Expressions that contain forms but are not themselves forms evaluate to a structure of the same kind as themselves, but where each element has been replaced by its value, recursively.

For example, if `father:` is an entity composer and `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 if `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))               is a type violation
<(father: per) (father: maria)>      evaluates to itself
<(father: per) (father per)>    =>   <(father: per) lars>
```

It is intended that an expression such as `(father: per)` shall 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 evaluation rules for formants may be partial, so that for some argument lists they do not provide any value. In such cases the form remains unevaluated, in the style of partial evaluation. For example, if the father of `per` is unknown to the evaluator, then the data expression `(father per)` evaluates to itself.

[6]The less-than and greater-than characters are used as angle brackets, in order to stay within the 8-bit ASCII character set.

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

2.3 Special Cases for Brackets and Whitespace

Data expressions are formed using elementary constructs that are composed using whitespace characters and the four types of bracket characters. In addition we need a few rules for special cases which are actually due to the multiple use of some characters.

A left square bracket is usually followed by a symbol or a form, or (in the IF language which is a superset of LDX) by a variable. The character immediately following the left square bracket can therefore be a left round parenthesis, a full stop (.), or a non-bracket character. These possibilities do not allow the left bracket to be followed by, for example, a less-than character or a curly bracket.

However, a complication arises in IF where atomic propositions are written as records, with the record composer as the predicate. For example, "Lars is older than John" may be written as `[older lars john]`, and "*a* equals *b*" is written as `[=` *a* *b*`]`. It is therefore natural to write "*a* is less than *b*" as `[<` *a* *b*`]`. Similar expressions apply for "greater than", "less than or equal" (<=), and "greater than or equal" (>=). In brief, if a less-than or greater-than character appears immediately after a left square bracket, then it is considered as an ordinary, symbol-forming character and not as a bracket character. This convention is introduced already in LDX in order to anticipate the forthcoming generalization to the Informatic Formula languages.

The complete rule for the character sequence `[<` actually contains one more variety, which will be specified in section 5.3.

Due to the great use of bracketing symbols and the relatively small number of actual bracket characters, we anticipate that at some point, at least in IF languages, it will be necessary to introduce *composite brackets*, for example as in

```
[* ... *]
```

The following rule is made to avoid ambiguity due to the use of forthcoming composite brackets. Whitespace is always required for separating two syntactic elements each of which is either a symbol or a number. It is also required between a left bracket character and a symbol *except* in the following cases:

1. If the character immediately following the left bracket is a letter or a digit (the normal case) or a stop character (.)
2. If the bracket character in question is a left *square* bracket and the character immediately following it is a colon, the equality sign, or a less-than or greater-than character.

For example, one must write

```
< *abc* *def* >
```

for a sequence of two elements where the first one is `*abc*`, since

```
<*abc* *def*>
```

becomes ambiguous if `<*` is defined as a composite bracket. – An analogous rule applies before a right bracket character.

The inclusion of the colon character in the second item of this rule is because of the syntax for maplet expressions, where a single colon is considered as a symbol and a record composer. It follows, unfortunately, that a number of constructs that could otherwise have been nice-looking composite brackets will not be usable in that way, for example the following ones:

```
[:    ...    :]
[.    ...    .]
```

3 Entity descriptions

Entity descriptions were introduced by the example already in section 1. They are the basic building-block of the Leonardo representation language, making it possible to assign attributes and properties to entities, and to store algorithms, axioms, documentary text, and other information pertaining to those entities.

An entity description is a textual object consisting of three parts: a *heading*, an *attribute part*, and a *property part*. The heading identifies the particular entity to which attributes and properties are to be assigned. This entity is called the *head* of the entity description. The syntax of the heading has been designed with ease of reading in mind, so that the separation between successive entity descriptions shall stand out clearly.

The syntax of the attribute part is fully defined by the Leonardo Data Expression Language, LDX, together with its associated languages. The syntax of the property part, on the other hand, consists of a few general conventions that are very simple, and that are complemented by the detailed syntax of the particular language for each kind of property, which can be either an associated language or a foreign one, for example, a programming language, or a variety of logic.

3.1 The Heading

The heading of an entity-description consists of three lines, and can be formed in a few different ways. Usually it consists of one line consisting only of dash signs, one line consisting of two dashes and a space, followed by an entity which is the head of the entity description, and finally a blank line. The purpose of the entity description is to assign attributes and properties to the head.

The first line of the heading must contain at least five dashes, but usually one puts so many that they fill the line.

In an alternative case([7]), the second line of the heading looks like in the following example:

```
== waypoint-12 (waypoint: [geo-coordinate: 42645 678 25])
```

[7]Not yet implemented at the time of writing this report.

In this case there is a composite entity that is formed using the entity composer `waypoint:`, and that takes a record as its single argument. This record presumably specifies a particular geographical position using its x, y, and z coordinates. However, it may be inconvenient to write out this composite entity repeatedly as a subexpression in other expressions, and the heading therefore specifies that the symbol `waypoint-12` can be used as a synonym for that composite entity. The other parts of the entity description assign attributes and properties to this expression, as usual. This representational facility is referred to as the *naming* of composite entities.

There is yet another syntax variant for headings, but one which is purely cosmetic, namely that the first heading line may be a sequence of equality signs instead of dashes. This has no semantic significance, and is used for separating groups of entities graphically in a printout representation.

3.2 The Attribute Part

The attribute part of an entity description consists of a sequence of maplets, written according to the following conventions:

1. Each maplet begins on a new line and starts in the first character position of that line.
2. If a maplet extends over several lines, then each of those lines besides the first one must begin with one or more whitespace characters (space or tab characters).

3.3 The Property Part

The property part of an entity description, finally, consists of an optional number of *property assignments*, with an optional number of blank lines before and after each assignment. Each property assignment consists of a *property line*, followed by an optional number of *content lines*. The property line consists of a *stop character* which is usually the `@` character, followed by a symbol that tags the property. Each one of the content lines consists of arbitrary printing characters or space and tab characters in the standard 8-bit ascii character set, *except* that the first character in a content line must not be the stop character.

It is recommended to have at least one blank line before the first property assignment, between property assignments, and after the last one. These blank lines are not considered part of the respective property values.

A property can not contain empty lines, but it can contain lines consisting entirely of whitespace. Empty content lines in an entity description that is input to the Leonardo system are ignored. (This rule applies in the present implementation but it has turned out to be inconvenient, and it may change in the future).

Normally, `@` is used as a stop character throughout entity descriptions. Special notation is foreseen for choosing another stop character, for the case where one needs to represent properties where some of the content lines begin with `@`.

4 Meta Level Information about Entities

4.1 Types of Meta Level Information

The two previous sections specified the *syntax* for expressing the descriptions of individual entities. Several kinds of meta level information is also required:

- *Catalog* information specifying how entity descriptions are grouped together into larger aggregates and where those aggregates are stored in the computer system at hand.
- *Taxonomic* information whereby a type is assigned to each entity and a hierarchical structure of type subsumption is defined.
- *Type extent* information, which specifies what attributes are appropriate to use for each type.
- *Arity* information for composers and formants.
- *Structure specification for attribute values*, specifying what structures are permitted for attribute values.
- *Global structure specification*, specifying restrictions on the permissible structures in a knowledge base as a whole.

The structure specifications for attribute values may request, for example, that the value of the `has-children:` attribute of an entity of type `person` or its subtypes shall be a set of entities each of which has the type `person` or one of its subtypes. The global structure specification, on the other hand, may for example specify that the sequence formed by following successive `has-children:` links from one person to the next, is not allowed to contain any cycles.

In Leonardo the catalog, taxonomic, type extent, and arity information is considered as one group, and structure specification for attribute values and for the global structure is considered as a second group. We refer to the first group as "catalog and type specification" (CAT) and to the second group as "structure specification". The term "syntax" is used for the conventions that were described in the previous sections, so CAT and structure specifications are not considered to be part of the syntax.

The reason for the separation between CAT specification and structure specification is that CAT can be conveniently expressed in LDX itself, whereas structure specification requires the use of the informatic formula language, IF, which is a superset of LDX. CAT information is important for the functioning of the most basic parts of the Leonardo system and is therefore built into its core. Structure specification, on the other hand, does not have the same central importance and can be considered as an add-on facility.

4.2 Entityfiles

Entities are grouped together for management purposes, forming *entityfiles*. All information that defines a Leonardo system, and all other information that it needs to retain persistently, is represented using entityfiles. Each entityfile has a name, which is again an entity. The name of the entityfile has an attribute called `contents` whose value is a sequence of the entities that are members of the entityfile in question. By convention, the name of the entityfile is always the first element in that sequence, so that the entityfile contains the description of itself.

The term "entityfile" is used for two reasons. It really refers to a "file" in the original sense of a sequence, but in the present implementations most entityfiles are represented as files in the sense of the operating system, containing a sequence of entity descriptions in textual form.

Although the discussion of design alternatives is a topic for another report, we shall anyway make a brief digression of that kind here. The use of a single operating-system file for the entity descriptions in a sequence of entities, represented using the LDX language, is not the only possible approach. Another possibility is to have a separate OS file for each entity-description. This is convenient for cases like entities representing books and articles in a library, if one usually only addresses a small number of those at the same time, but it becomes inconvenient if one needs a large number of entity-descriptions in memory at the same time for the purpose of search.

The concrete work with building a number of applied systems using the Leonardo system as a platform has led to several additional structures for the persistent storage of entity descriptions. These solutions are based on pragmatic considerations, such as the desire to have files of reasonable size, as well as considerations that are due to the conceptual structure and the common-sense naming conventions that apply in each application. Our plan is to make a retrospective on the implementations that have actually evolved in several such applications, and to use it as a base for a solution that is both general-purpose and well-adapted to actual needs.

4.3 Information States

The concept of information state is introduced as an auxiliary concept for the definitions concerning types and arity, and is derived from the definition of entity descriptions. We first define an *entity-state* corresponding to a particular entity description as an alternative representation of its attribute part. It is a maplet of the form

```
[: e {[= a1 v1][= a2 v2] ... [= an vn]}]
```

where `e` is the entity in the header of the entity description, and the subsequent maplets are those that occur in its attribute part. For example, the entity-state of the entity description for `acm`, in the example at the beginning of this report, is as follows:

```
[: acm {
    [: type organization]
    [: sections {sigplan sigact sigsim}]
    [: fullname "Association for Computing Machinery"] }]
```

An *information state* is a set of entity-states, and it is therefore a mapping. For a given entityfile F, the *corresponding information state* is the set of the entity-states corresponding to the entity descriptions in F.

If M is a mapping (in the sense of LDX) containing a maplet of the form `[:` a v`]` then $M(a)$ is interpreted as v. Furthermore, if S is an information state then $S(a, c)$ is defined as $S(a)(c)$. For example, if F is an entity-file containing the entity description for `acm` shown above, and if S is the information state corresponding to F, then S(`acm,type`) = `organization`.

4.4 Type Membership and Type Extent Information

Type membership information and type extent information for entityfiles is defined in terms of the corresponding information states, as follows.

An information state S is said to be *typed* iff S(`e,type`) is defined for every maplet `[:` e m`]` in S.

An entity t is said to be a *type in* an information state D iff D(`t,type`) equals `leotype` or an entity that is a subtype of `leotype` according to D. The definition of the expression "subtype according to" will follow below. Its definition and the definitions given here are recursively interdependent.

An information state D is said to be *type describing* iff it satisfies the following conditions:

1. It is typed.
2. For every entity t that is a type in D, D(`t,attributes`) is defined and its value is a set or sequence of entities.
3. For every entity t that is a type in D, if D(`t,subsumed-by`) is defined, then its value is an entity that is again a type in D.

An information state S is said to *conform to* a type describing information state D iff it is typed and the following conditions apply for each entity-state `[:` e m`]` in S:

1. S(e`,type`) is a type according to D.
2. For every maplet `[:` a v`]` that is a member of m, a is a member of the set D(S(e`,type`)`,attributes`)

An entityfile F is said to *conform to* a type describing information state D iff its corresponding information state conforms to D.

In principle it is possible to let each entityfile F rely on another entityfile C for its CAT specifications, so that the CAT-level correctness of F is defined as conformity with the information state corresponding to C. In practice, however, one should think of the total information structure in an activation of the Leonardo system (that is, an executing instance of the software) as a type describing information state. This information state is constructed and maintained by "loading" a number of entityfiles in the course of a computational session, and each time that an entityfile is loaded it has to conform to the current information state in the activation, or otherwise some corrective action is taken. Reciprocally, the entityfiles that the activation is able to produce to files, or as outgoing messages, shall be expected to conform to its current information state.

One of the advantages of this language design is that we do not need to introduce a separate data description language; the meta-information is represented in LDX itself. For example, one can impose the requirement on the current information state in a Leonardo activation that at each point in time (except in the midst of certain transitions) it shall be type describing and it shall conform to itself.

4.5 Type hierarchy

If *D* is a type describing information state and *t* and *t'* are types in *D*, then *t* is said to be a *subtype of t' according to* D iff either *t* = *t'* or there is a sequence $t_0 = t, t_1, ..., t_{k-1}, t_k = t'$ such that $t_{i+1} = D(t_i, \texttt{subsumed-by})$ for all applicable *i*. This is the definition that was needed in the previous section in order to close the definition recursion.

4.6 Catalogs

Catalog information is important in the Leonardo system, and is organized as follows, in the present implementation. We already mentioned that each entityfile is represented by an entity that serves to name the entityfile. Let *f* be such an entity. The symbol `location:` is an entity composer which is used so that `(location: f)` is an entity having an attribute `filepath` whose value is a string representing the path, in the sense of the OS file system, for the file containing the textual representation of the entity descriptions for the entities in *f*.

One may wonder why this attribute is not assigned to *f* itself. The reason is that the entity description for *f* is within the file itself, so it is not available at the point where the Leonardo software wishes to load the contents of that entityfile. The entity descriptions for entities of the form `(location: f)` are kept in separate catalog entityfiles.

5 Wrapping of Associated Languages

In the interest of simplicity it is desirable to have a uniform wrapping convention for all associated languages in LRL. The present section specifies the wrapping convention and gives an overview of the major associated languages.

5.1 General Wrapping Convention in LRL

LRL uses a convention where each associated language obtains an entity serving as its name, for example `xml` or `tex` ([8]), and where an expression in the associated language `html`, for example, is written in wrapped form as

```
[§html ...  §]
```

The character sequences `[§` and `§]` are allowed within the wrapped objects provided that they are used recursively to embed and wrap subordinate objects, for example for embedding HTML within Latex. They are not allowed to occur in other ways. For any other purpose, occurrences of these

[8]We use `tex` for Latex and `TeX` for plain TeX code.

pairs of characters should be split up using whitespace or rewritten in some way which may be specific to the associated language at hand.

This convention was chosen since the § character is rarely used, and in those cases where it is used it is often expressed by a coding convention anyway. The cases where a wrapped expression absolutely needs to contain the character sequence `[§` or `§]` for its own purpose seem to be rare, therefore.

5.2 Sublanguages for Informatic Formulas

The formal languages that are used in computer science differ in several ways from those used e.g. in mathematics. The names of functions, variables, etc in mathematics tend to consist of single characters, possibly with subscripts or superscripts, and the number of such symbols is fairly limited in any given context. The formalisms in computer science, including programming languages, markup languages, and so on, use large numbers of symbols which therefore have to be represented by 'identifiers', i.e. sequences of characters. We shall refer to this kind of formulas as *informatic formulas.*

Several kinds of informatic formulas are needed in the Leonardo Representation Language, in particular for the *evaluation language*, for *scripting languages*, and for varieties of *logic* that are used for declarative purposes. LDX itself does not meet the needs of these, in particular since it does not have a variable construct. The evaluation language is used for expressions that can be evaluated in specific situations, for example when a webpage is displayed on the screen, or when a document is formatted, or when an action is executed. Scripting languages are special-purpose languages that are designed for a specific range of use and that are often closely tied to a particular software.

Leonardo uses parenthesized expressions in the Lisp tradition for its evaluation language, and it is recommended to use such expressions for other informatic formula languages as well. This means that expressions in an informatic formula language consist of elements that are symbols, strings, numbers, or wrapped expressions in other sublanguages, and these elements are composed recursively using expressions of the form

```
(operation argument argument ... argument)
```

The evaluation sublanguage is denoted by the symbol `eval`, so that a wrapped expression in the evaluation sublanguage can be written as

```
[§eval (function arg arg ...  arg)§]
```

The evaluation language and the representation of predicate logic as an LRL informatic language use the same notation for variables, namely, as an identifier beginning with (`.`), the period character. The binding of variables is to be retained through a wrapper. For example, if an expression in the evaluation language binds a variable `.x` and contains a wrapped expression in a markup language, then it shall be possible to use `.x` (suitably wrapped, typically as `[§eval .x§]`) inside the markup expression, for the purpose of substitution.

5.3 Markup of Text

The markup languages for structured text are important examples of associated languages. The syntactic convention for wrapping allows for several markup languages to be used, including Latex, HTML, and XML (with some restrictions). In addition there is an ”in-house” markup language, called ST (for ”Structured Text”) which has the particular advantage of being reciprocally associated, so that it can in turn embed expressions in LDX and in the evaluation language.

Conversion between these markup languages is supported, in particular from ST to Latex and HTML. The use of multiple markup languages may seem like a violation of our main goal with respect to consolidation of software technology, namely, to eliminate notational and conceptual redundancy. Indeed, in an ideal system we would like to only use ST since it is reciprocal. However, we also wish to be able to use Leonardo for a range of experimental applications, and then it is necessary to support languages such as XML, HTML, and Latex.

Using the syntax for wrapping of associated languages, LDX provides a “shell” within which one can have a choice between several alternative markup languages. The Leonardo Representation Language considers expressions in these as *text objects* for which there are the following representational varieties initially.

```
[§ect ...  §]                    for escape-coded text
[§tex {...} §]                   for latex coded text
[§html <tag> ...  </tag>§]       for an html expression
[§xml <tag> ...  </tag>§]        for an xml expression
[§st <style ...  >§]             for an expression in ST
```

Here, ‘escape-coded text’ means plain text which has been augmented with a simple coding convention for special characters, but without any other formatting facilities. Escape-coded text is a subset of ST.

Although HTML, Latex etc are not reciprocally associated with LDX and we do not support embedding of LDX code in them, at least the syntax allows reciprocal embedding *between* the various markup languages for text, so that Latex code can be embedded inside HTML code for example.

For notational convenience, and specifically for those cases where markup expressions appear directly inside LDX expressions, we also define the following concise formats.

```
["..."]                  for escape-coded text
[{...}]                  for latex coded text
[<style ...>]            for an expression in ST
```

The full representation using `[§` and `§]` is necessary when expressions in different text languages are nested inside each other, whereas the concise variants are sufficient when text-markup expressions occur as attribute values in LDX.

This abbreviation for an expression in ST is somewhat at odds with the notation for expressing a less-than relationship as a record, which was described in section 2.3. The full rule for the very special case of a left square bracket character, immediately followed by a less-than character, must therefore be as follows: If the right square bracket character that follows and matches the left square bracket at hand is immediately preceded by a greater-than sign, then the entire sequence

```
[< ... >]
```

is considered as an abbreviation for

```
[§st < ... >§]
```

and otherwise the less-than character in question is considered as an ordinary, symbol-forming character and not as a bracket character.

For text objects that are stored as files the formatting convention is obtained from the file extension.

5.4 Outline of the ST Formatting Language

The ST markup conventions provide a counterpart of the core Latex functionality but with a cleaner integration of formulas, as well as the core of HTML functionality. We do not address all of an XML counterpart here since the data-structuring aspects of XML are taken care of by LDX.

ST builds on the AML markup language that was developed and implemented by the present author in an earlier project. The revision of the language design and the implemention are still in progress, so the following is only an outline. The full specification of ST is intended to follow later.

Every *ST object* has the form

```
<[sr: ...]  co1  co2 ... con>
```

where each subexpression `coi` can be either plain text, a new ST object, or a wrapped expression in another LRL associated language, and `sr:` is a record type for a *format record*, a particular kind of record that specifies some aspect of the structure or appearance of the text. The following is a simple example:

```
<[style: :emphasize t :font ariel] emphasized text
            in ariel font>
```

In particular, simple formatting operations such as specifying emphasized or boldface style are viewed as abbreviations of format records. Thus in the ST object

```
<e this phrase will be typeset in italic font>
```

the initial `e` is interpreted as an abbreviation for

```
[style: :emphasize t]
```

which is a simple format record that specifies emphasized font, thereby overriding the style that is inherited from superior level objects.

The following is an example of a nested ST object using one format record for specifying a www link to the Leonardo website, and inside it a format record abbreviation specifying emphasized (italic) text. HTML-like conventions are used:

```
<[weblink: "http://www.ida.liu.se/ext/leonardo/"
      :target "sepwin"] The <e Leonardo> system.>
```

An additional convention is introduced purely for convenience when reading the marked-up text, so that the examples above can be written equivalently as follows:

```
<style/ emphasized text in ariel font [:emphasize t :font ariel]>
<weblink/ The <e Leonardo> system.
    ["http://www.ida.liu.se/ext/leonardo/" :target "sepwin"]>
```

In this way the details of the markup appear after the text that they apply to, rather than before it, which seems to be much easier to read.

5.5 Wrapping the LDX Language

Since ST is reciprocally associated with LDX, it must be possible to include wrapped LDX expressions in it. This can be done using the general wrapping convention, as in the following example

```
[§ldx <[amount:  USD 2060][amount:  RUR 14000]>§]
```

LDX expressions for sets and records can in fact be included in ST without explicit wrapping, since ST reserves square and curly brackets for that purpose.

6 Discussion of the LDX Language Design

A more general discussion of the design decisions in the LDX language is the topic of a separate report. This is in line with the methodological policy that was mentioned initially, and where we wish to distinguish between the *specification* of a design and a *discussion and analysis* of the design, its evolution, and its relation to other designs addressing similar goals. However, although the present report is intended as a language specification, for the most important aspects of the language, it may still be of interest to include a brief discussion of some aspects of the language design.

6.1 Entityfiles

The term ‘file’ in ‘entityfile’ is chosen both for the original meaning of a file as a ‘sequence’, and since each entityfile can obtain its textual representation as one file in the sense of the operating system. The latter aspect is important in the present implementation of Leonardo that relies on a conventional operating system, but it will be reconsidered in future implementations where the functionalities of files and file directories are handled within Leonardo itself.

The textual form of entityfiles has a number of uses:

- As a presentation of the information structure for the user, to be read both on the screen, and in printouts on paper;

- As a mechanism for persistent storage: the information in the current Leonardo system is kept in entityfiles between runs;
- As a tool for large-scale editing operations: systematic changes in many entity-descriptions at once can sometimes be done conveniently using a text editor and the textual form of the entityfile;
- As a mechanism for modularization and for transfer of content between different instances of the Leonardo system, for example, for different users.

The syntax for entity-descriptions represents a compromise between the needs of these different usages. For example, it is the presentation aspect that motivates the choice of syntax for the attribute section of an entity-description, otherwise we could have used entity-states directly.

In principle it is possible to implement a Leonardo system in such a way that entity descriptions are stored in the system at all times, as a kind of database. In practice, it is however convenient to set things up so that entities have a dual representation: both as datastructures in the current information state of the running software system, and a textual form using the syntax for LDX and the other syntactic conventions that have been introduced here.

6.2 The Treatment of Types

Section 4.1 introduced the distinction between the catalog and type specification (CAT) and the structure specification, where only the former is part of the Leonardo kernel. This may seem strange from the perspective of conventional programming languages, where a type system specifying the admissible structure for objects in each type is often considered as fundamental to the language design. The two questions that may then be asked concerning the design used here, are: (1) is there a useful use of the kernel system that relies only on the CAT specification, and (2) are there problems that arise in the subsequent addition of a facility for structure specifications, in the sense of section 4.1, when they are not included in the core of the system?

The first question can be answered in the affirmative based on the experience with the actual Leonardo system. What happens, of course, is that one uses the programming style that is characteristic of interpretive programming languages, and where checks for correct structure are performed dynamically, for example on entry to a function or other computational unit.

For the second question we do not have the same grounds for an affirmative answer at this point. The problems that one may think about are those relating to performance and those relating to safety and to the completeness of the type checks. With respect to performance, it may be noted that some “interpretive” programming languages, such as CommonLisp, report performance in compiled mode that compares well with what is obtained in conventional typed languages. The question of the completeness of the type checks is one that we leave open for the time being; we do not consider it to have the highest priority.

6.3 Locality of Names

We have described how software in Leonardo is organized as a set of entity descriptions, where each entity has a name. This suggests that all entity-names are globally defined during their use in a Leonardo system. On the other hand, one of the major reasons for having a hierarchical structure in a conventional, algorithmic programming language is to provide locality for names. In another vein, OWL and other languages defined by the WWW Consortium provide a mechanism where symbols are labelled with an identifier for the vocabulary where the symbol is defined. One may ask whether important facilities such as these have been lost in the Leonardo approach.

We have already described one, partial solution to this problem, namely the use of typed symbols which was introduced in section 2.1. In this respect, please notice that the type indication in a typed symbol can be a composite expression; it does not have to be a single, untyped symbol. This provides considerable expressivity, as illustrated by the following example:

```
atlanta[(and city (located-in: georgia[state]))]
atlanta[journal]
```

Here, the city of Atlanta in the US state of Georgia is distinguished from the Atlanta Magazine. Just characterizing the first Atlanta as a city will not be sufficient if there is an Atlanta in some other state, and the use of the typed identifier `georgia[state]` is useful ([9]) to distinguish from the country of Georgia in the Caucasus.

The construct of composite entities provides an even more versatile facility for dealing with multiple uses of a particular symbol. Suppose, for example, that one wishes to distinguish between the hand of a person and the hand of a clock as two different types. One may then decide to use an entity composer `as-part-of:` whereby one can write `(as-part-of: person hand)` and `(as-part-of: clock hand)` for these two kinds of hands. Both these composite entities will then be declared as types and have attributes assigned to them accordingly.

Similarly, if two different ontologies `ultimate-onto` and `universal-onto` have different definitions of the concept of ‘citizen’, then one can refer to one or the other as `(as-defined-by: ultimate-onto citizen)` and `(as-defined-by: universal-onto citizen)`. If in addition there are different definitions of what `ultimate-onto` is then the first argument of `as-defined-by:` can be another `as-defined-by:` expression.

It is our hypothesis that the existence of these very general constructs in LDX make it unnecessary to introduce separate facilities for defining the scope or the vocabulary that a symbol belongs to. Experience with concrete applications will serve to confirm or refute this hypothesis. Following the principle of Ockham’s razor, we do not wish to introduce additional constructs if it is not clear that they are needed.

[9]... although probably not indispensable, if one can assume that there is no city called Atlanta in the country of Georgia

7 Directions for Continued Work

This report has specified the syntax for expressions in the Leonardo Data Expression Language (LDX) which is the leading sublanguage of the Leonardo Representation Language (LRL). It has also briefly outlined the structure of some of its associated languages. Additional material is being prepared in several forthcoming reports that proceed in the following directions:

- The representation of entityfiles and aggregates of such files, and their use for version management, generation of documentation, and the like.
- Static knowledge representation that uses entity descriptions as its base, for example, the representation of taxonomies and part-whole relationships.
- The representation of time, actions, and events, and its use for simulation and cognitive robotics systems.
- The structure of the Leonardo executive, and how it uses entity-descriptions as well as action/event structures as its operational data structure.
- Implementation of LRX and of the basic layers of the Leonardo executive and evaluator in various computational environments.
- Applications of Leonardo, for example for the management of documents and websites.
- Discussion and analysis of the design decisions and the revisions of them.

These reports will be posted on the Leonardo website ([10]), together with updated versions of the present report. Some of the material will take the form of HTML pages, and may not necessarily have the appearance of a traditional report. Please use and cite that site for access to the latest information about the system.

8 Acknowledgements

The work on Leonardo has been partly supported by the Knut and Alice Wallenberg Foundation through the WITAS project, and by the INRIA (France) for a stay as a guest researcher. This support is gratefully acknowledged.

[10] `http://www.ida.liu.se/ext/leonardo/`