CASL

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

Erik Sandewall

Leonardo Document Preparation Facility

This project memo pertains to the development of the Leonardo system.
Identified as PM-leonardo-010, it is disseminated through the CAISOR website
and has URL `http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/010/`

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:	2010-05-26

Introduction

The Leonardo Document Preparation Facility is a part of the Leonardo system that supports the authoring of articles and reports, and of static and dynamic webpages. It is closely integrated with the system and most aspects of it are included in the system kernel such as the `remus` agent. The present document describes this facility (henceforth called just the Facility) with the goal that the reader shall be able to use it effectively for her or his own needs.

The goal is for this facility to be as systematic as possible and to avoid redundance e.g. through the emergence of multiple subsystems with similar functionality. At the same time, it has been developed in a demand-driven fashion by extending it with new capabilities when there was a need for it in our own usage of the system. This means that the design is not always as clean as one would desire, and in particular the reportoire of formatting operators is not entirely systematic. We expect to rectify this in due course, but for the present report the priority is to describe the system as it is, and not as we intend it to become.

This said, there are also some minor details where the need for local improvements were noticed when the report was written, and where we have taken the liberty to violate the just said rule by describing an option that is not yet implemented. There should be a separate list of such exceptions, and it should be very short. Reader, be warned.

The following report should be read before the present one in order to have the required background concerning notations and systems:

Erik Sandewall. Knowledge Representation Framework: Overview of Languages and Mechanisms.

The following two reports by the same author are also useful for background:

Leonardo Installation, Startup and Self-Description

Compendium of Programming Techniques for Knowledge-Based Autonomous Systems. Part I: List Processing

Chapter 1

Organization of Documents and Webpages

The Leonardo Document Preparation Facility is a part of the Leonardo system that supports the authoring of articles and reports, and of static and dynamic webpages. The basic idea is to use languages in the Knowledge Representation Framework as a markup language that can be translated to LaTeX and to HMTL, where the LaTeX representation can in turn be used for generating the final document in PDF form.

The present document describes the Leonardo Document Preparation Facility and will refer to it briefly as the Facility. We shall describe the markup language, the directory structures that are used for keeping documents and webpages, and the various command verbs that can be used when authoring.

The primary way of using the Facility is through the Leonardo command-line executive (CLU) together with a text editor. There is however also a GUI interface that can replace a large part of the use of the command-line executive. This will be described in a separate chapter, but it has not yet been tested and used extensively.

The present chapter will describe the directory structures being used and the linkage to auxiliary software outside Leonardo.

1.1 Directory Structure for Documents

By 'document' we here mean articles, reports, textbooks, etc. which are to be produced in PDF format. The emphasis is on documents with predominantly scientific and technical contents.

The Facility presumes that each document obtains its own directory under the operating system. This directory will then contain source files for the text, separate files for diagrams in a variety of formats, for example `.eps` , intermediate files, bibliography files, and so on, leading up to final files in PDF or other formats.

Each document and the directory that contains its files is characterized by a Leonardo entity, and it is recommended to use composite entities for this.

The entity formant `doc:` is defined for this purpose, but the user may add other similar operators and define them according to his or her preferences for how to organize the collection of documents. Each document entity specifies the location of the document's directory. For example, the entity

```
(doc: caisor 2010 14)
```

specifies a directory that is located at

```
Series/caisor/2010/pm-014/
```

where the `Series` directory is on the same level as the individual where the Facility is being performed. The access to the document directory is done by relative addressing, so as seen from the current directory of the Leonardo session the access path is therefore

```
../../../../Series/caisor/2010/pm-014/
```

The function `doc:` is defined in the entityfile `doc-onto` which is located in the agent's `Ontology` directory. Alternative functions for designating document directories can be introduced by analogy with it.

Each document directory must at least contain a file whose name is `def.leo` and that is an entityfile containing basic information about the document. For simple documents it will only contain one entity, namely itself, and its contents may include for example

```
-------------------------------------------------------------
-- (doc: caisor 2010 14)

[: type docco]
[: contents <(doc: caisor 2010 14)>]
[: title "Leonardo Document Preparation Facility"]
[: docname "doc-prep-facil"]
[: nrversions 1]
[: year 2010]
[: date-completed "2010-05-24"]
[: date-posted "2010-05-31"]
-------------------------------------------------------------
```

There may be additional entities in this entityfile in some cases, for example if the document contains diagrams that can be described, or even fully defined through Leonardo entities.

A *result file* in a document directory is a file containing the finished document, typically in PDF format. Usually there is only one result file in a document directory, and its name is specified by the `docname` attribute, for example `doc-prep-facil.pdf` in the case above. If that attribute does not have a value then the default is `paper.pdf` .

In a document with a simple structure there will also be a file called `body.tsl` containing source text, and three `.tex` files namely:

- `body.tex` for a translation of the contents of `body.tsl`
- `paper.tex` (or whatever is the name of the result file) for the top-level latex file
- `descrip.tex` for a file containing the title of the article, the name(s) of the author(s), and other information for the front page.

The latter two files are normally short and have stereotypical contents. The latex processor is invoked with `paper.tex` or its counterpart as the argument. This file does include operations (i.e. the Latex `input` command) on the other two. Examples of these files are shown in the appendix.

This structure makes it possible to separate the main contents of the document from the cover and other "front matter." It leaves a lot of flexibility for the user to configure the structure as he or she wishes. For example, if one produces a number of articles with similar layout on the front page, such as for departmental reports, then one can use identical `paper.tex` files for all those documents and let them refer to a common Latex style file that is located in a place that is available by relative addressing, such as at

```
../../../../Series/caisor/styles/latex/
```

in the example above.

1.2 Auxiliary Software

In order to use the Facility one needs to define four auxiliary softwares to it, namely a Latex processor such as Miktex, a pdf viewer, a text editor and a web browser. Their locations must be specified using entity descriptions like the following ones:

```
------------------------------------------------------------
-- notepad

[: type os-command]
[: commandphrase "C:\WINDOWS\system32\notepad.exe "]

------------------------------------------------------------
-- acrobat

[: type os-command]
[: commandphrase "C:\Progra~1\Adobe\Reader 9.0\Reader\AcroRd32.exe "]

------------------------------------------------------------
-- iexplorer

[: type os-command]
[: commandphrase "C:\Progra~1\Internet Explorer\IEXPLORE.EXE "]

------------------------------------------------------------
-- bibtex

[: type os-command]
[: commandphrase "C:\Progra~1\MiKTeX 2.7\miktex\bin\bibtex.exe "]

------------------------------------------------------------
-- dvipdf

[: type os-command]
[: commandphrase "C:\Progra~1\MiKTeX 2.7\miktex\bin\dvipdfm.exe "]
```

```
------------------------------------------------------------
-- latex

[: type os-command]
[: commandphrase "C:\Progra~1\MiKTeX 2.7\miktex\bin\latex.exe "]

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

It is recommended to locate these entity definitions in an entityfile in the `Leohost` directory since they are specific to the host where the session is run, especially if one wishes to use the same agent alternatingly on different hosts. The final space in each of these strings is necessary since the system concatenates these strings with a string containing the file that is to be processed.

1.3 Finding a Document

It is a matter of taste whether one wishes to name documents by number, like in the examples above, or by a mnemonic code. In any case it is inconvenient to let the full title of the document be the directory name, since one will then have to type it in again each time one is going to work with the document. The `doc:` function is defined in such a way that if its last argument is a number then it is converted to an entity of the form `pm-nnn` otherwise it is retained as it is.

If one has many documents to work with then it may be a challenge to recall what was the number of a particular document one wants to access. This may be resolved using the aforementioned GUI, or using a command that lists all the documents by name/number and full title.

Chapter 2

Directory Structure for Static Webpages

The directory structure for the authoring of webpages is analogous to the one for documents. We are concerned here about the structure that is used during authoring; transfer of the checked-out pages to a public server is a separate question.

Since the word "webpage" is used in a fairly broad way we use the term *webnote* for a single, static page. Each webnote obtains its own directory, like for documents.

The entity former `wen:` is used like in the following example:

```
(wen: krf framework intro)
```

and represents a webnote that is located at

```
Sites/krf/framework/intro/
```

where the `Sites` directory is a sideways neighbor to `Series` and to the invoking Leonardo individual. Like for the function `doc:` the user should define his own, similar function if other directory structures are desired.

A typical webnote directory will contain a source file called `body.tsl` and its translation called `body.html`, a defining entityfile called `def.leo`, and usually also a number of additional files pertaining to the webnote. The choice and form of those pages depends strongly on how the user wishes to organize the use of *styles* in the webpage sense. One possibility is to make use of the standard (CSS) stylesheet facility; another possibility is to obtain the same effect using methods that are built into Leonardo. A third possibility is of course to do neither.

2.1 No Use of a Style Mechanism

Add text here.

2.2 Using CSS

Add text here.

2.3 Leonardo Webnote Pattern

The Leonardo webnote pattern is a way of organizing a collection of webnotes under a common left-side menu, and with some uniformly used facilities, such as the use of timestamps and the preservation of webnote history. The reader should first check a website that is organized in this way, e.g. at

`http://www.ida.liu.se/ext/caisor/`

Please check in particular the *Version history* link at the bottom of the page.

Webnotes that adopt this pattern use the following organization. The main current page is always called `page.html`, for example, as seen from a browser,

`http://www.ida.liu.se/ext/krf/framework/intro/page.html`

In addition, each recorded generation of the webnote has a name containing its timestamp, for example

`http://www.ida.liu.se/ext/krf/framework/intro/page-100523.html`

for the page posted on `2010-05-23`. The page called `page.html` may contain a copy of the latest timestamped page, or a forwarding link to it. Finally there is a page called `versions.html` containing the versions list that one sees when clicking the *Version history* link.

Each successive `page-`*`date`*`.html` file is generated from the current `body.html` file by wrapping it with title lines and a bottom line. The information for these are obtained from the webnote's `def.leo` file.

According to this pattern, the choice of style may occur both in the translation step from `body.tsl` to `body.html`, and in the wrapping step from `body.html` to the timestamped `page` file. The method for defining the style will be described in the chapter on webscripts. It follows that only the current version of a webnote can be restyled, whereas older versions will retain their original style. This may be considered as an advantage or as a disadvantage, or both.

Chapter 3

Facility Command Verbs

As already described, the main contents of a document or a webnote is usually in a file called `body.tsl.` The main development step is then to text-edit `body.tsl` and to invoke an appropriate command whereby the revised file is processed, obtaining and displaying the corresponding PDF file, for a document, or an HTML file, for a webnote. A few special operations are needed when the document or webnote is created initially. The present chapter describes these initial and cyclic operations.

In this chapter we shall specify session commands without the surrounding square brackets.

3.1 Creating a Document or Webnote

In order to create a document one must first perform the following operations manually, i.e. without the help of command verbs in the Facility:

- Create a directory for the document, and a subdirectory for it called `cl`
- Create its `paper.tex` and `descrip.tex` files
- Create an initial `body.tsl` file (possibly empty)

Similarly, in order to create a webnote one must first perform the following operations manually:

- Create a directory for the document, and a subdirectory for it called `cl`
- Create an initial `body.tsl` file (possibly empty)

After this, in each of the cases, one issues a `Rev` command with the appropriate `doc:` or `wen:` expression as its argument, followed by a few additonal commands as in the following examples. The `Rev` command defines its argument as the 'current' document or webnote, so that other commands in the family are given without argument; they apply to the current document. The initial command sequence for a document is therefore like in the following example.

```
Rev (doc: caisor 2010 14)
doci
```

The `doci` command does document initialization. Similarly, in the case of a webnote:

```
Rev (wen: krf framework intro)
weni
wco
```

3.2 Editing a Document

The following commands are useful in the editing cycle.

```
cocl body      Convert body.tsl to body.tex
lam            Execute latex on top-level latex file
bim            Execute bibtex on top-level latex file
dpdf           Convert result of lam command to pdf
acro           Display current result (pdf) file
clad           The previous four commands in succession
rco            Load def.leo file for current document
wco            Write def.leo file for current document
```

The argument for `cocl` can of course be chosen freely, so that a document can use several source files which are composed using inclusion commands in the `paper.tex` file, or so that TSL source files include each other.

3.3 Editing a Webnote

The following commands are useful in the editing cycle.

```
coch body      Convert body.tsl to body.html
rco            Load def.leo file for current document
wco            Write def.leo file for current document
dispwen        Display page using current web browser
```

Notice that it is not obligatory to use the TSL in the `body.tsl` file; one may also edit the `body.html` file directly and refrain from using the `coch` command.

The following additional commands apply for the Leonardo webnote pattern.

```
ghd            Generate page-date file by wrapping body.html
ghp            Generate page.html with a forwarding link
ghv            Generate versions.html
ghal           Execute [rco] followed by the previous three
cogh           Execute [coch body] followed by [ghal]
```

3.4 Additional Operations

Errors during the translation process should be trapped by the system, but alas sometimes they are not. In such cases one should execute `(closo)` ('close source and output') in order to close input and output files to the operation in question.

The following operations invoke the selected text editor, such as Notepad in the example above.

```
ecd             Text-edit and reload current def.leo file
ebt             Text-edit current body.tsl file
ebh             Text-edit current body.html file
```

Note: The commands that invoke the text editor, the browser, and the pdf viewer are presently defined so that they directly use the software names shown above (notepad, iexplorer, acrobat). This should of course be changed so that user preferences are made explicit.

3.5 Document Index Pages

The `Sites` directory contains the material for one or more static websites, and additional operations are needed for exporting content to operating servers. It is also natural to post documents on such servers when they are ready for being made public, and as a first step they should be copied from the `Series` directory (according to the example directory naming that was used here) to their appropriate place in the `Sites` directory.

Each document will then obtain its own subdirectory in the `Server` structure, just like each webnote. Such a directory will be called an *index directory,* or *document index directory* if there should be an ambiguity. Each index directory shall at least contain two pages: a page called `index.html` that is the natural entry page from a browser, and a PDF file for the current version of the document. The index page shall contain general information about the document plus a link to the full text.

Additional material in the index directory may include earlier versions of the article, errata pages, attached information that for some reason one does not wish to include in the PDF file, for example very large illustrations, and so forth. In all cases the index page shall be the natural starting point for viewing those other materials.

The following are the two major command verbs with respect to index pages:

```
glix            Creates index directory and index file
xpdf            Produces pdf file in index directory
```

These two commands depend on specific conventions for the location and structure of the index directories and, in the case of the `glix` command, scripts for the contents and appearance of the index pages. The present definitions of these two commands are therefore too specific and have to be hand-edited to fit the setup of each user.

To be precise, the present system relies on a few specific site names, such as `caisor-site, piex-site, labmop-site,` and so forth. Each site is associated with location information and with a webscript for index pages in

that site. To introduce an additional site name, one must define a webscript for it and introduce entries for it in a few case statements in the entityfile `formops,` in particular the function `ccdir` which defines the access path for index pages in the site in question, and in the operation `glix.`

Chapter 4

The Scripting Languages TSL and DSL

Besides for formatting documents and webnotes, the Facility also contains the means for defining *dynamic* webpages whose contents are computed for each request. Such requests are considered as commands in the Document Scripting Language, DSL, so that a webserver request like

```
http://myserver/plantselect?type=bush&flowercolor=yellow
```

will be processed within the system as a DSL command

```
[plantselect :type bush :flowercolor yellow]
```

This representation for DSL commands is the same as for commands in the Session Command Language, SCL, although the set of command verbs is of course different.

DSL command verbs are defined using scripts that specify the generation of HTML code. The Text Scripting Language, TSL, is used for webnotes (static webpages) and for documents, as described in the previous chapters, so it can also be translated to HTML. The difference between DSL and TSL can be illustrated by the following example of a simple DSL expression:

```
[fragment "Both an" [e "italic"] "and a" [b "boldface"]
      "word." ]
```

and the corresponding TSL expression:

```
[fragment Both an [e italic] and a [b boldface] word.]
```

Both expressions produce the text

Both an *italic* and a **boldface** word.

The difference between DSL and TSL is merely one of surface notation: they use the same set of command-words, and their internal representation in the system is the same. DSL uses the standard KRE parser, TSL uses a small (one page) separate parser that invokes the KRE parser for some purposes.

Dynamic webpages are usually defined in DSL since they tend to make heavy use of knowledgebase access. The source texts for documents and

webnotes are more conveniently written in TSL, in most cases, since this avoids the need for a lot of string quotes as illustrated by the examples.

4.1 TSL-oriented Verbs

The TSL parser reads a string in TSL format and converts it to an internal representation in terms of KR expressions. If it is used to parse the contents of a file then it reads and parses one paragraph at a time, where blank lines separate paragraphs.

While processing a string, the TSL parser primarily identifies occurrences of a left square bracket, and also occurrences of a few interpunction characters if they appear immediately after a whitespace character, namely the colon, the exclamation mark, and the hash sign. Everything else is considered as plain text.

As long as there is no recognized occurrence of the three special characters, left square brackets are expected to be matched by corresponding right square brackets; the first symbol after the left square bracket is considered as an operator, and the remaining contents is considered as strings and subexpressions as shown in the example above. The following operators are frequently used in TSL-coded texts, but notice that they can equally well be used in DSL expressions.

4.1.1 Document Structure

`[chapter ...], [section ...], [subsection ...], [subsubsection ...]`

Produce 'headings' on the levels suggested by these names. Defined for Latex; only the `subsection` command is also defined for HTML where it is mapped to heading level 2.

`[heading ... :level n :fontsize sz]`

Produces a heading of the level indicated by the positive integer `n` where 1 is the highest, like in HTML. Defined for both HTML and Latex; in the latter level 1 is mapped to `section`, 2 to `subsection`, etc. The value of the optional `:fontsize` parameter shall be e.g. `"10pt"` and the default is `"11pt"`

`[itemize [item ...][item ...] ... [item ...]]`

Produces an itemized list; defined for both Latex and HTML. Usually each item begins a new line in the source text, but this is not obligatory. Line separation is free, but notice that there must not be a blank line inside an expression such as this one.

`[itemlist [item ...][item ...] ... [item ...]]`

Synonymous with the `itemize` command; HTML only.

`[numblist [item ...][item ...] ... [item ...]]`

Similar to `itemlist` but with numbered items. Defined for HTML only.

`[footnote ...]`

Produces a footnote in Latex; also defined in HTML but just produces the contents of the arguments there, awaiting a better implementation.

`[brfoot ...]`

Produces a footnote in Latex [1] and wraps the footnote index in square brackets.

4.1.2 Inclusion

`[includefile file]`

The argument shall evaluate to a string containing a relative access path from the session directory to a file. Its contents are included as is into the file being produced. HTML only, but notice that `[input file]` has the same effect in the case of Latex.

`[include e p :source-xtn sx]`

The arguments shall be an entities and are not evaluated; the value of `(get e p)` is included in the file being produced. If `sx` is `tsl` then assume that this string is written in TSL format, and process it accordingly, otherwise use it as is. The second argument may be omitted and the default value for it is `Body.html`. This command is useful when a short text for inclusion is conveniently stored as the property of an entity. Only defined for HTML (definition for Latex exists but looks strange).

4.1.3 Local Text-Style Control

`[font ... :size sz :color co :fontsize fs]`

Produces the contents of the arguments, wrapped in the HTML `font` command defined by the optional `:size` and `:color` parameters. If the `:fontsize` parameter is present then it is mapped to a corresponding `font-size` inside `style` command. HTML only.

`[e ...], [b ...], [t ...]`

The expression(s) given as argument(s) are produced in emphasized, boldface, and 'typewriter' style, respectively. Both Latex and HTML.

`[style ... :bf t :em t]`

Combines `b` and `e` commands, and allows the use of evaluated parameters for determining whether to use the style in question or not. It is intended to add more parameters to this command. HTML only.

`[dq ...]`

The expression(s) given as argument(s) are produced but wrapped between double quotes. Latex only.

`[parze ...]`

The expression(s) given as argument(s) are produced but wrapped between round parentheses. Both Latex and HTML.

[1] This is an example of how it appears.

`[sqbr ...]`

The expression(s) given as argument(s) are produced but wrapped between square brackets. Both Latex and HTML.

`[$ ...]`

The expression(s) given as argument(s) are produced but wrapped between dollar signs (indicating a formula). Latex only.

`[kre ...]`

The argument shall be given as a KR expression and is not evaluated. It is presented with its correct appearance in the result document. Latex only.

`[tcolon s]`

The argument shall be a symbol; it is presented in the output in `t` font and preceded by a colon character. One can see many examples of the use of this command in Leonardo documentation, for example in the description of the `font` command above. Latex only.

There is an important rule for those commands having parameters, such as `style` and `heading:` the parameter may be given as an arbitrary CEL expression. This means that when the TSL parser encounters a tag, i.e. a symbol beginning with a colon, then it calls the DSL parser recursively for parsing the CEL expression immediately following the tag, gives the parsed expression to evaluation, and uses its value for the execution of the command. Arguments are not in general evaluated, on the other hand; it is up to the specific command whether it will evaluate and/or execute its arguments.

4.1.4 Verbatim presentation

`[verbatim s1 s2 ... sn]`

The arguments shall be strings; these are presented verbatim on successive lines. HTML only; can be used in both DSL and TSL.

```
[txt
  (several lines)
    :end txt]
```

The expressions `[txt` and `:end txt]` must be alone on their respective lines and must be written exactly as shown here. Exactly one space between `:end` and `txt],` no space before `[txt` on its line, and so forth. The lines in-between are presented verbatim in the result document. This construct is defined in both Latex and HTML, but can only be used in TSL. (In DSL the same effect is best achieved using an `include` command).

The choice of notation for this construct may merit some explanation. Recall that XML-style markup languages regularly use tagging constructs of the form

`<operator> ... ... </operator>`

We consider this to be unnecessarily clumsy in most situations, and for KR expressions we prefer more compact expressions of the form

```
[operator ... ...]
```

and similarly for other brackets and parentheses. However, there may be occasional situations where legibility is improved by having a balancing occurrence of the operator at the end of an expression. We therefore make the informal convention, for the case of KRE records, that one can write

```
[operator ... ... :end operator]
```

The `:end` tag is reserved for this purpose, and the convention is that it has no semantic contents or use, although of course one may consider using it for a check that expressions are correctly bracketed. In the general case this convention, and any operations on it are used within the general scheme for representing KR expressions as datastructures. This does not apply in the case of the `txt` operator since it has to be treated as a special case by the TSL parser, but it is useful for uniformity to use the same notation using the `:end` tag anyway.

4.1.5 Control Operations

```
[if cond fr1 fr2]
```

The first argument is a CEL expression; the second argument and the optional third argument are DSL or TSL expressions according to the context where the command occurs. The condition is evaluated using the same evaluator as for SCL and the second or third argument is then chosen accordingly. Both Latex and HTML.

```
[repeat a s ...]
```

The first argument shall be given as a symbol (not preceded by a point or a colon); the second argument shall be a set or sequence. The variable obtained from the first argument by prefixing it with a point is bound successively to each member of the second argument, and for each such binding all the following arguments are processed and the result sent to the result file. An example that uses this textual operator is shown at the end of this chapter. Defined in HTML. The translator for Latex has a definition for `repeat` that suffers from some restrictions, but also a definition for a verb `nrepeat` which is believed to be entirely compatible with HTML `repeat.`

```
[repeat a s ... :numbering n]
```

This is executed like the previous case, except that the variable `.n` is bound successively to 1, 2, etc in the successive cycles for the loop variable. HTML only.

4.2 Source Texts for Translation to Latex

In many situations it is useful to have source texts that can be translated correctly to both HTML and Latex, but there are also situations where one knows beforehand that only Latex will be used. In these cases it is natural to use the following additional possibilities.

4.2.1 Document Top-Level Commands

The following commands are used for the preamble of a document, for example, for generating the file `paper.tex` as described in Chapter 1.

```
[documentclass ... :flags {f1 f2 f3}]
```

This generates a Latex expression of the form

```
\documentclass[f1,f2,f3]{...}
```

where the number and choice of flags is of course open.

```
[document ...]
```

This generates a Latex expression of the form

```
\begin{document}
  ...
\end{document}
```

Recall that blank lines may not occur within this source expression, so this is only intended for short files, like the normal use of `paper.tex,` which contain `input` or `include` commands referring to larger files.

4.2.2 Latex-Specific Conventions

In translation to Latex there is a convention that undefined operators result in producing a Latex command with the same name, and with the given arguments in the standard form for arguments in Latex. This means that many Latex commands can be used directly in Latex-oriented TSL text, for example the `input` command. This option does not apply in translation to HTML.

Since the TSL parser is not sensitive to curly brackets, backslash characters or the dollar sign, it is possible to insert Latex code using these characters and it will go straight through to the Latex file being produced. However this will not work if the same source file is later on used for translation to HTML, of course.

4.2.3 Spacing, Interpunction and Newline Control

The issues of correct spacing, interpunction and newline control turn out to be tricky due to the approach of parsing DSL and TSL source to the same internal representation. There is an issue to which degree spaces are to be preserved between successive subexpressions, and the TSL and DSL variants are not entirely consistent in this respect. In particular, when TSL defined source is translated to Latex then space characters in the source are preserved to the greatest possible extent, and a source-side expressions like

```
[b half][e way]
```

will appear without any space between the two halves of the word, whereas if the souorce is DSL and/or if it is produced to HTML then a space does appear there. The processor for Latex output juggles various flags in attempts to get this right as often as possible, but without complete success. It is

probably best to clean this up a bit more before attempting to document it.

The following verbs, defined only for Latex, may be useful for the time being when dealing with this.

```
[spc ]
```

Enforces a space character.

```
[br ]
```

Enforces a newline.

```
[newline ]
:end txt
Enforces a newline.

[txt
[wrap ...]
```

Produces its arguments, wrapped inside curly brackets as needed by Latex. It will therefore be correct to write e.g.

```
\input[wrap .filename]
```

in order to generate a Latex `input` command where the argument is given as the value of a variable. The `wrap` command is clearly a makeshift device in the hacking category.

4.3 TSL Special Characters

The use of the colon special character in tags has already been described. Likewise, when the TSL parser recognizes an exclamation mark that occurs immediately after a whitespace, it expects that this exclamation mark is immediately followed by a KR expression and invokes the DSL parser for it. The expression obtained is evaluated and its value is expected to be on the internal form used by TSL and DSL, and it is used for producing output to the destination stream in the standard way.

Suppose for example that the following entity description has been loaded into the session where the formatting is performed:

```
------------------------------------------------------------
-- nono

[: type entity]
[: english [fragment "This is" [e "absolutely"] "forbidden"]]

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

and a TSL source text contains

```
[fragment Then the guard said [dq !(get nono english)]]
```

then it will be formatted to

Then the guard said “This is *absolutely* forbidden”

An occurrence of a hash sign immediately after a whitespace will likewise lead the TSL parser to expect that that character is followed by a KR expression. This case is used when one wishes to include the textual form of that KR expression in the produced document. For example, the TSL source text

```
The set #{red white blue} has three members
```

will produce Latex code that permits the curly brackets to appear correctly in the resulting PDF file. The hash character is only supported for Latex production.

4.4 Other TSL-Oriented Notation

It was already mentioned that the TSL parser reads one paragraph at a time from a source file, processes it, and then turns to the next paragraph. This is for reasons of both performance and debuggability: the processing time grows more than linearly with the size of the fragment being processed (in the present implementation), and with the chosen approach it is easy to see where the processing was interrupted if a difficult bug appears. However, the obvious disadvantage is that this gives problems in those situations where one will like a command to extend over several paragraphs.

There are two ways of solving this problem. One way is to prepare a source file where one 'paragraph' in the source contains one or more paragraph separators inside it, so that it produces several paragraphs in the resulting Latex or HTML file. The following command is such a separator.

```
[/ ]
```

Another possibility is to use the special `begin` and `end` commands in TSL, as in

```
[begin e]
This may be a long text consisting of
several paragraphs.
[end e]
```

The result of this is that the `e` command is extended from the begin command to the end command. However at present this is only defined for commands not having any arguments, as is the case for the `e` command.

4.5 DSL-Oriented Verbs

Most verbs in this section are mostly used in DSL scripts. Such scripts arise for two main purposes: for defining dynamic webpages, and for defining the "wrappers" around the bodies of webnotes. Webnote content is defined (usually) in a file called `body.tsl` which is translated to a file called `body.html` which in turn is given as an argument to a DSL script that provides styling, header and footer contents, and so forth for the finished webpage.

The commands in this section apply for HTML only unless the contrary is specified explicitly.

4.5.1 HTML Document Structure

`[webpage ...]`

Produces the commands given as arguments. Is intended to be used on the top level of a dynamic webpage or a webnote wrapper script.

`[doctype ... :dtd d]`

Intended for use as the first argument of the `webpage` command; produces

```
<!DOCTYPE ... "d">
```

where `d` is the value of the `:dtd` parameter.

`[pass-on :path p]`

Intended for specifying a webpage that forwards to another page. Produces

```
<META http-equiv="refresh" content="0; url=p">
<META http-equiv="expires" content="now">
```

`[html ... ]`

Produces an expression enclosed by `(html) ... (/html)` . – In this definition and the following ones, please read less-than and greater-than characters instead of the round parentheses. (Problem getting Latex to produce those angle brackets but will solve it eventually).

`[head ... ]`

Produces an expression enclosed by `(head) ... (/head)` .

`[title ... ]`

Produces an expression enclosed by `(title) ... (/title)` .

`[body ... :fonts <font1 font2 ...> :bgcolor color]`

Produces an expression enclosed by `(body) ... (/body)` containing the clause for `bgcolor` if present. The following other parameters may also be included: `style, text, link, vlink.` If the `fonts` parameter is included then a separate HTML `fonts` command is also generated. If the `:fontsize` parameter is included then a `font-size` inside `style` command is generated.

4.5.2 Text Structure and Styles

`[fragment ar1 ar2 ... :fonts <font1 ... fontk> :nospace t]`

This verb is defined for both Latex and HTML. It produces the processed argument sequence, wrapped by the fonts information if present. If the `:nospace` parameter is present with a non-null value then no whitespace will be generated between the elements. The verb `wseq` is defined similarly but with `:nospace` as the only parameter.

`[pfragment ar1 ar2 ... :fonts <font1 ... fontk>]`

Similar to the previous one, except that the parameters in the `pfragment` expression are imposed on each of the commands given as arguments. The verb `pseq` is defined similarly.

`[divstyle ... :tag1 v1 :tag2 v2 ...>`

Produces a corresponding HTML expression of the form

`<div style="tag1:v1;tag2:v2 ..."> ... </div>`

where the given arguments generate the `...` and the given parameters generate the style expression.

4.5.3 Special Document Components

`[pval :val x]`

The parameter value is produced to the output file without any formatting, simply using the Lisp function `princ`.

`[image :source s :width w :height h]`

Produces the corresponding HTML `img` expression for presenting an image given as `s`.

Additional verbs to be added in this category.

4.5.4 Links

`[link ar1 ar2 ... :path p :target targ]`

In HTML, it produces the formatted arguments as a clickable phrase directed at the position indicated by `p` and for display in the "target" field `targ`. It also allows optional parameters `:style` and `:OnMouseOver` which are mapped directly to the HTML counterpart. The value of the `:path` parameter may be either of the following:

- A string
- An entity representing a webnote, formed e.g. using the formant `wen:`
- A record representing a dynamic webpage.

The command is also defined for Latex but there it merely produces a footnote containing the value of the `path` parameter.

4.5.5 Tables

`[table ...]`

Produces a table consisting of lines and columns. The parameters for `:frame, :rules,` and `:border` are defined like for HTML.

`[row ...]`

In the scope of a `table` action, produces one line of boxes. No parameters.

`[box ... :width wi :align al]`

In the scope of a `table` action, produces one box in the current row. The `width` and `align` fields are defined like in HTML. Other accepted parameters are `:valign, :rowspan, :colspan, :height` and `:bgcolor.` Also, if present, the parameter `fontsize` maps to a `style font-size` item like for some other commands.

4.5.6 Requests

```
[request ... :to to :method me :target ta]
```

Produces an HTML `form` expression with the contents specified as arguments. The `to` parameter generates the HTML `action` field and should be an entity representing a webpage verb. The `method` and `target` parameters generate the corresponding fields in HTML. The method can be `get, put,` or `post` with the obvious meanings, but also `upload` which produces

```
method=post, enctype="multipart/form-data"
```

```
[pass par val]
```

Inside a `request` expression, a `pass` record specifies that the binding of `par` to `val` shall be transmitted to the destination action. This corresponds to an `input type=hidden` expression in HTML. No colon or point before the first argument.

```
[forward <par1 par2 ... park>]
```

Inside a `request` expression, it generates `pass` expressions for each one of the parameters `pari` , whereby their respective values in the sending environment are made available in the target environment.

```
[sendbutton xpr]
```

Inside a `request` expression, it generates a "submit" button that is labelled by the argument which should be a string.

```
[input text :tag ta :show sh]
[input textarea :tag ta :show sh]
```

Inside a `request` expression these commands generate `input` or `textarea` fields within which data can be entered and submitted to the request. The `:tag` parameter specifies the tag assigned to this input field in the request being transmitted. The `:show` parameter specifies the initial field contents before they have been changed by the user. Additional parameters are `:charsize` or (equivalently) `:charwidth` for the width of the field, and `:lines` for the number of lines in a textarea.

```
[field ...]
```

In the scope of a `table` action, this means `[box [input text ...] ]` . It allows the parameters `:tag` and `:charsize` and not others.

```
[radiolist tag {[: v1 s1][: v2 s2] ... }]
```

This generates the HTML for a vertically aligned suite of radio buttons, i.e. buttons where only one can be selected, where each button is labelled by one of the strings `si` and selecting this button has the effect that the parameter given as the first argument, `tag` above, is bound to the corresponding symbol `vi.` For example, the following command

```
[radiolist theme {[: health "Health"][transports "Transports"]}]
```

produces the following HTML code

```
<input type="radio" name="theme" value="health">Health<br>
<input type="radio" name="theme" value="transports">Transports<br>
```

4.5.7 Frames

```
[frameset cols ...]
[frameset rows ...]
```

Generates a `frameset` expression in HTML. The first argument specifies whether the component frames are to occur columnwise or rowwise. The remaining arguments should be `frame` records or subordinate `frameset` records. The `border` and `frameborder` tags are used like in HTML. The `size` tag is used for the same purpose as in a `frame` action record which is defined next.

```
[frame ... :size sz :name na :path pa :scrolling sc]
```

Specifies one frame within the scope of a `frameset` expression. The value of the `size` parameter specifies the height or width of the frame at hand in the surrounding `frameset` expression. The `name` , `path` and `scrolling` parameters are used like in HTML.

Notice that a `frameset` command in HTML specifies the height or width of all its subordinate frames or framesets in one single list, but in the corresponding DSL command each component specifies its own height or width using its `size` parameter.

4.5.8 Interpreter Calls

```
[invoke v]
```

Invoke the argument command; HTML only. The evaluated argument can be either of the following:

- An entity representing a textual command: Executed without arguments or parameters
- A term: evaluated and its value is produced to output as described in Section 4.6
- A command (represented as a record as usual): Execute it.

```
[exec c]
```

The evaluated argument shall be a DSL command which is then executed. The command verb must be defined using an `Actiondef` as defined in the next chapter. HTML only. This definition should be generalized.

4.6 Odd Arguments in DSL and TSL

A DSL expression shall in principle be a record where the arguments are again DSL expressions, strings or numbers. Strings and numbers are copied

directly to the result file. The following are the odd cases aside from this main pattern.

4.6.1 Symbols

There are a few symbols that may occur as arguments in themselves in DSL and that have special functions. The `/` symbol enforces a new line, like the HTML command `br`. Similarly, the `//` symbol enforces a new paragraph, and the `--` symbol enforces a horizontal line. The first two are defined both for Latex and HTML, the last one only for HTML.

The same symbols are available as operators in TSL where they are expressed as `[/]` `[//]` and `[--]` respectively. (At present a space is required before the right square bracket due to an unfixed bug). Only the first one is defined for Latex, all three for HTML generation.

Additional uses of symbols are in preparation.

4.6.2 Forms

Terms that occur as arguments where a string or a formatting command is expected are evaluated, and the value is printed as a KR expression in the HTML case, and given again to the formatter in the Latex case. (This discrepancy should be reviewed).

4.7 Tacit Commands

Tacit commands are those that do not produce any output to the result file, but which are included in a source document or a script for other reasons. The following commands are only defined for HTML generation.

```
[comment ... ]
```

This command has no side-effects either.

```
[load :ef ef]
```

Loads the entityfile given as an entity in the parameter.

```
[tracemess :msg "..."]
```

Prints the value of the `:msg` parameter on the agent console, which at the same time is the server console.

```
[trace x1 x2 ...]
```

Analogous to the `fragment` command, but the HTML being produced is sent to the agent console and not to the result file.

```
[traceparams ]
```

Prints the arguments and the parameters of the nearest enclosing command on the agent console.

```
[pstack "..."]
```

Prints the optional argument string on the agent console and then prints the contents of the stack of textual command parameters on that console.

Chapter 5

DSL/TSL Verb Definitions

The previous chapter defined the vocabulary of elementary verbs for DSL and TSL. Additional verbs can be defined in terms of those, or directly using additional Lisp code. This is used for two purposes: in order to use the new verbs as 'procedures' or 'macros' in the source files for documents and webnotes, and in order to define dynamic webpage commands like `plantselect` that was mentioned as an example in the beginning of the previous chapter. The present chapter defines how such definitions are expressed in Leonardo entityfiles.

When studying these definitions, please recall that parameter values are always evaluated before a verb is invoked, whereas the treatment of arguments is specific to each command. Most command use the arguments for DSL or TSL subexpressions that are to be formatted and produced to the output file, but there are exceptions to this.

5.1 Simple Textual Verb Definitions

Verbs for use in DSL and TSL are called *textual verbs*. The following is a simple example of the definition of such a verb.

```
-------------------------------------------------------------
-- plantpage

[: type webserver-verb]

@Textdef
[body :bgcolor "#eeffdd" ^
    [heading .p :level 2]
    "Latin name: "
    [if [hasvalue (get .p latin-name)]
        (get .p latin-name)
        "(Not available)" ] //
    [link [b "Picture"] "<br>" :style "text-decoration:none;"
               :path (get .p picture-link) ] ]

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

This defines a verb for a dynamic webpage that can be invoked using the following DSL command

```
[plantpage :p Dandelion]
```

and using the following HTML request to `localhost`

```
http://localhost/plantpage?p=Dandelion
```

The use of the double-slash symbol is an example of a single symbol in DSL which was defined in the last section of the previous chapter.

If there is repeated use of similar expressions, such as might easily be the case with the conditional expression in this example, then one may encapsulate them as a 'macro', for example as follows.

```
-------------------------------------------------------------
-- condattr

[: type leoverb]

@Textdef
[fragment
    [if [hasvalue (get .p .a)]
        (get .p .a)
        "(Not available)" ] // ]

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

and rewrite the definition for `plantpage` more compactly as

```
-------------------------------------------------------------
-- plantpage

[: type webserver-verb]

@Textdef
[body :bgcolor "#eeffdd" ^
    [heading .p :level 2]
    "Latin name: " [condattr :p .p :a latin-name]
    [link [b "Picture"] "<br>" :style "text-decoration:none;"
               :path (get .p picture-link) ] ]

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

However, the following two considerations must not be forgotten. First, the entityfile containing one or more definitions of textual verbs must have the entity `verbfile` in its `in-categories` attribute, for example as follows.

```
-------------------------------------------------------------
-- plantsite

[: type entityfile]
[: in-categories {verbfile}]

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

The effect of this is that at the end of loading the entityfile, a scan is made of all the entity definitions in it, and occurrences of the `Textdef` property

are converted from text form to datastructure form under the Lisp property `textdef.` Secondly, any verb that is going to be used to define a dynamic webpage, such as `plantpage` in the example, must also have a `for-domain` attribute which specifies which of several virtual servers [1] in the session is going to publish the verb. This attribute as well as other related attributes will be described in the section on webserver definition.

If one wishes to define, as a textual verb, a symbol that is already defined for another purpose in the Leonardo agent at hand, then one must choose another name for the entity and specify the desired name using the attribute `published-as,` for example

```
-------------------------------------------------------------
-- condattr-verb

[: type leoverb]
[: published-as condattr]

@Textdef
[fragment
    [if [hasvalue (get .p .a)]
        (get .p .a)
        "(Not available)" ] // ]

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

5.2 The Passing of Parameters

Textual verbs are normally defined and used with parameters and not with arguments, as shown in the above examples. The predefined elementary verbs are obvious exceptions, and there are also ways of defining additional verbs in this way (documentation later), but the use of parameters is preferred.

The second definition of `plantpage` showed how an incoming parameter `p` to the definition was passed on to a subordinate verb. This is a common situation and sometimes there is a list of several parameters that have to be passed in the same way. The particular tag `:forward` can be used for convenience in such cases, and the above example can be rewritten as

```
-------------------------------------------------------------
-- plantpage

[: type webserver-verb]

@Textdef
[body :bgcolor "#eeffdd" ^
    [heading .p] //
    "Latin name: " [condattr :a latin-name :forward {p}]
    [link [b "Picture"] "<br>" :style "text-decoration:none;"
               :path (get .p picture-link) ] ]
```

[1] Terminology: I actually think virtual servers is something else in the ACL lingo. Must be checked.

The use of `forward` is of course pointless when it only forwards one parameter, but the forwarded set may contain any number of parameter symbols. Notice that they are written *without* preceding colon or point character.

5.3 Executable Procedures for Webpage Verbs

The verb `plantpage` was defined so that it merely displays information from the knowledgebase, but in many cases one wishes a dynamic verb to perform some computation and to have some effects on the knowledgebase in the session of the server. Such procedures can be defined using the properties `Actiondef` or `Performdef` for the webpage verb in question. An `Actiondef` property is written in Lisp; a `Performdef` property is written in the Session Command Language, i.e. using the same verbs as can be used in command-line input.

Almost all presently defined webpage verbs are written using `Actiondef` properties, and the use of `Performdef` is in a development stage. The following is an abbreviated example of an `Actiondef` definition for the verb where the user changes her password:

```
@Actiondef
(lambda (args params ai)
  (let* ((old-pw (cadr (assoc :old-pw params)))
         (new-pw (cadr (assoc :new-pw params)))
         (sesid (cadr (assoc :sesid params)))
         (service-id (--- omitted ---))
         (user (get (intern sesid) 'with-user)) )
        (cond ((not (member user (cdadr
                       (get (get service-id 'user-register)
                            'contents ))))
               (princ "Unknown user")(terpri)
               `(rec& fail\: nil ((:reason unknown-user))) )
              ;;  Here, check that the specified old password
              ;;  is correct, if not, return a fail: record
              ;;  like under the previous condition
              (t (setf (get user 'has-password) new-pw)
                 (write-ef (get service-id 'user-register))
                 `(rec& ok\: nil ((:sesid ,sesid))) )
                       )))
```

This definition obtains the argument list `args` and the parameter list `params` of the command in question, together with a representation of computational context as `ai`. It obtains the values for some of the incoming parameters from `params` and checks that the password change request is OK. If so it returns a record headed by the record former `ok:` otherwise it returns a record headed by the record former `fail:`. The record returned is given to the `Textdef` property, so the parameters in the returned record may be used as variables there, in order to determine the choice and the contents of the resulting output to the browser.

Additional details about the use of `Actiondef` definitions are intended to

follow in the report on Lisp programming in Leonardo.

5.4 The request Operator

Many uses of dynamic webpages are organized in such a way that the user is first presented with a *specification page* where she or he enters parameters for a request, for example by completing input fields or by choosing radio buttons, and where these parameters are then sent to an *effect page* for computation and for presentation of the results. Such a tandem of two webpages can be represented as a single entity in DSL. The specification page is defined using a property called `Specifydef,` the computation is defined using an `Actiondef` or `Performdef` property as just described, and the result of the computation is defined using a `Textdef` property.

Suppose we would like to define `plantpage` in this way, so that there is a separate page where the user types in her choice of plant to be presented. The following definition could be used.

```
@Specifydef
[body :bgcolor "#eeffdd" ^
   [heading "Enter choice of plant" :level 2]
   [request :to plantpage ^
      [input :show "" :tag p :charsize 50]
      [sendbutton "Enter"] ]]
```

This definition represents a displayed page containing a heading, a field of size 50 for data input initially containing blank space, and a button labelled `Enter.` This specification page can be invoked on `localhost` using

```
http://localhost/plantpage-specify
```

and a link to it may be included in the `Textdef` or `Specifydef` definition of another entity using the DSL command

```
[link "Link to plantpage" :path [plantpage-specify]]
```

Specification pages may use parameters whose values are provided to them in the obvious way using either of the two ways of invoking them, for example as follows, if the text of the heading is to be specified in the invocation,

```
[link "Link to plantpage"
      :path [plantpage-specify :heading "Enter choice of plant"]]
```

The `request` expression may forward parameter values (corresponding to the `hidden` operator in HTML) using either of the following kinds of commands

```
[pass p .pp]
[forward {p q r}]
```

The former command passes the value that it receives as `.pp` to the request page but with the tag `:p` whereas the latter command passes the values of all three parameters but using the same variables as they arrived with.

5.4.1 Using Composite Expressions as Parameters

The syntax described here makes it possible to pass arbitrare KR expressions as parameters, for example, sequences, sets, and records. This is unproblematic when textual verbs are used as 'macros' i.e. they are invoked in the definitions of other verbs, but it is less easy if the invocation is made via an HTTP request. In these cases it is necessary to encode the composite expression, or to make a request in `post` mode so that whitespace and special characters such as brackets can be accomodated. (Documentation of this should be forthcoming).

5.4.2 An Example

The following is a concluding example of DSL code written using some of these commands.

```
[request
   [pass e .e]
   [pass continue-at [entry-of-address :e .e]]
   [table [repeat a <firstname lastname email>
             [row [box .a]
                  [box [input :show (get .e .a)
                              :tag .a
                              :charsize 50 ]]]]]
   [sendbutton "Enter"]
   :to entry-of-email ]
```

This expression is used in an environment where the variable `.e` has a value; this value is presumed to be an entity. It displays a table of three lines, one for each of the three attributes `firstname, lastname` and `email.` On each line there is one box containing the attribute and one box initially containing the current value of the attribute in question for the entity `.e,` and the user is allowed to change that value. The updated values, together with the value of `.e` itself is sent to the command `entry-of-email.` Notice that some of the more intransparent HTML commands have been replaced by more natural ones, such as for `row` and `box` inside a table. The `pass` verb provides a value to the destination, corresponding to the `hidden` construct in HTML.

Notice also that this definition can be converted to a generic operation for entry of arbitrary attributes simply by changing the explicit list of three attributes to a parameter, for example `attrs.` In this case it could be invoked using a command such as

```
[input-attrs :e .e :attrs <firstname lastname email>]
```

whereby the request expression is instantiated using the current value of the variable `e` and the specific choice of the list of attributes. This is an example of the usefulness of having composite KR expressions as parameters. (However, realistically speaking the `:to` parameter should also be provided.)

Notice furthermore the second `pass` expression that forwards an entire DSL command to the receiving page. The assumption is that `entry-of-email` shall update the database as requested, display a confirmation message, and

then continue (after either a time delay or a click by the user on an OK button) to the parameterized page specified by the value of its `:continue-at` parameter. The `pass` verb evaluates its arguments but does not execute them, even if they are commands; the execution takes place down the line using an expression of the form `[exec .continue-at]` This is an example of where the parameter value is a composite expression that may need to be encoded in order to be passed as an HTML request.

Chapter 6

Generation of Document Components from the Knowledgebase

One advantage of the close integration of the various languages and representational conventions in the Knowledge Representation Framework is that it facilitates cross-connections between different facilities. For example, there is a system documentation facility whereby entities representing command verbs and other system constructs can have properties containing the text for their documentation, and whereby documents can be defined to include such definitions for a list of entities. This is merely one example of such integration. The present chapter is dedicated to examples of such integration.

6.1 Documentation Generation

The conventions for systems documentation generation are as follows. Each entity concerned shall have an value for the property `Doc` and this value shall be expressed in TSL notation.

A document in TSL notation that wishes to include such properties may contain commands of the following forms

```
[docitems :list <v1 v2 ... vn>]
```

Generates the documentation for each of the verbs mentioned in the parameter. The documentation consists of a header line with the verb in question and its parameters, followed by the text in the `Doc` property of the entity. The list of parameters should be provided as the `args` attribute of each of the verbs.

```
[sysfunhead :op get :args <c a>]
```

Generates only the headline, in the same style as used by the `docitems` command. The text following the head is not added automatically and must be given in the source document. This command is used for functions, i.e. the head is enclosed by round parentheses.

```
[sysdochead :op loadf :args <ef>]
```

Similar but for verbs and predicates, i.e. the heading is enclosed by square brackets.

In a session where a document using this is being formatted, one must first perform the following operation for each source file `tslf` in TSL containing such constructs.

```
[sydload tslf]
```

The rest is automatic.

This facility has been used for some of the current Leonardo documentation, in particular for the KRF Overview document.

Chapter 7

Dynamic Resources and Hyperpages

Textual verbs that are to be invoked using HTML requests must be declared to be included in a particular *dynamic resource,* and they may optionally also be included in a particular *hyperpage.* The dynamic resource is needed because the Facility makes use of a feature in the Allegro CommonLisp webserver, namely, that a session with the system may operate several virtual servers each of which uses its own port. (There are also other uses of the dynamic resource construct, in particular for agent-to-agent message-passing).

By 'hyperpage' we mean an aggregation of some webnotes that are connected together using a common menue, and usually also a common style and appearance. The hyperpage construct is therefore used for defining webpage style: if a hyperpage entity defines a particular style then this style is adopted by all webnotes that are attached to that hyperpage. Moreover, hyperpage entities may be used for defining menu contents and other overriding structures.

The present chapter contains descriptions of these facilities, although some aspects of dynamic resources must instead be described in the separate memo on agent networks.

7.1 Dynamic Resources

Each Leonardo agent may *participate* in one or more dynamic resources. The available dynamic resources are defined in `indivmap-kb` in the entity-file `server-resources-catal.` One of the resources, and one which will often be present is `dres.coordin` which in principle is used for communicating system information between agents, i.e. for coordination purposes. However, it may also be used for trying out webpage commands, as long as one makes sure that the port it is using is not open to the Internet or a large local network. The following is the major part of its definition.

```
------------------------------------------------------------
-- dres.coordin

[: type server-resource]
```

```
[: defined-in indivmap-kb]
[: has-default-port 38082]
```

For each agent, the set of dynamic resources that it participates in is specified in the entityfile `agents-catal` in `indivmap-kb.` This is because this information is not only of interest for itself, but also for other agents. The following is a typical definition there.

```
-------------------------------------------------------------
-- agent.remus

[: type leo-agent-name]
[: leo-id lar-001-023-015-002]
[: in-individual indiv.L-reg]
[: has-ip-offset -10]
[: is-server-in {dres.coordin}]
-------------------------------------------------------------
```

By these definitions, the agent `remus` will be able to use a virtual server for `dres.coordin` on port 38072, obtained as the sum of the default port and the ip offset of the two entities. This arrangement is used since several agents may operate on the same host at the same time.

Most nontrivial sessions with a Leonardo agent require the command

```
[loadk indivmap-kb]
```

to be executed at the beginning of the session. This command will load most of the entityfiles in `indivmap-kb` but also several knowledgeblocks that are involved with webserver services. This startup will in particular observe whether `dres.coordin` is included in the `is-server-in` attribute of the current agent, and if so it will start a virtual server for that resource.

Other dynamic resources do not have their servers started automatically in the same way; this has to be requested by the user using the following command

```
[stare dres.anotherone]
```

but notice that this is only possible for dynamic resources that are included in the `is-server-in` attribute. The command

```
[resta]
```

lists all the dynamic resources in that attribute, with information whether a virtual server has been started for the resource or question, and if so what is its port number.

In order to define a textual verb for a particular server, the following must be observed. The verb must have the dynamic resource in question as the value of its `in-dyn-resource` attribute. The entityfile containing the definition of the verb must include the symbol `verbfile` in its `in-categories` attribute, as already mentioned above. Finally, that entityfile must be loaded *after* the virtual server has been started, using the `stare` command if applicable. However in the particular case of `dres.coordin` this is not an issue since its server is started as a part of the loading of `indivmap-kb.` Once the virtual server has been started, it is possible to re-load the entityfiles containing the textual verb definitions repeatedly, for example in order to update the definition of the verb.

7.2 Hyperpage Entities and Style Definitions

The following is a simple definition of a hyperpage entity.

```
-------------------------------------------------------------
-- cappa

[: type caisorsite]
[: has-style caisorstyle]
-------------------------------------------------------------
```

The type name is a bit obsolete and should be changed asap. This definition refers to a style definition which may be as follows.

```
-------------------------------------------------------------
-- caisorstyle

[: type webstyle]

@Styledefs
{[body :fonts <Verdana Arial Helvetica sans-serif> :fontsize 10pt]
 [heading :fontsize 10pt]
 [td :fontsize 10pt]
 [h2 :fontsize 12pt]
 [h3 :fontsize 10pt]
 [link :style "text-decoration:none; font-size:10pt;"]
 [box :fontsize 10pt]
 [font :fontsize 10pt] }

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

This style definition is a set of records each of which specifies parameters that are to be added to any occurrences of the DSL verb in question, in any webnote or textual verb that is attached to the hyperpage in question. For example, the entity `plantpage` that was used in the examples above could have the attribute assignment

```
[: intopic cappa]
```

Hm, have to check again how this is done, don't rely on it at present. Similarly, a webnote may specify its attachment to a hyperpage using the attribute `intopic` in its `def.leo` entityfile, for example

```
[: intopic cappa]
```

Hm, should change that attribute name as well?

7.3 The Leonardo Webnote Pattern and Website Entities

The major use of hyperpage entities is as a basis for specifying the various aspects of the Leonardo Webnote Pattern that was described in Section 2.3, including the use of a menue and an overall page structure that accomodates the menue. This requires a number of other attributes and properties.

Notice however that the use of these is not obligatory, and it is possible to use hyperpage entities merely as a holder for style if one wishes so.

The following is an example of a hyperpage definition for using the webnote pattern.

```
-------------------------------------------------------------
-- cappa

[: type caisorsite]
[: webtitle "CAPPA - Configuration of Publication Agents"]
[: pretitle "Configuration of Article Publication Agents"]
[: has-caption "Provided by the Experimental Electronic Press"]
[: has-style caisorstyle]
[: initpage "../cappa/cappa/intro/page.html"]
[: in-directory "../../../../Sites/cappa/"]
[: file-entities <gs-index gs-mainframe gs-head gs-meta
                    blank-bg head-bg white-bg>]
[: initmenue menue1]
[: fontlist <Verdana Arial Helvetica sans-serif>]
[: bgcolor "#cceeaa"]
[: textcolor "#002244"]
[: linkcolor "#006699"]
[: vlink-color "#006699"]
[: headcolor "#aacc99"]

@Menue1
[webpage
    [sitedoctype]
    [html [body :fonts   (get .cursite fontlist)
                :bgcolor (get .cursite bgcolor)
                :text    (get .cursite textcolor)
                :link    (get .cursite linkcolor)
                :vlink   (get .cursite vlink-color)  ^
            [table
                [menuline :thick 3]
                [menuitem :path "../cappa/cappa/intro/page.html"
                          :caption "Introduction and Welcome"
                          :then vspace ]
                [menuline :thick 1]
                [menuitem :path "../cappa/cappa/access/page.html"
                          :caption "Access to CAPPA"
                          :then vspace ]
                [menuline :thick 3]
                      ] ]]]

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

The present implementation of hyperpages uses the `frame` construct in HTML, but the definitions have been made in such a way that they shall also support a more modern implementation using the `div` command. Anyway, in terms of the present implementation, the `in-directory` attribute of the hyperpage entity specifies the relative location of the main directory for this hyperpage. This directory will contain HTML files called `index, mainframe, head,` and so forth as defined by the `file-entities` attribute

and the published-as attributes of its members. For example, there is a definition of the gs-index entity whose published-as attribute has the value 'index' *Have to arrange to obtain a plain double quote here, damn Latex.* The same directory will also contain an HTML file called menue1 whose contents are defined by the Menue1 property. The value of the initmenue attribute specifies that this is the first menue that is presented when the hyperpage is visited. Choices of clicks may replace it by other menues whose contents are defined by other, similar properties.

The last six attributes in the entity description are used for the generation of menue1.html as well as for all the files listed in the file-entities attribute, and with the obvious meanings. The earlier attributes are used as follows.

webtitle - Appears in the HEAD property of the file index.html , i.e. the main file for the structure of frames.

pretitle - Appears as a level-3 heading on each webnote in the hyperpage. It is followed, in each webnote, by a level-2 heading that is specific to that webnote.

has-caption - Appears in a separate, small frame field at the top of the frame layout.

init-page - The webnote that appears in the main field of the frame layout when the hyperpage is first entered.

In order to generate the required files one first selects the hyperpage in question using the Rev command, for example

```
Rev cappa
```

and one makes sure that the hyperpage definition has been loaded into the session. Then the following commands are used.

```
initsite
```

Creates the directory for the hyperpage and makes basic checks.

```
gsite
```

Generates the HTML files that are defined by the current hyperpage entity, including both the menue and the files listed in the file-entities attribute.

```
gsitot
```

Performs the [gsite] command, but in addition it also loops over the members of the value of the has-pages command, which should be a set or sequence of entities, and executes the [ghal] command for each of them. This means that it re-generates all the webnotes that belong to the hyperpage, provided that they are listed in the has-pages attribute. This operation is useful after a style change.

```
gmen menue2
```

Used if a property Menue2 has been defined as a menue, and if so generates the page menue2.html with the contents defined by it.

```
gme
```

Performs `[gmen menue1]` and is useful for regenerating the menue after its definition has been modified.

The definitions of these commands is somewhat shaky at present, and there has been problems with the menue generation command in particular.

Chapter 8

Considerations on Style

Some semi-philosophical considerations on webpage style and stylesheets may be in place here.

8.1 The Power of Procedures

The importance of using procedures does not have to be explained to any reader of this report. The examples above have shown that it is straightforward to organize a DSL package in terms of procedures which can be small, precise and generic in ways that does not have a counterpart in HTML or in XML.

Consider, for example, the very common situation where a website is organized as a number of pages each of which consists of menue columns to the left and to the right, a heading field at the top of the page, and a body of informatiion between these. Since the heading and the menues stay the same, one wishes to organize the definitions so that each verb only specifies the contents of the body in the middle. The natural way to do this in a procedure-oriented system such as DSL is to introduce one verb for the overall page structure and to let it have a parameter specifying the body contents. The definition of such a verb may follow the following pattern.

```
-------------------------------------------------------
-- pagetop

[: type leoverb]

@Textdef
[html [body
    [divstyle [myhead] ...]
    [divstyle [myleftmenue] ...]
    [divstyle [myrightmenue] ...]
    [divstyle [invoke .body] ...] ]]
```

Recall that `divstyle` generates an HTML command of the form `<div style="..."> ... </div>` where the parameters of `divstyle` specify the contents between the double quotes. The `divstyle` commands in the definition of `pagetop` specify the location and the appearance of the respective

divisions.

Given a definition such as this, each link from one page to another in this website can be written on the form

```
[link "otherverb"  :path [pagetop :body [otherverb ...]]
      :style ... ]
```

This will produce a clickable word "otherverb" with the style specified by the `style` parameter, leading to a page with the selected top-level structure and where the middle-part contents are specified by `[otherverb ...]` One may be unwilling to repeat this link expression for every link, especially if the `style` parameter consists of many parts, and in such a case it is natural to introduce an additional verb for it, such as

```
-----------------------------------------------------
-- mylink

[: type leoverb]

@Textdef
    [link :path [pagetop :body .dest]
          :text .text
          :style ... ]
```

Then each link can be written simply as

```
   [mylink :text "otherverb" :dest [otherverb ...]]
```

In other words, the brief definitions for the two verbs `pagetop` and `mylink` are sufficient for encapsulating the definition of the layout as well as the style information for both divisions and links.

8.2 Procedures versus Style Sheets

The manner of working that has been described here is natural if one is used to programming with procedures and modules. It differs from the use of Cascading Style Sheets (CSS) which is currently the standard method for defining layout and styling. The approach to the definitions of style when CSS is used is to first define webpages using standard HTML commands such as `body` , `div` and `a` , and then to impose additional parameters on the occurrences of these commands using style definitions.

From a programming-language point of view, the CSS approach is therefore an example of *advising* (ref. Teitelman 1969?) where a simple base program is decorated with additional details using separately defined amendments. Introducing an additional language level is always a nuisance, but one must also ask whether the advising approach of CSS provides additional advantages that are not obtained using the procedural approach.

One argument in the context of the present DSL implementation may be that it results in a larger expression being transferred from server to browser, since for example all style information has to be added to all places where it is used (in every link expression, for example), whereas using CSS the advising operation can be done in the client. This is however a temporary obstacle since it ceases to apply if the web client can execute DSL definitions

such as the definition of `mylink` above, and since this can presumably be emulated using Javascript code in a shorter perspective.

Another argument may be with respect to the "cascading" aspect of CSS, which means that advise to a given command may be obtained from several different sources, including both sources in the server and in the client. In principle this means that each of those sources, including the user of the web browser can specify aspects of style by advising universally defined commands, such as `body` or `h2.` Commands that are defined in the procedural approach, such as `mylink` do not offer this possibility since the style information is written into the verb definition. One can of course consider adding advising to the procedural scheme, so that for example a style advise to the `link` verb in DSL would have higher priority than the style rules that are given in the definition of `mylink,` but then we are back to having a more complex language structure. This is a significant aspect, and we are not yet ready to make a proposal for whether and how to provide multi-source style information in the Document Scripting Language.

This is not to say however that the use the Facility relies entirely on the procedural or 'macro' approach. The style concept for the Leonardo webnote pattern was described in the previous chapter.