KRF

Knowledge Representation Framework Project

Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Using the Lisp Language in the Knowledge Representation Framework

This series contains technical reports and tutorial texts from the project on the Knowledge Representation Framework (KRF).

The present report, PM-krf-024, can persistently be accessed as follows:

Project Memo URL:	`http://www.ida.liu.se/ext/caisor/pm-archive/krf/024/`
AIP (Article Index Page):	`http://aip.name/se/Sandewall.Erik.-/2011/002/`
Date of manuscript:	2011-01-04

Copyright: Open Access with the conditions that are specified in the AIP page.

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

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

Chapter 1

Simple Lisp Programming

We shall first address the situation where the Leonardo-level data structures are not used at all, except that each Lisp function definition is also defined as a Leonardo entity. The present chapter will describe things that one needs to know in order to write simple Lisp programs in this way. Chapter 2 will add a number of other things that may also be useful to know in such a context. Subsequent chapters will address the situation where one wishes to operate on Leonardo-level data structures such as sets and sequences.

1.1 Lisp Execution and Debugging Sessions

A session with a Leonardo system can be in Lisp command-loop mode or in Leonardo command-loop mode. In Lisp mode it receives one S-expression at a time, evaluates it and returns the value. It can also receive and operate on a number of session commands, in particular the command `:zoom` which is used when an error has occurred, and which prints out a summary of the current stack. Entry of the expression `(cle)` switches to Leonardo mode.

In Leonardo mode, entry of the expression `lisp` returns to Lisp mode. The major types of input in this mode are:

- Enter an SCL command. The outermost surrounding square brackets are optional and can be omitted.
- Enter a stop character (.) followed by an CEL expression. The expression is evaluated and its value is printed. This is actually a special case of the previous item, with the stop character as the verb.
- Enter an S-expression which is a list (not a single symbol). It is evaluated by the Lisp `eval` function and the value is printed, i.e. it behaves like the Lisp-level command loop.
- The particular command `lisp` returns to the Lisp level

The session can be exited by typing `(exit)` to the Lisp level and by using the command `exit` to the Leonardo level.

Notice that if you happen to type `(cle)` to the Leonardo level then the Leonardo command loop will call itself recursively, which means that when you then type `lisp` you get back to the next-lower level of Leonardo command loops, and you have to do `lisp` repeatedly if you wish to get back to the Lisp level.

1.2 Lisp Function Definitions in Entityfiles

When Lisp code is written in the Leonardo environment, it is natural to organize it in entityfiles, with one entity for each function definition. The following is a simple example of an entity description for a Lisp function.

```
--------------------------------------------------------------
-- square

[: type lispdef]

(defun square (x)(* x x))

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

It is natural to let the entity have the same name as the function, but this is not obligatory, and exceptions are necessary if the function will have a name that has already been taken as an entityname in the Leonardo system.

The type `lispdef` or its subsumees should be used for definitions such as this one. Older code in Leonardo often uses `entity` as a type for all purposes, but this practice is being phased out.

When an entityfile containing this entity description is read, then a string consisting of the line(s) for the function definition is assigned as the `leo-definition` property of the entity `square,` so the above is equivalent to writing

```
--------------------------------------------------------------
-- square

[: type lispdef]

@leo-definition
(defun square (x)(* x x))

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

The general rule is that when the entityfile parser is reading the maplet section of an entity description and encounters a line containing a left parenthesis in its first column, then it considers the maplet section to be finished and the present line as being the first line of a property for `leo-definition.` An implicit `leo-definition` property must precede any other property.

A simple define-and-test cycle is organized by having a Leonardo session and a text editor open concurrently on the computer screen, by editing entityfiles containing `lispdef` entities in the text editor, and by reloading them into the session after each edit. Other and more advanced setups may also be considered, of course.

1.3 Debugging and Loading

The Leonardo system does not provide any additional support for debugging, besides what is provided by the host system. The best way to debug cases that actually run into errors from the point of view of the Lisp interpreter is therefore to work on the Lisp level. The `cle` executive traps many errors but does not provide good information about them, so for nontrivial debugging it is recommended to leave the command-line executive and work directly on the Lisp level.

On the other hand, for those debugging situations where your definitions obtain a value without an error from the point of view of the system, but the value obtained is not the one you wanted, in those cases one may as well work on the Leonardo command-loop level.

In any case, each time you need to change a function definition, you will do it by text-editing the definition in its entityfile and then reload that file. This is done using the Leonardo-level command `loadfil` with argument, or `loadf` without argument, as defined in the List Processing compendium. If you are working on the Lisp level, then this requires using `(cle)` to get to the Leonardo level, doing `loadf,` and then going back to the Lisp level. More conveniently, however, one can use the Lisp function `load-ef` which corresponds to the `loadfil` command, and write e.g.

```
(load-ef 'myfile)
```

to the Lisp-level executive.

Chapter 2

Writing Lisp Programs in a Leonardo Environment

The parts of CEL that have been described so far can readily be translated to `Lisp,` simply by mapping CEL sequences to Lisp lists, and by allowing symbols, strings and numbers to remain as they are. The present chapter is intended to contain the material that is required in order to write and run *simple examples of* Lisp programs in a Leonardo environment. It does not address the topic of how to define additional functions and command verbs using Lisp-level programming; for this please refer to the separate report "Lisp-Level Programming in Leonardo."

2.1 Introductory Steps

2.1.1 Step I: Defining CEL Functions on Sequences

We recommend as first exercises to write and test some simple definitions of CEL functions on sequences. This is done using the following steps. Notice that the outermost square brackets in the commands shall not be typed to the command line executive.

- Start a session with a Leonardo agent.
- Issue the command `[setk test-kb]` in order to select the knowledgebase `test-kb` which is predefined in the course variant of Leonardo agents.
- Create an entityfile for your definitions by issuing the command `[crefil myfile]`
- In the directory structure of your agent, identify the file `Test/myfile.leo` and open it for editing in your favorite text editor.
- Add your function definitions in that file, immediately before the concluding line consisting of a sequence of the form `ooooooooooo.` A correct file shall consist of entity-descriptions that are separated by `---------` lines, and ending with a `oooooooooo` line.

- When you have entered your definitions, go back to the session and issue the command `[loadfil myfile]` (in the example) in order to load the entityfile with your new definitions.
- Test your definitions by issuing evaluation commands consisting of a point followed by the expression to be evaluated.
- Iterate this process so as to debug your definitions and adding further ones.

For example, you may choose to enter the definition of `abs` that was shown above, including its `Execute/CEL` property but without necessarily including its `Execute/Lisp` property, and then test it by entering command lines such as

```
.(abs -4)
```

It is a good idea to try some recursively defined functions, and some cases where one function calls another one. Write functions on sequences and not merely functions on scalars.

2.1.2 Step II: Defining Lisp Functions on Sequences

When you are comfortable with writing the CEL definitions, then proceed to doing the same with Lisp functions. Notice the straightforward correspondence between the two types of functions. Section 5.3 (forthcoming) will contain a table with the names for built-in functions in Lisp.

For example, the following CEL definition

```
[def [equal (foo .a .b)
            (concat <(e1 .a)(e1 .b)> (t1 .a) (t1 .b)) ]]
```

can be translated to the following Lisp definition:

```
(defun foo (a b)(append (list (car a)(car b)) (cdr a) (cdr b)))
```

2.1.3 Step III: Try using the CEL-to-Lisp Translator

Since there is such a direct correspondence between the sequence-oriented subset of CEL and the basic parts of Lisp it has been natural to write a translator that does the job automatically. The translator is defined in the entityfile `gxl-to-lisp` which is not loaded in the standard startup sequence, so one has to make a separate command

```
loadfil gxl-to-lisp
```

in order to load it. After this the translator is obtained under the Leonardo command-line executive (CLE) using the command `coe.lisp,` like in

```
coe.lisp abs
```

and under the Lisp command-line loop by evaluating

```
(coe.lisp 'abs)
```

For example, the `abs` function can be defined for use in both CEL and Lisp using the following entity description.

```
---------------------------------------------------------------
-- abs

[: type leofunction]

@Comment
Obtains the absolute value of the number given as argument.

@Execute/GSL
[com.seq
   [def [equal (abs .a) (if [ls .a 0] (- .a) .a)]
   [coe.lisp abs] ]

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

For such a simple definition one can of course just as well write out both definitions directly, but for more complex definitions it may be worthwhile to use the translator. It can also be used for checking that you have understood the use of the `quote` operator in Lisp correctly.

2.2 The Syntax of CommonLisp S-expressions

As you proceed to writing more complex function definitions directly in Lisp it is important to know about the detailed rules for how such definitions may be written. This is the topic of the present section. Additional and even more detailed information can be obtained from a Lisp textbook, or from the on-line documentation of the Allegro CommonLisp system.

Scalars. Full CommonLisp contains a fairly extensive set of datatypes, but the classical ones in Lisp are the *symbol* , the *string* and the *number.* There are also separate datatypes for *characters* and for *arrays,* to mention just a few. These will not be considered in the present compendium.

Strings are written enclosed by the double-quote character. They may contain any of the whitespace characters. A double-quote within a string shall be preceded by a backslash character which is used as escape character. A backslash character inside a string must therefore be preceded by an additional backslash.

Numbers are written in their standard ways; please see the on-line documentation if further detail is required. Varieties of numbers include integers, single-precision reals, double-precision-reals, large integers, and others.

Symbols are normally written as a sequence of characters that is not allowed to contain a whitespace (i.e. blank, newline, or tab character) nor any of the characters shown on the following line

```
( ) " ' ` , \ | #
```

In those exceptional situations where one wishes to use symbols containing any of these characters, they can be written by preceding each such character with the backslash character, or by enclosing the entire symbol with the vertical-line character. In the latter case the backslash serves as escape character, which is needed if there is a backslash or vertical-line character within the symbol.

Any sequence of characters that can be interpreted as a number will normally be interpreted as such, and must be marked using backslash or vertical line if it is intended as a symbol.

Lists. The textual representation of a list is a string consisting of a left parenthesis, a sequence of *list elements* optionally separated by whitespace characters, and a final right parenthesis. A list element can be either a symbol, a string, a number, or (recursively) a list. If two successive elements are are both non-lists then they must be separated using some whitespace, otherwise the whitespace separation is optional.

Full Lisp also allows a particular kind of lists which are written as for example

```
(a b c d e f . g)
```

where the final element – `g` in the example – is separated from the main part of the lisp using a point character. We disregard this construct in the present compendium.