Leonardo

Leonardo Project
Division for Artificial Intelligence and Integrated Computer Systems (AIICS)
Department of Computer and Information Science, Linköping University

Erik Sandewall

Leonardo Installation and Startup

This series contains technical reports from the Leonardo project, for the development of a software infrastructure for knowledge-based systems.

The present report, PM-leonardo-012, can persistently be accessed as follows:

Project Memo URL:	http://http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/012/
AIP (Article Index Page):	http://aip.name/se/Sandewall.Erik.-/2010/005/
Date of manuscript:	2011-03-04

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

Leonardowebsite:	http://http://www.ida.liu.se/ext/leonardo/
AIP naming scheme:	http://aip.name/info/
The author:	http://www.ida.liu.se/∼erisa/

The Leonardo system is a software platform for knowledgebased systems, that is, systems that make active use of structured information about some aspect of the world. The present report describes how to install a Leonardo system in a computer running a Windows, Linux or Mac operating system, up to the point where you have entered its command-line dialog. The report consists of the following chapters.

- Obtaining an Allegro CommonLisp system (if you do not have it on your computer system already).
- Installing a basic Leonardo system.
- Starting and using the dialog in a command-line session.
- Registration of simple 'consumer' agents and their users. This is not required for simple use, but it is relevant for obtaining system updates.
- Registration of more advanced 'interactor' agents. This is required for making use of message-sending between Leonardo agents.
- Special instructions for installing the Leonardo Student System.

Other reports describe other aspects of the Leonardo system. In particular, the report "Knowledge Representation Framework: Overview of Languages and Mechanisms" should be read before the present report. It will be referred to as the *KRF Overview* in the text that follows.

The report "*Initial Leonardo Exercises*" describes some things that one may wish to try out as a beginner with the system.

Chapter 1

Obtaining a CommonLisp Implementation

Leonardo is implemented in CommonLisp, so you need a CommonLisp implementation in order to run it. Several implementations of CommonLisp exist. The actual implementation of Leonardo was done using Allegro CommonLisp. In principle it should be possible to run it on any other correct implementation of CommonLisp as well, but there are some minor technical problems so that this does not work at present. (We hope to be able to correct these problems soon). We shall therefore first describe how to obtain an Allegro CommonLisp (ACL) system, and then briefly discuss the alternatives.

1.1 Initial Considerations

1.1.1 Choice between Server and Personal System

Your first choice is whether you wish to run Leonardo on a server or on a personal computer running a Windows, Linux or Mac operating system. Users at IDA (Linköping University Computer Science department) can use Leonardo on the department-wide Sun system. However, many will anyway wish to have everything installed on their desktops or laptops.

1.1.2 Choice between ANSI Base and Modern Base Lisp

When installing a Lisp system and a corresponding Leonardo agent, you should be aware of the distinction between ANSI Base and Modern Base Lisp systems. ANSI Base systems are designed in such a way that some aspects of their input are automatically converted to upper-case characters: you type lower case and you prepare your input files in lower case, but they are converted to upper case when read and appear in upper case when results come out again. Modern Base systems do not do this; they preserve both upper and lower case.

ANSI Base systems are therefore case-insensitive, to a large extent, and such was the historical practice in Lisp and it became entrenched through the ANSI Standard. ACL offers both possibilities, but some other implementations of Lisp offer only ANSI Base.

Leonardo has been implemented in a context of Modern Base but can be converted to using ANSI Base. However, the case-sensitive facilities of Modern Base are useful in many applications, so we recommend using it.

1.1.3 Implementations of CommonLisp

The Wikipedia page for CommonLisp [1] is a good source for general information about the language, and also for reference to major implementations. Another overview of implementations is found at [2] .

The primary ancestry tree for CommonLisp originates with MacLisp which was implemented at MIT Project Mac already in the 1960's and 1970's. After the standardization of Lisp dialects that resulted in CommonLisp it evolved into GNU Common Lisp (GCL) at MIT and into CMU Common Lisp (CMUCL). Notice however that GCL is not fully compliant with the standard.

The Steel Bank Common Lisp (SBCL) is a branch from CMUCL which claims to "be distinguished from CMU CL by a greater emphasis on maintainability."

Allegro Common Lisp (ACL) is a commercial system produced by Franz, Inc. This is the system where Leonardo has been developed.

Lispworks is another commercial system produced by LispWorks Ltd. in the U.K.

There is also a number of other implementations as described on the cited webpage. As it often happens there are minor differences between the implementations, in spite of the fairly detailed standardization, which means that Leonardo does not immediately run on all of these implementations. We are presently working on the modifications to the system that will be required for running it on CMU Common Lisp.

1.2 Allegro CommonLisp

The following alternatives are available if you are affiliated with our department (IDA):

- Run Leonardo on the department's computer network for research which uses SUN computers, and where Allegro CommonLisp is already installed.
- Obtain a copy of the ACL license from the TUS group and download the software from the Franz website shown above to your choice of computer (Linux, MacOS or Windows).

[1] http://en.wikipedia.org/wiki/Common_Lisp
[2] http://common-lisp.net/~dlw/LispSurvey.html

Other users may have similar options from their own institutions, and otherwise there are the following possibilities:

- Purchase a full ACL system (fairly expensive).
- Purchase a student system (if you are a student, moderately priced).
- Obtain the free 'Allegro Express' system from Franz (requires you to re-register from time to time)

For the download alternatives, the Franz website is located at

```
http://www.franz.com/
```

The following are the details for each of these alternatives.

1.2.1 Running on IDA's Research System

The executable for use with Leonardo is called `mlisp` and is located at `/sw/allegro-8.2/`

1.2.2 Running Standard ACL on a Desktop or Laptop

Download the system from the website shown above and follow the instructions as you proceed. Also, obtain a license file which is a small ASCII file called `devel.lic` and add it to the top level directory of the installed ACL system. You obtain it with your purchase; if you are affiliated with IDA you can obtain it from the TUS group or the Leonardo group.

The executable that is to be used for the purposes of Leonardo is found in the top-level directory and is called `mlisp` with the appropriate extension, for example `mlisp.exe` for Windows.

1.2.3 Using the Allegro Express System

On the Franz website, click "Downloads", then click "Allegro CL Free Express Edition", then "Download now ..." where you get to identify yourself, and then download according to your choice of operating system. The download is an executable file that you execute to install (Windows) or unzip as usual (Linux et al).

Before the Allegro Express system can be used for Leonardo it must be configured for this purpose. The system that you get from the download has two undesirable features: it is an ANSI Base system, and it comes with an attached editor that does not really fit into the Leonardo way of doing things. Fortunately, it is able to convert itself to a Modern Base system. Do as follows, once the system has installed itself.

On a Windows system: Position yourself where the system is located, which is usually at

```
C:\Program Files\acl81-express\
```

Click the file `allegro-express.exe` in this directory so as to start the run. This will open a group of two windows. Delete the editing window (the one to the right). In the window called the debugging window, type in or paste in the following expressions and hit the Return key after each of them:

```
(build-lisp-image "mlisp.dxl" :case-mode :case-sensitive-lower
                    :include-ide nil :restart-app-function nil
                    :restart-init-function nil)
```

and

```
(sys:copy-file "sys:allegro-express.exe" "sys:mlisp.exe")
```

Keep an eye on the contents of the `acl81-express` directory as you do this. The effect of the first operation should be the addition of a file called `mlisp.dxl` in that directory; the effect of the second operation should be the addition of a file `mlisp.exe` .

If the second operation fails and `mlisp.exe` does not show up, then it is also possible to simply drag-and-drop `allegro-express.exe` to a copy that you rename `mlisp.exe` .

After this, click the newly created file `mlisp.exe` . If everything has worked out right, this should open one single window that just shows a simple greeting. Check that it is correct by typing a single (lower-case) `t` into it. If it is correct then it shall answer you with `t` but if you have accidentally obtained an ANSI Base system then it will answer you with `T` instead. In this case, try the above procedure again.

On a Linux or Mac system: The same steps as for Windows apply.

1.3 CMU Common Lisp

The work on adapting the Leonardo system for use with CMUCL is in progress. We have identified the incompatibilities that will require adjustment of the existing code. [3] There does not seem to be any significant problem with these; it will just take some time to go over the code.

In addition there is a possibly non-trivial problem since ACL contains a built-in web server which Leonardo uses, both as a basis for web services and as the transport layer for message-passing. There have been some problems in getting this web server to work with CMUCL and it is not yet clear how this problem will be addressed.

[3] This work has been done by John Olsson.

Chapter 2

Installing a Basic Leonardo System

2.1 Characteristic Properties of Leonardo Systems

Unlike conventional software, Leonardo is organized in such a way that *programs and data are integrated.* In a conventional programming language you write a program, and then you apply that program to your data. In Leonardo you have instead one single structure that is called an *agent* that contains both programs and data. In fact, it also contains various other kinds of information that are intermediate between programs and data. Everything you do during a Leonardo session (once the system is up and running) uses and modifies the contents of that agent.

2.1.1 The Persistent Representation of Agents

The persistent representation of an agent is as a directory with its subdirectories and files in the computer's memory, or on a detachable memory device (DMD) such as a USB stick. This will be called the *agent directory.* A Leonardo agent can only execute if it is located in a *Leonardo individual* containing one or more, related agents and some auxiliary information for them. An *individual directory* is therefore a directory containing one or more agent directories as immediate subdirectories, plus an additional subdirectory called `Indivmap` that contains a number of catalogs with information about the computing environment and neighboring individuals.

Leonardo agents and individuals are *mobile* in the sense that they can be moved from one host to another and execute sessions on each of these. The internal information in the agent is preserved as the agent is moved. It can obtain information from each host that it visits, and accumulate it, and the agent can also leave information behind when it moves on. For example, an agent that is located on a USB stick can be moved between hosts simply by moving this memory device. It is also possible to move an agent in the sense of moving its entire file structure from one memory device to another, for example from a USB to a hard disk, or from the hard disk of one host

to the hard disk of another one. These possibilities are used very frequently in our own work.

There is also a stronger sense of "mobile agent" where the agent decides *by itself* to move from one host to another. This is useful in some applications and the Leonardo system design is prepared for this functionality, but it has not yet been needed in any of our own applications.

2.1.2 Configurations and Directory Names

It is the nature of things that there are different agents with different contents, including both procedural contents and 'data' contents, but there are also many situations where it is desirable to have several *exemplars* of the same *species* of agents, that is, several copies that differ in some of the details but whose overall structure is the same. For example, if an application is realized as a Leonardo individual containing one or more agents, then one may wish to have one exemplar for production use and another one for further development and testing. [1]

The following structure is used in order to organize the resulting multitude of agents. Each species of agents is given a particular name; `remus` and `leoregistrar` are examples of currently used species. Each individual has a particular *configuration* which specifies what species must or may be present in an individual with that configuration. For example, the configuration `Registrar` is specified to contain agents of the species `remus` and `leoregistrar.`

An individual can not contain more than one agent of the same species. Therefore, for communication within an individual it is sufficient to refer to agents using their respective species name.

For communication between individuals it is necessary to distinguish between different individuals with the same configuration. Therefore, each individual has a *directory name* which is formed by appending a serial number to the name of its configuration, obtaining for example the directory name `Registrar-2.` There is a server facility for registration of individuals, which is a mechanism for obtaining unique directory names within a certain context.

Agents have *proper names,* similar to the directory names of individuals. However, it is not possible to use the same serial number for the individual and its agents, since some species are used in several configurations. The assignment of proper names for agents is based on a particular aspect of the design, namely, that some species are used as *kernel agents* that provide the basic machinery for the individual, and each configuration of individuals designates one particular agent species that is to be used as kernels. Consequently, kernel agents are assigned proper names consisting of their species name plus a serial number that is assigned by the registrar, and other agents in the same individual obtain a proper name consisting of their species name followed by the serial number of the kernel agent in the same individual.

[1] In the English language the word 'exemplar' has a dual meaning, so it can refer both to one out of many copies of e.g. a book, and also to a prototype that copies are made from. We use the word in the former sense.

For example, since the individual `Registrar-2` consists of agents of the species `remus` and `leoregistrar,` if the former agent has the proper name `remus-43` then the latter one will have the proper name `leoregistrar-43.` One will then assume that there are other `remus` agents with numbers from 1 to 42, but there is no similar assumption for agents of the `leoregistrar` species.

As already mentioned, the persistent representation of a Leonardo individual is as a directory having immediate subdirectories for each of the agents in the individual. It is strongly recommended that the directory of the individual is named by the individual's directory name. It is also required that the directory of each of the included agents is named by the agent's species (not its proper name). In our example, the directory `Registrar-2` will have subdirectories `remus` and `leoregistrar.`

2.1.3 Agent Integrity and Clones

The standard file systems for computers make a distinction between *copying* and *moving* a file or a directory. After it has been copied it is still present in its old location, but after it has been moved this is not the case. The same distinction is made for Leonardo individuals, but in a stronger sense. If an individual, represented as a directory structure containing a number of files is *moved* then it shall retain its name, but if it is *copied* then it needs to register in order to obtain a new directory name. Since the copy has the same configuration as the original, its new name can be obtained simply by replacing the serial number at the end of the directory name.

A directory structure that has been obtained as a copy of a Leonardo individual is called a *clone.* A clone can be converted into becoming a new individual, but there are two successive steps that must be taken for this purpose. The first step is *personalization* which results in an *unregistered individual;* the second step is *registration.* Personalization consists of obtaining a few basic facts about the individual which it stores in itself, namely, about its *owner* and its *location,* i.e. what host or detachable memory device it is located on. Registration consists of contacting a *registrar agent,* reporting the information obtained during personalization, and obtaining a directory name for the individual and a proper name for its kernel agent. (The proper names for the other agents in the individual can be derived from the proper name of the kernel).

The naming conventions for directory names that were described above are therefore only applicable for registered individuals. It is recommended that clones and unregistered individuals shall use names of other forms, so that a clone of the registered individual `Pawn-4` may be called e.g. `My-Pawn` in order to emphasize the difference. When it is registered it obtains its definite directory name e.g. `Pawn-12.`

The integrity of individuals is an important concept in the Leonardo architecture: different individuals have different names; each of them has the possibility to evolve in its own ways, and individuals can exchange information and influence each others in various ways.

2.1.4 Reproduction and Distribution

There are three ways whereby new Leonardo agents can be produced. One way is by *cloning* where one takes an existing individual, makes a clone of it, and personalizes the clone. In this way one obtains a new individual with the same configuration as the given one. It will be referred to as an *offspring* of the initial individual.

The second way is by *breeding* where one agent in an individual produces an additional agent in the same individual. The new agent will initially have very little contents, but it can *inherit* information from its 'mother' i.e. use that information while it is still located in the mother; it can *import* information from its mother or from other agents i.e. add it to its own structure, and finally it can be *extended* with additional information that its user writes specially for it.

The third way is by *transplanting* where an entire agent is copied into another individual. The following is an example of a transplanting scenario. An individual with a certain configuration has produced a number of offspring. Then it breeds an additional agent within itself, and the new agent is extended in the ways just described. After this, copies of the new agent can be transplanted into one or more of the offspring in order to provide the additional facilities in them as well.

It follows that when a person wishes to begin using a Leonardo system, then the way to do it is to obtain a clone of an existing individual, to personalize it, and to register it. Both personalization and registration are performed by starting a session in the kernel agent of the individual and performing operations there. This will be described in the following sections.

2.2 Starting a Session with a Clone

The following is what you need in order to begin using Leonardo:

- A computer where an appropriate CommonLisp system has been installed, as described in Chapter 1.
- A clone of a Leonardo individual, in the form of a directory with subdirectories containing the necessary information, located on a memory device of the same computer.

The top-level directory contains one immediate subdirectory called `Indivmap` and it may also contain a subdirectory called `Leohost.` These should be ignored for the purpose of this chapter; all other subdirectories represent agents in the individual. It is likely that one of them will be called `remus` and it is possible that it is the only one besides those just mentioned.

At this point, identify the kernel agent and visit its subdirectory at `Process/main` for example at `remus/Process/main` if the kernel agent is called `remus.` Identify the following files there:

```
startleo.bat
linuleo.lex
```

These are the *invocation files* that can be used for starting Leonardo in a Windows systems, or in a Linux or MacOS system, respectively.

Before invoking the system from either of these files, you need to make sure that it contains the correct path to the CommonLisp implementation on your computer. Edit the file and make sure that this is the case.

The following are examples of the contents of these invocation files.

```
echo %computername% >..\temp\_compnam.sii
start C:\acl82\mlisp.exe -L Boot/cl/startleo.leos
```

and

```
#!/bin/bash
/usr/local/acl/acl81/mlisp8 -L Boot/cl/linuleo.leos
```

If the clone at hand has been copied via an intermediate position in a Windows system before arriving at its present location in a Linux or MacOS system, then it may be the case that the `linuleo.lex` file contains Windows-style characters in the newline position, and that the shell will not be able to start executing it. In such a situation one has to edit or re-type the file so that newline is represented in the correct way for the operating system at hand.

If the invocation file is started from a terminal window in Linux or MacOS then the command-line dialog with the system will happen in that window, otherwise a new command-line window will be opened. In either case Leonardo will begin by displaying a banner similar to the following:

```
;   Loading D:\Pawn-1\remus\Process\main\Boot\cl\bootfuns.leos

************************************************************
Now starting up a Leonardo session  --  please stand by

Welcome to a session with the following Leonardo software agent
   (kercl: soc.root 1 soc.root 46)
This is session number 68 with this agent
Network model in indivmap-kb will be loaded -- done
Server and communication support in els-kb will be loaded -- done.

************************************************************
This session executes on the following Lisp system:

International Allegro CL Enterprise Edition
8.2 [Windows] (Jan 25, 2010 14:53)
Copyright (C) 1985-2010, Franz Inc. ...  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   [TC19961] Linkoping University

************************************************************

ses.001)
```

The last line of asterisks marks that the command-line dialog starts. It may be that the system will ask some questions to the user before that. The command-line dialog begins with the prompt `ses.001)` where `ses` indicates the Leonardo system's "attitude" at this point and `001` is the serial number of the interaction. Subsequent interactions receive the following numbers. The system's attitude specifies how it will deal with various types of situations that may come up, for example, how to react when the preconditions for a command are being violated.

Input to the CLE *command-line executive* shall be on a single line and shall be ended in the usual way with the 'enter' or 'return' key. Useful things to do for a start are described in Section 2.4 and in the separate memo "*Initial Leonardo Exercises.*"

In order to conclude the session with the Leonardo agent one should simply give the following command

```
exit
```

or merely kill the command-line window where the session has been held.

2.3 Startup Dialog and Personalization

Leonardo individuals have a capability for detecting whether they are clones, unregistered individuals, or registered individuals. This is referred to as the *mode* of the individual. Distribution copies of Leonardo systems are normally clones. The mode is detected during session startup, and when a session with a clone is started, then during the startup phase it will propose to the user to do personalization at once, whereby it will become an

unregistered individual. The user may decline the offer and delay personalization to a later occasion, in which case the agent starts the Command-Line Executive (CLE) that is described in the next section.

The individual's current mode is reflected in the session-starting banner which says something like the following in the case of a clone.

```
Welcome to a session with a clone of the following Leonardo software agent
```

to be compared with the phrase for a registered agent,

```
Welcome to a session with the following Leonardo software agent
```

The mechanism for detecting copying, i.e. cloning is not perfect. It will sometimes mistakenly believe that cloning has taken place when it has not, and it can be manipulated so as not to recognize a clone for what it is, but this happens only in special circumstances. In a simple case where you have received a copy of a Leonardo individual for your own use and you have placed it on your own computer or detachable memory device, it will correctly recognize that it is a clone. However, since there are situations when a bona fide *move* operation on an individual is interpreted as a copy, an individual in clone mode will always first ask whether itself is really a clone. If the user answers no then it will reset itself to the applicable non-clone mode.

If the user agrees to personalize the clone then it will inquire about its *owner* and verify the information about its *location.* The owner is represented as an entity of the form `/Familyname.Givenname` , for example `/Svensson.Sven` possibly with a prefix. Any letter on the keyboard at hand may be used, including the use of diacritics such as in `/Ångström.Pär.` Notice that the name components are identified as family name and given name, and not as last name and first name, since the order of the names differs between countries. However, if a user wishes to use more than one name of each kind, or to include other kinds of components in the aggregated name, such as `de` or `von` or `Jr.` then this has to be done in the attributes of the username entity but not in that entity itself.

The username entity begins with a prefix, which can also be the empty string, followed by a slash and then the family name and the given name. The prefix is used for obtaining separate namespaces for different user communities.

The location can be either a *hostname* or a *devicename* depending on whether the individual is stored on a permanent storage unit of a computer host, or on a detachable memory device such as a USB memory stick. In the former case the location entity has the prefix `host.` for example `host.aldebaran` where the part after the prefix shall normally be the name that the host's operating system assigns to it. (Exceptions must be made in case of name conflicts). The names of detachable memory devices are chosen by the user and they should use the prefix `dmd.`

In the case of the user name, the personalization routine will simply ask for the user's family name, then ask for the given name, and then compose the resulting username entity. It will also ask for the user's email address. Username entities are collected in the file `Indivmap/users-catal.leo` and the system will check whether this user entity is already present there. If not, it will be added, with the email address as one of the attributes. If it is already present, then the system will check whether the previously

known email address agrees with the one that has just been entered, and it will react if there is a disagreement. This is done both to keep the email addresses up-to-date and as a safeguard against name conflicts.

In the case of host names, the default is to have a host name that uses the name that the host has in its operating system, e.g. `host.aldebaran.` In principle it should therefore be possible for the routine to find this out itself and without having to ask the user. There are a few problems with this, however:

- In the versions of ACL for Linux and MacOS systems, the existing function for inquiring the operating system about the host name apparently does not work properly.
- If a given community of users and their hosts contains several hosts with the same name according to their operating systems, then some alternative names must be used.

The personalization routine will therefore attempt to identify the appropriate hostname by considering the following alternatives *in reverse order,* i.e. the first listed alternatives are the defaults, and the later listed alternatives are checked first since they allow for exceptions.

- On a Windows system, use the ACL function that obtains the hostname from the operating system.
- On a Linux or MacOS system, check whether there exists a file at `~/Leohost/hostname` and if so, obtain the hostname from the first line in that file. The hostname shall be given *without* the prefix `host.`
- On a Windows system, check whether there exists a file at the location `C:/Leohost/hostname.txt` and if so, obtain it like in the previous item.
- The individual's kernel agent contains a file called `sessionmode.cl` that may provide a host name; if it does then it overrides all the above.

The path to `sessionmode.cl` is

```
remus/Process/main/State/cl/sessionmode.cl
```

if `remus` is the kernel agent. The line in question can be e.g.

```
(setq *current-host* 'cassiopeia)
```

if a hostname is assigned, or

```
(setq *current-host* nil)
```

if no hostname is set there.

If none of these cases applies then the personalization routine will try to do something reasonable in the given situation. The exact routine for this situation is in a state of evolution. In principle it shall ask the user for what is the correct host name, and attempt to create the necessary `hostname` file for storing the information. If some of the above applies then it shall inform the user of the choice so that the user has a chance to override by changing the contents of the `hostname` file or the `sessionmode` file.

Notice that usually it is better to use the `hostname` file rather than the `sessionmode` file, since the former solves the issue once and for all if there are several individuals on the host, or several agents in an individual. If the `sessionmode` file is used for setting the hostname then it must be edited again for each agent in each individual on the host. Also, if the individual is moved regularly between several hosts then a setting in `sessionmode.cl` has to be changed each time, whereas settings in `hostname` handle the situation automatically. The use of `sessionmode` for this purpose shall therefore only be seen as a last resort.

In fact there is yet another possibility, namely the following assignment which is also made in `sessionmode.cl`

```
(setq *default-host* nil)
```

If this assignment is changed from `nil` to something else, then that name will be used with the lowest priority, i.e. if none of the above applies. This is convenient if an individual is moved regularly between several hosts where some of them provide the name using their OS or their `hostname` files, and some others do not. Then `sessionmode.cl` must only be changed for a move between hosts of the latter kind, but not for the former kind.

2.4 Simple Command-Line Inputs

There are three major types of inputs to the command-line executive:

- A *command* consisting of a verb followed by its arguments, if applicable.
- A *term* that is to be evaluated in the Leonardo system and in order to obtain its value.
- An *S-expression* that is to be evaluated by the underlying Common-Lisp system.

In the first case the line input begins with a symbol which is usually formed using alphanumeric characters. In the second case it begins with one of the following characters

```
( [ { <
```

representing a form, a record, a set or a sequence, respectively. In the case of Lisp input, the first character in the line must be a point (full stop) and the rest of the line is the S-expression.

Each input results normally in an output that is shown on the next line, followed by a new prompt. Errors in input syntax and in execution are usually caught and result in some kind of error message, and then a new prompt. The following are a few examples of interactions.

```
ses.041) (+ 4 5)
  => 9

ses.042) (tl <a b c d>)
  => <b c d>
```

```
ses.043) [member c {a b c d}]
  => [true]

ses.044) ssv .x c

ses.045) [member .x {a b c d}]
  => [true]

ses.046) {a .x d}
  => {a c d}

ses.051) put cherry colors {yellow red black}
  put

ses.052) (union (get cherry colors) {white})
  => {yellow red black white}

ses.053) [member red (get cherry colors)]
  => [true]

ses.054) [-member red (get cherry colors)]
  => [false]

ses.055) ssv .chc (get cherry colors)

ses.057) (or [member white .chc][member red .chc])
  => [true]

ses.058) (str.concat "ab" "cd")
  => "abcd"
```

As shown by these examples, if a symbol is preceded by a point then it is considered as a variable. The `ssv` command assigns a value to a variable for the duration of the session. Expressions can be formed as sets and seqences in the obvious way; when they are evaluated then symbols and numbers evaluate to themselves, and variables evaluate to the value that has been assigned to them. Literals i.e. expressions led by a predicate are enclosed in square brackets. The negation of a predicate is obtained by preceding it with a dash character. (The operator `not` is also defined). Composite propositional expressions are formed using the standard connectives and surrounded by round parentheses.

There is a reportoire of functions, including functions that operate on numbers, strings and sequences (lists). There is also a reportoire of command verbs, where the `put` command appears in one of the examples above.

The report "*List Processing in the Knowledge Representation Framework*" contains definitions of basic functions, predicates and commands of these kinds. See also the report "*Knowledge Representation Framework: Overview of Languages and Mechanisms*" for basic syntax definitions.

Input of Lisp S-expressions for direct interpretation by the underlying system is illustrated by the following trivial examples.

```
ses.059) .(list 'a 'b 'c)
(a b c)
```

```
ses.060) .(cddr '(a b c d))
(c d)
```

In a longer perspective it is intended that applications based on the Leonardo platform shall be possible to implement using the CEL language, i.e. along the lines of the first two items above. However, at the present time it is often necessary to resort to Lisp-level programming for some aspects of an application.

2.5 Moving and Copying Individuals

As already described there is a difference between moving and copying individuals. If an individual is moved then it continues to be the same individual; if it is copied then the copy is a clone and it needs to be personalized in order to regain a normal identity in the Leonardo sense.

Each registered individual has a directory name, for example `Pawn-12.` This name is used for some purposes by the Leonardo software, in particular as the argument to some of the commands. It is a directory name in the sense that this name occurs in some of the catalogs or 'directories' that are maintained by the Leonardo software itself. It is also recommended that the directory name is used as the name of the individual's main directory, but this recommendation is because it tends to simplify the handling especially when you have a significant population of individuals, and it has no technical significance.

The following are therefore the operations that you can make to an individual and that will be considered as a 'move' from the Leonardo point of view, i.e. it is still the same individual:

- Rename the main directory of the individual.
- Use the operating system's *move* operation in order to move the individual's directory with all subdirectories to a new place in the same file system.
- If an individual is located on a detachable memory device, move that device to another host.
- Use any facility for moving the individual's directory, with subdirectories, to another host or memory device.

What is in common between these cases is that they do not add another individual besides the given one. The clone detection facility in the Leonardo software will correctly consider the first three of these cases as being a move and not a copy. Even the fourth item is sometimes considered as a move, but we have seen some cases where it was interpreted as a copy, and the system suspected that the moved individual was actually a clone. If this occurs then the impression is corrected by the user during the startup of the first session after the move.

Notice, however, that if you perform an intended move by first copying your individual, in the sense of the operating system, and then deleting the original, then the system will in fact consider your moved individual as a clone.

The Leonardo system's mechanism for detecting cloneship is based on observations of the *date of creation,* in the sense of the file system, of one particular file that is located at

```
remus/Process/clonemon/vino.txt
```

The observations are maintained in the ef `clonemon.leo` in the same directory. The name `vino` alludes to the classical proverb *In vino veritas,* and is considered as an acronym for *Verba individis novi observandi sunt,* or *Notice the words of the new individual.*

Chapter 3

Facilitation and Registration

The simple exercises that were shown in the previous chapter can be done in in an unregistered individual, or even in a clone without taking any further measures. However, some services in a Leonardo individual require the use of other software on the same computer host, for example a text editor, a LaTeX formatter, or a pdf rendering program. Such external software will be referred to as *facilities* for the Leonardo individual, and the use of facilities requires *facilitation* whereby the individual is provided with information about the name, the location, and the means of using the facilities. In fact, facilitation also serves to specify the user's preferences with respect which of several alternative facilities she or he prefers to use for a particular purpose.

For some services it is also necessary to enable Internet-related services in the individual. Besides requiring certain facilities, this also requires that the individual shall *register*. The present chapter describes how facilitation is done in general, and how registration is done in the case where the Leonardo system is executed in the Allegro CommonLisp environment.

3.1 Facilitation

The previous chapter described how an individual may move between different hosts and still retain the identity as being the same individual. However, if the individual needs to use facilities outside itself in the host where it is operating, then it needs information about where these are located in each of the hosts that it is visiting. Two design principles are possible: either the individual will contain information within itself about every host that is of interest, or each host will contain information that can be used by every visiting individual. Both alternatives are made possible in Leonardo, but the latter alternative is the preferred one.

For this purpose each participating host is required to contain a directory called `Leohost` that will contain the information about the facilities in itself. One specific use of this directory has already been mentioned in Chapter 2,

and more will be added here. The customary location of the `Leohost` directory is at

```
C:/Leohost
~/Leohost
```

in a Windows environment and in a Linux or MacOS environment, respectively.

For those cases where it is impossible or not desired to locate them there, there is an exception mechanism whereby one uses a file with the name `leohost-pointer.txt` (in Windows) or `leohost-pointer` (in Linux or MacOS) which shall contain the path for the `Leohost` directory in the host at hand. This file shall be located as a "sister" of the directory of the individual, so it is accessed as `../leohost-pointer` (with the extension if applicable) from the top level directory of the individual.

The location of the `Leohost` directory in the current host is identified at one point in the session startup procedure, and the information in it is loaded accordingly. If no `Leohost` directory is found then the procedure proposes to initialize it. This creates a directory with the appropriate files in the appropriate place, but the exact contents of these files has to be reset by the user.

The `Leohost` directory has the same structure as other knowledgeblocks, so it contains an ef called `leohost-kb` that contains information about the other entityfiles in that block. It also always contains an entityfile called `facilities-catal` whose entities are like in the following examples of simple cases.

```
-------------------------------------------------------
-- facil.iexplorer

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

[: type os-command]
[: commandphrase "C:\Progra~1\Mozilla Firefox\firefox.exe "]
-------------------------------------------------------
```

In simple cases like these, the facility is invoked on a particular file by taking the value of the `commandphrase` attribute, concatenating it with the path to the file in question, and handing the result to the operating system for execution. There are also features that allow more complex invocation behavior, but these have rarely been needed.

When the `Leohost` directory has been initialized, the user must edit the `facilities-catal` file and put the right commandphrase for each of the facilities that is mentioned there or, at least, those facilities that he or she is planning to use.

Since different users may have different preferences about which facilities to use, the `Indivmap` directory also contains an ef `facility-preferences` with entities like the following one.

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

```
-- facil.web-browser

[: type facility-category]
[: linux-preference facil.firefox]
[: windows-preference facil.iexplorer]
-------------------------------------------------------
```

The owner of an individual should therefore check the contents of this file as well and adjust them to his or her preferences.

Providing this information about facilities is sometimes all that is needed, but in other cases there is a lot of additional work for setting up the proper interface between Leonardo and the facility in question. However, doing the facilitation is always the first step.

The AllegroLisp system itself is also a facility in the sense of this section; it is used in cases where an operation during a session with an agent shall start another session in the same host, with another agent or (exceptionally) [1] with the same one. This facility is specified in the same way as the others, and the following is an example of such an item description in a Windows environment.

```
-------------------------------------------------------
-- allegro

[: type os-command]
[: commandphrase "C:\acl82\mlisp.exe -L "]
-------------------------------------------------------
```

3.2 AllegroServe Facility and Catalog of Hosts

3.2.1 AllegroServe

A number of facilities are related to the use of the Internet. This applies to the web browser, which has already been mentioned, but it also applies to web servers and to software for message exchange between agents. The Allegro CommonLisp (ACL) system contains a package called AllegroServe that provides both a web server and procedures for communicating with web servers. Leonardo uses it for these purposes, and in addition it is being used as the basis for agent-to-agent communication. The Leonardo-level software that is based on AllegroServe is located in the knowledgeblock `els-kb`. The startup procedure of a Leonardo session attempts to load this package, but it will only succeed in doing this if AllegroServe has been properly set up in the facilitation phase. The banner in the example of session startup in Chapter 2 contained the following line:

```
Server and communication support in els-kb will be loaded -- done.
```

which is an indication that this was successful. If AllegroServe did not load then it will say `failed` instead of `done,` and in this case the first step must be to correct this problem.

[1] In general it is not recommended to have several concurrent sessions with the same agent, but the system does not prevent it and there are certain situations where it is useful to do this.

The AllegroServe facility is not invoked using a command line, like the ones in the previous examples, but using function calls to a number of functions that are defined in that package. Loading the AllegroServe package is therefore the very first step when loading `els-kb.` For reference, this is done in the Lisp function `load-aserve` in the ef `elsbase` in `els-kb.` Since AllegroServe is a part of the ACL system it is located inside the latter, so `els-kb` just needs to know where ACL is located. It has two ways of obtaining this information. One is using the ef `facilities-catal` in `Leohost` and specifically using the entity `allegrodir` which may look as follows.

```
------------------------------------------------------
-- allegrodir

[: type os-command]
[: commandphrase "C:\acl82\"]
------------------------------------------------------
```

Thus it is similar to the entity `allegro` but without the file name at the end of the commandphrase. The other possibility is using information about the host at hand in the entityfile `individuals-catal.` The other way of specifying the location of ACL, and therefore of AllegroServe, is using the information in `Indivmap.` To see how this is done, the reader is recommended to inspect the entityfile `Indivmap/hosts-catal.leo` which contains entity-descriptions for hosts with names like `host.aldebaran` and with a number of attributes for each host. In order to provide a Leonardo individual with the information that it needs in order that its agents shall find AllegroServe, its `hosts-catal` file must contain an entity for the host where this individual is located, and this entity shall look like the following.

```
------------------------------------------------------
-- host.dell-laptop-1

[: type detachable-host]
[: has-allegro-path "C:\acl82\"]
------------------------------------------------------
```

together with a number of other attributes.

These two ways of specifying the path to the ACL system work equally way, and it is a matter of choice which one of them one shall prefer. If the `Leohost` directory with its contents can be put in place without difficulty then this may be the easiest way; in the opposite case one may opt for using `hosts-catal,` in particular since it will anyway be necessary to set up an entity there that represents the current host, especially in order to prepare for message-passing.

Notice however that regardless of which method is used, this information must be available in each of the hosts that an individual is going to visit, in the sense of executing a session there. These methods can be mixed, so that the path is defined in `faciities-catal` on some hosts and in `hosts-catal` for some other hosts.

3.2.2 The Catalog of Hosts

This subsection describes how to define the location of ACL and AllegroServe using `hosts-catal.` The first thing to do is to identify the name of

the current host. It shall be an identifier beginning with `host.` and followed by the computer name as used by its operating system, all in lower-case letters.

If one should be in doubt about the hostname, then one can find it by evaluating the following expression in the command-line dialog

```
(get current-host value)
```

This will return the entity representing the host at hand according to the criteria that were specified in Section 2.3. If the returned entity should not be the correct one, or be missing, then one should provide this information using the handles that were described in 2.3, and then restart the session.

Once the system has the right identifier for its current host, the next task is to place an entity of this kind in the entity-file `hosts-catal.` The only attribute that is needed at this point is the one leading to the ACL system, so the addition to `hosts-catal` may look as follows.

```
------------------------------------------------------------
-- host.mike's-laptop
[: type stationary-host]
[: has-allegro-path "C:\acl82\"]
------------------------------------------------------------
```

After you have done this edit in `hosts-catal` then continue the session with the following commands.

```
loadfil hosts-catal
loadk els-kb
```

The first command reloads the entityfile that you have just edited, and the second command attempts to load both AllegroServe and the parts of the Leonardo system that builds on AllegroServe. If the second command proceeds without an error message then all is fine.

Alternatively, just start a new session and check that the session-start banner reports the expected

```
Server and communication support in els-kb will be loaded -- done.
```

If this should not work out then you must check the arrangement with the hosts information and try again.

It may of course happen that `hosts-catal` already contains an entity that describes another host but using the same name as your own. This situation can best be handled by selecting another name for the purpose of Leonardo than what is used by the operating system, and to assert it using the handles that were described in Section 2.3.

The `els-kb` package provides the following services for the agent:

- Ability to operate a web server.
- Ability to send a request to another Leonardo agent and make use of its response. The target agent may be located on another host and elsewhere on the Internet.
- Ability to receive messages from other Leonardo agents and to respond to them.

The second one of these services is used when a clone registers itself. When you have arrived to the point where `els-kb` is loaded successfully you can proceed to the next section which explains how this is done.

3.3 Registration

The session startup routines will detect whether the individual at hand is a clone or not, as was described in Section 2.3, and if it is a clone then it will propose to personalize it at once. After personalization it is an unregistered individual, and it can be used as such. However, each time a session is started with an unregistered individual, the startup dialog will suggest that it should be registered. This operation requires very little effort and there is no disadvantage with it. However, a clone must be personalized before it can be registered.

In order to cause the individual to register, you start a session with `remus` or in general with its kernel agent and write the registration command. This will cause the agent to contact the registrar agent in order to obtain serial numbers for the individual and for the kernel agent as described in Subsection 2.1.2, and then to update its own information accordingly. There is therefore a *registrar part* and a *local part* to the registration procedure, where the registrar part takes place in the registrar agent, and the local part is where the registrating individual updates its own structure.

One particular aspect of the registrar part is that it assigns the directory name of the individual being registered. It is recommended that this name shall be used to actually name the main directory of the individual. This may seem like a chicken-and-egg problem, since the individual must have *some* name even when it is in the clone or unregistered mode, but at that point one does not yet know what directory name it shall have. The way to solve this problem is to give the clone any convenient name, and the simplest is to just let it keep the name that it had in the distribution. As soon as the registration has taken place one changes its name to be the assigned directory name. This requires however that you first terminate the session where registration was made, then change the name, and then start a new session if you should need one. Therefore the last step in the local part of registration has to be made manually when the user changes the name of the individual's main directory. [2]

In order to register an individual using a session with its kernel agent, you should first verify that it has loaded `els-kb` correctly as mentioned above, and that the host has Internet access. Then issue the following commands

```
loadk indiv-kb
attre dres.maintain
```

This loads a knowledgeblock containing the commmand for contacting the registrar, receiving its results, and performing the local part of registration. Then, in order to verify that everything is in order, it is recommended to 'ping' the appropriate registrar by issuing

```
pingreg
```

[2] We expect to automate this process in due time, but it has not been done yet.

If this operation is confirmed then one uses the following command for requesting the individual to be registered.

```
indreg
```

If registration is successful then this will have the effect of making the required changes in the kernel agent and in the shared `Indivmap` structure. If the individual contains other agents besides the kernel agent then they will not be affected, but each of them will notice that registration has occurred the next time a session is started in it, and then it will also update itself automatically. In order to see how this happens, look at the following file in the agent

```
Process/main/Defblock/selfdescr.leo
```

before and after the start of the first session with a non-kernel agent.

Therefore, the only thing that you have to do as a user after the successful completion of the `indreg` command is to close the session and change the name of the individual's directory as just described. If you should accidentally lose it then the name can be obtained using the command

```
my-dirname
```

in a session where `loadk indiv-kb` has been performed.

3.4 Representing Ownership of Individuals

Each individual has an *owner* which is specified in two places. The entityfile `individuals-catal` in `Indivmap` contains descriptions of all individuals that are known to the individual at hand, including itself, and with the information about who is the owner of each of them as one of the attributes. Moreover, each individual has an entityfile `indiv-descr` in `Indivmap` that contains one single entity called `current-individual` that the individual uses for information about itself. In both places the owner is represented using the `has-owner` attribute, for example as

```
[: has-owner /Sandewall.Erik]
```

This attribute is prompted for during personalization, but at that point it is only assigned to the attribute for `current-individual.` The registration operation obtains the network name for the current individual, and this is the entity that is added to `individuals-catal` again with the `has-owner` attribute.

Ownership can be changed at any time, and not merely at personalization time, using the following command *Not yet implemented, sorry.*

```
new-owner /Larsson.Lars
```

This will update the local `has-owner` attribute and send the information about the new owner to the registrar.

The `new-owner` command will first check whether the identifier for the new owner is known in the `users-catal` file of the individual. If not, it will prompt the user for information about this owner. Afterwards, when the agent contacts the registrar for the change of ownership, it may again happen that the registrar does not have any information about the new owner,

and in this case it will ask the "client" (i.e., the agent requesting the change of ownership) for this information. – The information about users and agent owners is limited to the most superficial one: correct spelling of given name and family name, email address, and homepage URL.

3.5 Specifying the Host and LAN

A computer host is said to be *mobile* for the purpose of Leonardo if it is attached to different local area networks at different times, which of course occurs frequently for laptop computers. Similarly, a Leonardo individual is said to be *mobile* if it is located on a detachable memory device that is used with different computers at different times, or if it is permanently located on a memory device of a mobile host. Therefore the distinction hinges on how the host and the memory device in question is actually used, and not merely on its physical characteristics.

The entityfiles in the `Indivmap` structure contain a model of a "society" of individuals and their infrastructure, including hosts, detachable memory devices, local area networks, and so forth. This model is used so that an agent can send a message to another agent merely by mentioning its network name, or even its proper name, and so that this will work even in the face of mobility of agents. At the same time, the use of mobile individuals does complicate this model somewhat, compared to what would have been the case if all individuals had been stationary.

3.5.1 Specifying the Host and LAN for Stationary Individuals

Not yet written. The present ad-oc advise is to look into the files

```
Indivmap/hosts-catal
Indivmap/lan-catal
```

and to proceed by analogy with the existing contents. If in doubt refer to the report "*Internet Related Services in Leonardo.*"

3.5.2 Specifying the Host and LAN for Detachable Individuals

Temporary contents of this subsection. In order not to complicate matters too much at this point, we recommend the reader to avoid these problems in the following ways. First, when asked by the system whether the host at hand is stationary or detachable, say that it is stationary. Also, when asked whether the individual is on a fixed or detachable memory device, say fixed.

Then, if the host at hand is in fact moved to another LAN, then visit the file `Indivmap/hosts-catal.leo,` find the entity for the present host, and update its `in-lan` attribute accordingly. Make sure that the LAN that is being referenced is also defined in the entityfile `Indivmap/lan-catal.leo.`

Similarly, if the individual is stored on a USB or other detachable memory device and this device is used on another host than before, then visit the

file `Indivmap/individuals-catal.leo`, find the entity for the present individual, and update its `in-host` attribute accordingly. It will probably be clear from comparison with neighboring entities how this is to be done, but in case of doubt please consult also in this case the report "*Internet Related Services in Leonardo.*"