CASL

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

Erik Sandewall

Lisp-Level Programming in Leonardo

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.

1.4 Attached Functions and Multiple Definitions

A characteristic aspect of Lisp programming is the use of *attached functions,* that is, Lisp functions that are assigned as properties of Lisp symbols. This is a standard way of implementing plug-ins. Such definitions can be written as in the following example.

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

[: type lispdef]

(leodef (square checkfn) (x) (numberp x))

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

This assigns the function `(lambda (x)(numberp x))` as the `checkfn` property of the symbol `square.` To make several definitions for the same entity one can use the `deflist` operation like in the following example.

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

[: type lispdef]
```

```
(deflist
   (defun square (x) (* x x))
   (leodef (square checkfn) (x) (and (numberp x)(not (equal x 0))))
   (leodef (square effectp) (a)(> a 0)) )

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

In this somewhat artificial example, the `checkfn` checks that the proposed arguments for invoking the function are correct, and the `effectp` specifies a property that shall hold for the returned value.

Chapter 2

Additional Aspects of Lisp Programming

2.1 Leonardo Macros

Definitions that are made using `defun, leodef,` and under `deflist` are subjected to the *Leonardo macro expander* before the assignment of function definition is made. Understanding the macro expander is not required knowledge in order to do Lisp programming for Leonardo, but it is used for some facilities that are of immediate use for the programmer, in particular, the access operations to Leonardo datastructures and the facility for System Output Phrases both of which are described below.

2.2 Defining Forms for Load-time Execution

Sometimes there is a need to specify a form for execution when an entityfile is loaded, for example for loading a package in the host CommonLisp system, or for setting some global information. The following is a simple example.

```
-------------------------------------------------------------
-- setnotrace

[: type loadtime-operation]

(setq *trace* nil)

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

The effect of this will be that the assignment to `*trace*` is made each time the entityfile is loaded and reloaded. This is sometimes fine, but if it is desired that the form shall only be executed the first time that the entityfile is loaded then there are two possibilities. The first possibility is the obvious one, namely, by writing the program so that it sets a flag the first time the file is loaded, and so that it refrains from the assignment if the flag has been set.

The other possibility is as shown by the following modification of the example.

```
-------------------------------------------------------------
-- setnotrace

[: type loadtime-operation]

@Exec-leos
(setq *trace* nil)

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

When the entityfile is loaded in the regular way then the form will not be executed; it will merely be stored as the `Exec-leos` property of `setnotrace.` However, for this particular property name, the `.leos` version of the entityfile contains the form as such, and not as a property assignment. The `.leos` version of an entityfile contains the same information as the `.leo` version, but expressed in terms of Lisp S-expressions so that it can be loaded into a Lisp system *before* the entityfile parser and other facilities have been loaded. Therefore, the first files that are loaded during system startup are loaded by their `.leos` version, like the parser itself for example.

In this way, a form in an `Exec-leos` property in a system startup file will be executed when the file is loaded the first time, but if the user later edits and re-loads it then those items in the file that have been written as `Exec-leos` properties are protected from execution.

Chapter 3

Lisp-Level Extensions to Leonardo

3.1 Internal Lisp Representation of KRE Data

Knowledge Representation Expressions are of course encoded as list structures within the Leonardo system, and the conversion between the textual or "serial" representation in entityfiles and the internal representation is done by the parser and the printer in the `core-kb` knowledgeblock. When Lisp code needs to access these datastructures, it can do so either by manipulating the list structures directly using standard Lisp function, or by using dedicated primitive operations that are available in Leonardo. The latter method is to be preferred in principle, since it makes the code less vulnerable to possible future changes in the internal representation, but it must be admitted that a large part of the existing Leonardo code has been written using the first method.

However, regardless of which method is used, it is necessary to know how the internal representation is done, for the simple reason that you need to be able to read and understand the list structures that arise and that are being operated on by your own program. It is inconvenient to always have to view them in their Leonardo (KRE) form, and if a program bug has resulted in a list structure that is not a correct representation of KRE this is not even possible.

The rules for the internal representation are simple.

- Leonardo untyped symbols are represented as Lisp symbols.
- Leonardo numbers are represented as Lisp numbers.
- Leonardo strings are represented as Lisp strings.
- A Leonardo sequence < `e1 e2 ...`> is represented as a Lisp list of the form `(seq& (e1 e2 ...))`.
- A Leonardo set { `e1 e2 ...` } is represented as a Lisp list of the form `(set& (e1 e2 ...))`.
- A Leonardo form `(fn a b ...)` is represented as a Lisp list of the form `(form& (fn a b ...))`.

- A Leonardo composite entity `(fn a b ...)` is represented as a Lisp symbol with a `symbexpr` property that is the list `(fn a b ...)`.
- A Leonardo variable `.vbl` is represented as a Lisp list of the form `(form& (param vbl))`, and is therefore equivalent to writing `(param vbl)` on the KRE level.
- The following shows a Leonardo record and the Lisp list that represents it:

  ```
  [op a1 a2 ... :t1 v1 :t2 v2 ...]
  (rec& op (a1 a2 ...) ((:t1 v1)(:t2 v2) ...))
  ```

A composer `fn` for composite entities shall have the type `Ecomposer` and shall have a Lisp property called `symbfun` which is a function that takes the arguments of the composer and returns a Lisp symbol that is a unique representation for the composite entity. The following is an example entity definition for such a composer.

```
------------------------------------------------------------
-- small

[: type lispdef]

(leodef (small symbfun) (x)
    (let* ((xpr (list 'small x))
           (id  (intern (princ-to-string xpr))) )
          (setf (get id 'symbexpr) xpr)
          id ))

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

In this way an expression `(small elephant)` will be represented internally as the Lisp symbol `|(small elephant)|` whose `symbexpr` property is the Lisp list `(small elephant)`. This is the usual way of choosing the symbol name, but it is not an obligatory one. The user may choose to make additional assignments in the `symbfun`, for example assigning the `type` of the generated symbol. Notice in particular that this function is called every time the Leonardo parser encounters an expression using the composer in question, so if the function is going to assign initial values for attributes that will be changed later on during the session, then it must be careful to check whether the attribute has already been assigned.

The parser distinguishes between forms and composite entities by the existence of a `symbfun` property for the operator. A composite entity is generated if there is such a property, *except* if some of the arguments is a Leonardo variable or a form. In those cases a form is created, in spite of the existence of a `symbfun`. This is significant for all programs that operate on conditions, propositions, or other constructs that use variables, since these must be able to relate the distinct representations for expressions containing variables and variable-free expressions. It also means that attributes can be assigned values for variable-free terms using symbolic functions, but not for those containing variables.

Some notational variants are accomodated by the Leonardo parser and results in the same internal representation as the base notation. As described in the KRF Overview report, some operators can be declared for infix use, so that one can write e.g. `(a + b + c)` equivalently with `(+ a b c)`. Similarly, it possible to write records so that some or all of the parameters precede the arguments, for example

```
[op :t1 v1 ^ a1 a2 ... :t2 v2 ...]
```

In all such cases a single representation is used internally and the choice of notational variant is lost. This means for example that when the object in question is produced in output it will always be the standard variant with prefix operators and parameters last that will appear.

3.2 Function and Macros for KRF Data

Rather than referring directly to the Lisp-level representation of KRE data using `car` and `cdr` functions, one may wish to use a set of access functions that isolates one's program from that representation, partly for readability of the code, and partly in order to prepare for a possible future change of representation. For performance reasons this is best done using macros, rather than ordinary Lisp functions. The entityfile `leoper` in `core-kb` contains a collection of such macros. The following is a concise account of them.

It must be admitted however that most of the existing Leonardo code has been written without the use of these macros.

3.2.1 Sequences

`(seq-test x)` is true if `x` is a sequence.

`(seq-body x)` takes a KRE sequence as argument, and returns the corresponding Lisp list. It is defined as `(cadr x)`.

`(make-seq x)` is the inverse of `seq-body`: it takes a Lisp list as argument and returns the Leonardo representation of the corresponding sequence. It is therefore defined as `(list 'seq& x)`.

`(add-seq x y)` takes an element `x` and a sequence `y` and returns an extended sequence where `x` is the first element, analogous to `cons`.

`(radd-seq x y)` is similar to `add-seq` but it does the operation destructively, i.e. any other reference to `y` is affected by way of side-effect.

3.2.2 Sets

The functions `set-test, set-body, make-set, add-set` and `radd-set` are analogous to the corresponding functions for sequences.

`(set-member x s)` returns a true value iff `x` is a member of the set `s`. It is actually defined as `(member x (cadr s))`.

3.2.3 Records

The function `rec-test` is analogous to the corresponding functions for sequences.

`(rec-type r)` returns the record former of the record `r`. Defined as `(cadr r)`.

`(rect-ut-body r)` returns the Lisp list of the arguments of the record `r`. The mnemonic `ut` is for "untagged."

`(rec-t-body r)` returns the parameters of the record `r`, represented as a Lisp list of pairs, i.e. `((tag1 val1)(tag2 val) ... )` where each of the `tagi` shall be a tag, for example `:color`.

`(rec-nth i r)` obtains the `i` -th argument of the record `r`, counting from 1 and up.

`(rec-t f r)` obtains the value of the `f` field of the record `r`. The first argument should be a tag, for example `:color` .

`(make-recsep rf a p)` returns a record where the symbol `rf` is the record former, the Lisp list `a` is the list of arguments, and the binding list `p` is the set of parameters. Therefore

```
(make-recsep (rec-type r) (rec-ut-body r) (rec-t-body r))
```

returns a record that is equal to `r`.

3.3 Defining Leonardo Commands

The following is a modification of the previous example (in Chapter 1) where the function can also be used as a Leonardo command.

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

[: type lispdef]

(leodef square sq (x)(* x x))

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

This entity description defines `square` as a Lisp function, like before, and it also defines `sq` as a command verb with the same effect as the function. When used as a command verb, the returned value is taken over by the command-line executive being used, such as the `cle` executive. This executive may choose to just show the returned value on the screen, or it may take additional action for presentation purposes. Different executives treat this differently.

If only a command definition and not a function definition is desired then one shall give `nil` for the function-name argument.

The internal representation of command-verb definitions is as follows. The `cle` command assumes that the definition of a command verb is stored in the `elem-perform` property of the same verb. The `ble` command executive (primitive, rarely used) assumes that is stored as a function in the `apdef` property of the verb. Definitions using `leodef` assign both these properties, so the user will not notice any difference.

If a symbol is defined as both a function and a command verb, then the full definition is assigned using the `defun` operator, and the `apdef` and `elem-perform` properties will just contain a `lambda` expression that immediately calls the function as defined by `defun.` In the example above, the `elem-perform` property of `sq` will be `(lambda (x)(square x)).` If the function name is given as `nil` then the full definition will occur directly in the `apdef` and `elem-perform` properties.