Nano-Lisp
The Tutorial Handbook

Francis Sergeraert

March 4, 2006

1 Various types of Nano-Lisp objects.

There are several type notions and in this documentation only the notion of *implementation type* (itype in short) is considered. There are only nine itypes in Nano-Lisp:

- *itype*: the itype of itype descriptors;
- *error*: the itype of error messages;
- *boolean*: the itype of boolean-objects;
- *symbol*: the itype of Nano-Lisp symbols;
- *list*: the itype of Nano-Lisp lists;
- *lambda*: the itype of user-defined functions;
- *macro*: the itype of user-defined macro definitions;
- *special*: the itype of predefined Nano-Lisp functional objects;
- *system*: the itype of "system" functional objects.

Each Nano-Lisp object has one and only one itype, so that the set of itypes defines a partition of the (countable) set of Nano-Lisp objects in nine subsets. The user can only enter objects made of symbols and lists; such an object is an *input-object*.

A symbol is a character-string satisfying the traditional rules of programming languages; the following rule is not complete but sufficient: a symbol is a character-string made of letters, digits and hyphens (that is, '`-`'), beginning with a letter: `example-of-symbol`, `a-symbol-with-the-digit-6`, `a`, `abc`, `a314159`, etc.

An input-object is a symbol or a list of input-objects. A *list* is a character string beginning with an opening parenthesis '`(`', followed by some objects separated by spaces, and ending with a closing parenthesis '`)`'. A list element may be also a list, and so on. Examples of input-objects:

```
a-symbol-which-is-an-input-object
(an input-object which is a list of symbols)
((the first list element) (the second list element))
((((((((((()()()))))))))))
```

A list may be empty but a list having a unique element, the empty list, is not an empty list: `(())`. This is the empty list: `()`.

Nano-Lisp handles and produces more general lists than the input-object lists; for example an element list could be a boolean-object, a lambda-object, a special-object; but such lists cannot be directly entered by the user.

Usually the evaluation process brings into play objects of any itype. The objects of itype *itype*, *boolean*, *special* and *system* are located by some predefined symbols, the value of which cannot be modified; such objects can also be used during an evaluation process. An error-object is created by Nano-Lisp when some wrong condition happens in a computation or can also be produced by the user himself when he stops a computation.

For Computation Theory, the macro- and system-objects are useless. The macro facility provides a powerful and convenient tool to extend the language kernel, but will not be used in the Nano-Lisp proof of Gödel's theorem.

The system-objects give the user various tools, mainly to help debugging.

2 The environment.

Nano-Lisp maintains an *environment* which is a (finite) ordered list of pairs `{symbol ⇒ Nano-Lisp object}`. The symbol is the *source* of the binding, the Nano-lisp object is its *target*. The environment is organized as a stack and the value of a symbol is defined if some pair of the environment begins with this symbol; the *value* of the symbol is then the second component of the highest pair of this environment beginning with this symbol. The user can modify the environment in various ways. The environment begins with the *global* environment and above this one there is the *local* environment. The following figure describes this organization:

`--`	← top of the local environment
`{a ⇒ value-3-of-a}`	
`--`	
`{a ⇒ value-2-of-a}`	
`--`	← bottom of the local environment
`--`	← top of the global environment
`--`	
`{a ⇒ value-1-of-a}`	
`--`	
`--`	← bottom of the global environment

This figure shows a situation where the symbol `a` is present three times in the environment, two times in the local one, one time in the global one. If the value of the symbol a is asked in a computation, the highest pair in the environment beginning with the right symbol is used and in this case the value of the symbol a would be the symbol value-3-of-a. If no right pair is present in the environment with the symbol asked for as the first component, this is an error, the computation stops and an error-object is returned.

In fact, below the global environment there is also the *system* environment (not showed above) which contains all the predefined system bindings. This system environment cannot be modified and no system symbol can be present in the user global and local environments; if you try to violate this rule, an error will be generated.

```
NL > (set ’progn ’anything)
<ERROR "The symbol PROGN is a system symbol">
NL > (environment s)
S-ENV -> ITYPE-ITYPE -> <ITYPE ITYPE>
S-ENV -> ERROR-ITYPE -> <ITYPE ERROR>
S-ENV -> BOOLEAN-ITYPE -> <ITYPE BOOLEAN>
S-ENV -> SYMBOL-ITYPE -> <ITYPE SYMBOL>
S-ENV -> LIST-ITYPE -> <ITYPE LIST>
S-ENV -> LAMBDA-ITYPE -> <ITYPE LAMBDA>
S-ENV -> MACRO-ITYPE -> <ITYPE MACRO>
S-ENV -> SPECIAL-ITYPE -> <ITYPE SPECIAL>
S-ENV -> SYSTEM-ITYPE -> <ITYPE SYSTEM>
S-ENV -> FALSE -> <FALSE>
S-ENV -> TRUE -> <TRUE>
S-ENV -> PROGN -> <SPECIAL PROGN>
S-ENV -> IF -> <SPECIAL IF>
S-ENV -> WHILE -> <SPECIL WHILE>
S-ENV -> QUOTE -> <SPECIAL QUOTE>
S-ENV -> EVAL -> <SPECIAL EVAL>
S-ENV -> ITYPE -> <SPECIAL ITYPE>
S-ENV -> ERROR -> <SPECIAL ERROR>
S-ENV -> SET -> <SPECIAL SET>
S-ENV -> LAMBDA -> <SPECIAL LAMBDA>
S-ENV -> MACRO -> <SPECIAL MACRO>
S-ENV -> EQUAL -> <SPECIAL EQUAL>
S-ENV -> CONS -> <SPECIAL CONS>
S-ENV -> FIRST -> <SPECIAL FIRST>
S-ENV -> REST -> <SPECIAL REST>
S-ENV -> PRINT -> <SYSTEM PRINT>
S-ENV -> STEP -> <SYSTEM STEP>
S-ENV -> ENVIRONMENT -> <SYSTEM ENVIRONMENT>
<SYSTEM ENVIRONMENT>
NL >
```

You see the statement (environment s) displays all the system symbols and their corresponding values. Because progn is a system symbol, it is not possible to try to define a new value for it.

3 The evaluator.

The precise *mathematical* definition of Nano-Lisp (and of any Lisp dialect) is a function:

$$\textit{evaluator} : (\mathrm{env}_1, \mathrm{obj}_1) \longmapsto (\mathrm{env}_2, \mathrm{obj}_2)$$

which associates to a pair (environment + object) a new pair (environment + object). The environment notion is explained in the previous section; it is a binding set satisfying several precise organization rules. One says the object obj_1 is evaluated in the environment env_1. The evaluator *returns* the object obj_2; furthermore maybe the previous environment has been modified; in other words $\mathrm{env}_1 \neq \mathrm{env}_2$; if so, one says the evaluation of obj_1 in the environment env_1 has produced *side effects*. But it is also possible that $\mathrm{env}_1 = \mathrm{env}_2$ and then the evaluation was *without side effects*.

In the *read-eval-print* terminology, the object obj_1 is read by the Lisp reader, then evaluated in the environment env_1; the returned object obj_2 is *printed* (that is, displayed) and the new environment is env_2. In an interactive session, probably a new object will be read and evaluated in the environment env_2 to produce a new result and a new environment, and so on.

The definition of the evaluator is not so simple and is essentially the matter of the *Reference Manual* of this series. The evaluator is more or less the Central Processing Unit (CPU) of a Lisp machine. This theoretical and practical machine definition has a special behaviour: the definition is *recursive*; for example evaluating a list usually needs evaluation of every list member: the evaluator must *call the evaluator*. All these points will be detailed later.

4 Handling symbols.

For example:

```
NL > b  ==>
<ERROR "The symbol B is unbound">
NL >
```

It will be explained later that for associating to the symbol `b` a value which is the symbol `value-for-b` you must proceed in this way:

```
NL > (set ’b ’value-for-b)  ==>
VALUE-FOR-B
NL > b  ==>
VALUE-FOR-B
NL >
```

The special-object value of the symbol `set` (in the initial environment) allows the user to *add* to the environment a pair `{b` $\Rightarrow$ `value-for-b}`. Don’t forget to *quote* the symbols `b` and `value-for-b` in the `set` statement for an important reason which will be explained later. Quoting an object consists in keying a quote character just before it.

If the symbol argument of the set statement is already present in the environment, then the highest pair of the environment is modified to introduce the new desired value; the old one is thrown away. This is true even if the highest pair is in the local environment (this is an important difference with Common-Lisp).

```
NL > (set ’b ’new-value-for-b)  ==>
NEW-VALUE-FOR-B
NL > b  ==>
NEW-VALUE-FOR-B
```

Note also that the `set` statement *returns* the *target* of the new binding added or modified in the environment; this is frequently useful.

The initial (system) environment contains exactly 30 pairs allowing the user to reach the 9 itype-objects, the 2 boolean-objects, the 14 special-objects and the 5 system-objects. In the following examples, you see the itype-object describing the symbol itype is the value of the symbol `symbol-itype`; the symbols `true` and `false` point to the two boolean-objects; the symbol `set` points to the special-object allowing the user to modify the environment (later fully described); the `environment` symbol points to the system object allowing the user to get some (probably debugging) information about the current environment:

```
NL > symbol-itype  ==>
<ITYPE SYMBOL>
NL > true  ==>
<TRUE>
NL > false  ==>
<FALSE>
NL > set  ==>
<SPECIAL SET>
NL > environment  ==>
<SYSTEM ENVIRONMENT>
NL >
```

It is important to understand that the *symbol* `true` *is not* the boolean displayed `<TRUE>`; this boolean is only the target of a binding where the source is the symbol. If the user enters the character string “`<TRUE>`”, Nano-Lisp will generate a reader error:

```
NL > (set ’<TRUE> ’some-value)  ==>
<ERROR "Reader: a wrong object:
       <TRUE>">
NL > (set ’some-symbol ’<TRUE>)  ==>
<ERROR "Reader: a wrong object:
       <TRUE>">
```

At the top level of the interpreter, the local environment is always empty.

5 Some simple examples.

This section contains only a few examples of elementary list handling in order to have a little material to continue explaying how the Nano-Lisp machine works. You can assign a list value to a symbol in this way:

```
NL > (set ’list1 ’(a b c d))  ==>
(A B C D)
NL > (set ’list2 ’(f g h i j))  ==>
(F G H I J)
NL > (set ’(a b) ’(c d))  ==>
<ERROR "The value:
        (A B)
        of the first argument of the SET-statement:
        (SET (QUOTE (A B)) (QUOTE (C D)))
        is not a symbol">
NL >
```

It does not make sense to assign any value to an object which is not a symbol, for example a list; such a try will generate an error. You can add an element to a list with the `cons` statement:

```
NL > (cons ’e list2)  ==>
(E F G H I J)
NL >
```

Note that this time the symbol `list2` is not quoted; in some sense this means the user is interested by the *value* of the symbol and not by the symbol itself. But there will be a little later a much more precise description of this interpretation; wait a moment. Note also the evaluation of the last statement has not changed the value of the symbol `list2`. If you intend to do so, you must combine the `cons` statement with a `set` statement:

```
NL > list2  ==>
(F G H I J)
NL > (set ’list2 (cons ’e list2))  ==>
(E F G H I J)
NL > list2  ==>
(E F G H I J)
NL >
```

The `set` statement has installed a new binding from the symbol `list2` towards the *result* of the execution of the `cons` statement. Again this mechanism will be detailed in a strictly logical way later. The old binding is lost. You can reach the first element of a list with the `first` statement, and the list without its first element with the `rest` statement. Again you can combine these statements with a `set` statement:

```
NL > (first list2)  ==>
E
NL > (rest list2)  ==>
(F G H I J)
NL > list2  ==>
(E F G H I J)
NL > (set ’list2 (rest list2))  ==>
(F G H I J)
NL > list2  ==>
(F G H I J)
```

Be still careful with the quoting choices.

If you intend to locate the first element of a list by some symbol, please don’t choose the name `first` for this symbol; this would generate an error:

```
NL > (set 'first (first list2))  ==>
<ERROR "The symbol FIRST is a system symbol">
NL > (set 'first-element (first list2))  ==>
F
NL > first-element  ==>
F
```

6 Evaluating a list.

You have already seen several examples of list evaluations, but at this point, it is important not to think you have essentially understood what a list evaluation is. This evaluation process is relatively complex, strictly defined, powerful, but ordinarily confusing for the beginners. This process is quite tricky because it contains in itself the definition of an arbitrary long computation, even possibly infinite! The origin of such a rather strange definition comes from λ-calculus [1] (see also [4] [2]), as modified by McCarthy when he defined the Lisp language [3]. Let us begin with an ordinary list:

```
NL > (a b c d)
<ERROR "The symbol A is unbound">
NL >
```

Evaluation Rule 1 — *Evaluating a list begins with the evaluation of its first element.*

This rule is recursive, so that if the first element of the list is again a list, the first element of the last one will be evaluated, and so on. For instance:

```
NL > (((a b c) d e) f g)  ==>
<ERROR "The symbol A is unbound">
NL >
```

and in such a case the evaluation cannot be processed because the symbol `a` has no value in the current environment. This rule has the following consequence: an empty list cannot be evaluated because not containing a first element.

Evaluation Rule 2 — *The result of the evaluation of the first element of an evaluated list must be a* functional object, *that is, an object whose itype is* lambda, macro, special *or* system.

Let us give a value to the symbol `a` which is not a functional object, for example the symbol `value-for-a` and let us try to evaluate a list beginning with the symbol `a`:

```
NL > (set 'a 'value-for-a)  ==>
VALUE-FOR-A
NL > (a b c d)  ==>
<ERROR "The value:
        VALUE-FOR-A
        of the first component of the list:
        (A B C D)
        is not a functional object">
NL >
```

You see the first rule is satisfied but not the second one, so that an error is generated. Now we can better understand why the various previous examples worked: the second rule was satisfied because the leading symbols in the evaluated lists had a (predefined) value which was a functional object, namely a special-object or a system-object:

```
NL > (first '(a b c d))  ==>
A
NL > first  ==>
<SPECIAL FIRST>
NL > (set 'a 'b)  ==>
B
NL > set  ==>
<SPECIAL SET>
NL > print  ==>
<SYSTEM PRINT>
NL >
```

Now we examine how the predefined functional objects work when they appear as the result of the evaluation of the first element of a list. The other elements of the list are generally used; they are called the *provisional* arguments. These provisional arguments are most often evaluated before being actually used by the functional object; the result of such an evaluation is a *definitive* argument. In the relatively rare cases where an argument is not evaluated before being used, the definitive argument is equal to the provisional one.

Evaluation Rule 3 — *Before actually working, with a few exceptions, a functional object firstly evaluates the* provisional arguments *to compute the* definitive arguments *which will be really used by the functional object. The only exceptions are the following:*

- *The special-objects* if *and* while *have a very special behaviour about this rule (detailed later);*
- *The special-objects* quote, error, lambda, macro, *and the system-objects* step *and* environment *do not evaluate their arguments;*
- *A macro-object does not evaluate its arguments before working; on the contrary it evaluates again the result provided by the macro definition (see the Macro section).*

The situation is a bit like for the conjugation rules in English: these rules are very simple (look, looked, looked) but there are some exceptions (irregular verbs); the point is that these exceptions are very important; in some sense the irregular verbs are the very root of the English language. The situation in Nano-Lisp (and Lisp in general) is essentially the same: the exceptions in the previous evaluation rule are the heart of the Nano-Lisp language.

The easiest exception to be understood concerns the special-object *quote*. This object does not evaluate its unique argument and simply returns it:

```
NL > quote
<SPECIAL QUOTE>
```

```
NL > a-symbol-without-a-value
<ERROR "The symbol A-SYMBOL-WITHOUT-A-VALUE is unbound">
NL > (quote a-symbol-without-a-value)
A-SYMBOL-WITHOUT-A-VALUE
NL > (a b c d)
<ERROR "The value:
        VALUE-FOR-A
        of the first component of the list:
        (A B C D)
        is not a functional object">
NL > (quote (a b c d))
(A B C D)
NL >
```

The special-object *quote* is available to allow you to prevent a non desired evaluation process. The evaluation mechanism between provisional and definitive arguments is so active that frequently it raises some trouble and a tool is necessary to prevent evaluation. The *quote* special-object does this "negative" work. This happens often and it is now traditional in the Lisp dialects, in Nano-Lisp also, to use the *macro-character* "`'`" as an abbreviation. Thus a character string `'something` is considered as an abbreviation of `(quote something)`. For example, the input string:

```
(set 'a 'a-value-for-a)
```

is actually understood by the Lisp reader as the string:

```
(set (quote a) (quote a-value-for-a))
```

If a list is so quoted, the closing parenthesis created by the quote macro-character is just behind the closing parenthesis of the quoted list; so the input string:

```
'(a b c d)
```

is equivalent to the string:

```
(quote (a b c d))
```

and this process is recursive: the following input string:

```
'(a 'b c)
```

is understood by the Lisp reader as the following:

```
(quote (a (quote b) c))
```

Now we can understand all the details of the evaluation process of the list:

```
(set 'a 'value-for-a)
```

There are 15 steps:

1. The Lisp reader translates the input string into:

   ```
   (set (quote a) (quote value-for-a))
   ```

2. Lisp observes the read object is a list.
3. Lisp extracts the first element of the list; it is the symbol `set`.
4. Lisp uses its environment to look for the value of the symbol `set`; it finds the special-object `<SPECIAL SET>`.
5. Lisp knows this special-object needs exactly two arguments, and these arguments must be evaluated before being used; so that...
6. Lisp evaluates the first provisional argument which is the list `(quote a)`.
7. Lisp looks at the first element of this list; it is the symbol `quote`.
8. Lisp evaluates this symbol; in the environment, the value of the symbol `quote` is the special-object `<SPECIAL QUOTE>`.
9. Lisp knows this special-object needs one provisional argument (here the symbol `a`), which is *not* evaluated so that this symbol is also the definitive argument.
10. The `<SPECIAL QUOTE>` functional object returns this definitive argument; the symbol `a` is therefore the result of the evaluation of `(quote a)` which was the first *provisional* argument of the `<SPECIAL SET>` functional object. The *definitive* corresponding argument for `<SPECIAL SET>` is the symbol `a`.
11. In the same way, the second provisional argument of `<SPECIAL SET>` is the list `(quote value-for-a)` and the definitive corresponding argument is the symbol `value-for-a`.
12. Now the definitive arguments for `<SPECIAL-SET>` are available and this functional object can really work. The environment is modified to introduce the new binding {`a` $\Rightarrow$ `value-for-a`}.
13. `<SPECIAL SET>` returns the target of the new installed binding, that is the symbol `value-for-a`.
14. This symbol is therefore the result of the evaluation of the initial list; Lisp displays this result, prompts and waits for the following object to be evaluated.
15. The end.

Such a detailed explanation about the behaviour of the Nano-Lisp interpreter always seems awkward to the beginners. The Lisp evaluation process is extremely powerful but needs a precise knowledge of its definition to be correctly used; there is a mandatory initiatory step in Lisp learning which cannot be avoided; if you have enough will to master this relatively hard technique, you will soon be handling a first-class tool; otherwise you will have to use again any traditional so-called “imperative” language, a slave work.

This first step in Lisp learning is made easier by a tool known as the *stepper*. The stepper is invoked by the `<SYSTEM STEP>` functional object which does not evaluate its argument: the definitive argument is equal to the provisional one. Then the stepper evaluates step-by-step this argument to show you the whole evaluation process. The stepper has many possibilities, but at this point it is sufficient to reply `<RET>` (return) each time the stepper waits; in this way the stepping will be complete:

```
NL > (step (set 'a 'new-value-for-a))  ==>
STEPPING enabled.
<RET> (Return or Enter) -> step this form;
s (skip)                -> no-step-eval this form;
q (quit)                -> quit stepping;
g (global)              -> displays the user global environment;
l (local)               -> displays the local environment.
STGO-2: (SET (QUOTE A) (QUOTE NEW-VALUE-FOR-A))
<RET> s q :  ==>
STGO-3: SET
<RET> s q :  ==>
STBK-3: <SPECIAL SET>
STGO-3: (QUOTE A)
<RET> s q :  ==>
STGO-4: QUOTE
<RET> s q :  ==>
STBK-4: <SPECIAL QUOTE>
STBK-3: A
STGO-3: (QUOTE NEW-VALUE-FOR-A)
<RET> s q :  ==>
STGO-4: QUOTE
<RET> s q :  ==>
STBK-4: <SPECIAL QUOTE>
STBK-3: NEW-VALUE-FOR-A
STBK-2: NEW-VALUE-FOR-A
STEPPING disabled.
NEW-VALUE-FOR-A
NL > (environment g)
G-ENV -> A -> NEW-VALUE-FOR-A
G-ENV -> LIST2 -> (F G H I J)
G-ENV -> LIST1 -> (A B C D)
<SYSTEM ENVIRONMENT>
NL >
```

You have a step-by-step decomposition of the evaluation process. Each line labelled STGO-n means the evaluator is at the evaluation depth n and is going to evaluate the displayed Nano-Lisp object. The initial object `(step ...)` was at the depth 1 but is not displayed because the stepper was not enabled at this moment. The list `(set ...)` is evaluated at the depth 2; each list `(quote ...)` is evaluated at the depth 3; each symbol `quote` is evaluated at the depth 4. You see also the symbol `a` is not evaluated because present inside a quote statement.

On the contrary each line labelled STBK-n displays the result of the evaluation of the corresponding previous STGO-n object. For example the symbol `new-value-for-a` is at `STBK-3` the result of the evaluation of the `STGO-3` object `(quote new-value-for-a)`. On the following line labelled `STBK-2`, the same symbol appears to be the result of the evaluation of the `(set ...)` statement;

in fact, such a statement, after having modified the environment (this is not showed by the stepper), returns the target of the new installed binding. The stepper is then disabled, its work is ended, but however returns the object just obtained, again the same symbol. The `(environment g)` statement allows the user to see the new configuration of the environment stack.

Now let us suppose you intend to look for the first element of the list which is the *value* of the symbol `list2`; it is again interesting to step the process:

```
NL > (step (first list2))  ==>
STEPPING enabled.
<RET> (Return or Enter) -> step this form;
s (skip)                -> no-step-eval this form;
q (quit)                -> quit stepping;
g (global)              -> displays the user global environment;
l (local)               -> displays the local environment.
STGO-2: (FIRST LIST2)
<RET> s q :  ==>
STGO-3: FIRST
<RET> s q :  ==>
STBK-3: <SPECIAL FIRST>
STGO-3: LIST2
<RET> s q :  ==>
STBK-3: (F G H I J)
STBK-2: F
STEPPING disabled.
F
NL >
```

You see the `list2` symbol is evaluated in such a way the definitive argument of `<SPECIAL FIRST>` is the list we are interested by. If the user erroneously quote the `list2` symbol, an error will be generated because the definitive argument will be the symbol itself, but the `<SPECIAL FIRST>` object is able to work only on a list:

```
NL > (step (first ’list2))  ==>
STEPPING enabled.
<RET> (Return or Enter) -> step this form;
s (skip)                -> no-step-eval this form;
q (quit)                -> quit stepping;
g (global)              -> displays the user global environment;
l (local)              -> displays the local environment.
STGO-2: (FIRST (QUOTE LIST2))
<RET> s q :  ==>
STGO-3: FIRST
<RET> s q :  ==>
STBK-3: <SPECIAL FIRST>
STGO-3: (QUOTE LIST2)
<RET> s q :  ==>
STGO-4: QUOTE
<RET> s q :  ==>
STBK-4: <SPECIAL QUOTE>
STBK-3: LIST2
STBK-2: <ERROR "In the FIRST statement:
        (FIRST (QUOTE LIST2))
        the value of the argument:
```

```
        LIST2
        should be a list">
STEPPING disabled.
<ERROR "In the FIRST statement:
        (FIRST (QUOTE LIST2))
        the value of the argument:
        LIST2
        should be a list">
NL >
```

The expression “the value of the argument” should be understood as “the definitive argument”. The first step in the art of Lisp programming is to advisely quote.

7 Creating and using a lambda-object.

A lambda-object is a user-defined functional object. There is a `first` prebound symbol but not a `second` one. The second element of a list can be reached combining the `first` and `rest` symbols:

```
NL > list2  ==>
(F G H I J)
NL > (first (rest list2))  ==>
G
NL >
```

The provisional argument `(rest list2)` is evaluated *before* being used by the `first` function (that is, the `<SPECIAL FIRST>` special-object, this will not be explained anymore), so that the `first` function returns the element which was looked for. Step this evaluation on your machine.

But the user would like to have a `second` function at his disposal. He obtains such a function in this way:

```
NL > (set ’second (lambda (list) (first (rest list))))  ==>
<LAMBDA ((LIST)
         (FIRST (REST LIST)))>
NL > list2  ==>
(F G H I J)
NL > (second list2)  ==>
G
NL > (environment g)  ==>
G-ENV -> SECOND -> <LAMBDA ((LIST)
                           (FIRST (REST LIST)))>
G-ENV -> LIST2 -> (F G H I J)
G-ENV -> LIST1 -> (A B C D)
<SYSTEM ENVIRONMENT>
NL >
```

A `(lambda ...)` statement constructs a lambda-object and returns it. Here and this is frequent, we locate this object by a symbol, the `second` symbol, in order to be able to use the constructed lambda-object later. A lambda statement must have two arguments which are not evaluated during the construction. The first argument is the *parameter list*, which must be a symbol list. Here the parameter list is `(list)`. No system symbol is authorized in a parameter list:

```
NL > (lambda (first) 'hello)  ==>
<ERROR "In the LAMBDA-statement:
        (LAMBDA (FIRST) (QUOTE HELLO))
        there is a system symbol in the parameter-list">
NL >
```

The second argument of a `(lambda ...)` statement is called the *body* of the lambda-object to be constructed. In the erroneous lambda statement just above, the body is the list `(quote hello)`. In the lambda-object previously associated to the `second` symbol, the body is the list `(first (rest list))`.

When the lambda-object is used later, the parameter list is considered by the evaluator; the length of this list must be equal to the argument number of the evaluation asked for. For example here the lambda-object located by the `second` symbol has a parameter list of length *one*, and when it is used in the `(second list2)` statement, there is also *one* argument.

Now we can give the details of the evaluation process of the `(second list2)` statement. Firstly the `second` symbol is evaluated, it is a lambda-object, hence a functional object. Because it is a lambda-object, the evaluator examines the length of the parameter list and verifies equality with the number of provisional arguments; here there is only one such argument, namely the symbol `list2` and the result of the comparison is positive.

Evaluation Rule 4 — *If the functional object which is the value of the first element of an evaluated list is a lambda-object, the length of the parameter list must be equal to the argument number.*

```
NL > (second list1 list2)
<ERROR "The following lambda-object cannot work:
        <LAMBDA ((LIST) (FIRST (REST LIST)))>
        The following list invoked it but has a wrong length:
        (SECOND LIST1 LIST2)">
NL >
```

Evaluation Rule 5 — *If a lambda-object is used in an evaluation process, each provisional argument is evaluated before being used.*

```
NL > (second symbol-without-a-value)  ==>
<ERROR "The symbol SYMBOL-WITHOUT-A-VALUE is unbound">
NL >
```

Here the symbol which is the *provisional* argument has no value so that an error is generated. Compare with:

```
NL > (quote symbol-without-a-value)  ==>
SYMBOL-WITHOUT-A-VALUE
NL >
```

where the `quote` special-object does not evaluate the provisional argument. Once the definitive arguments are calculated by the evaluator, it pushes as many new bindings on the local environment as there are elements in the parameter list; each parameter is bound to the corresponding definitive argument. Thus in our example, the binding {`list` ⇒ `(f g h i j)`} is added to the current local environment, that is the empty local environment. In fact the precise description is the following but the difference with the previous one is subtle and will be detailed only much later.

Evaluation Rule 6 — *When a* lambda-object *is going to work, the local environment* in which the lambda-object was constructed *is augmented with new bindings from the parameters towards the corresponding definitive arguments; this new local environment becomes the current one during the evaluation of the lambda body.*

Evaluation Rule 7 — *After having updated the local environment as explained in the previous rule, the evaluator evaluates the body statement of the lambda-object, saves its value in a secret box, restores the local environment which was present before the activation of the lambda-object and returns the saved value.*

In this section we examine only the simplest consequences of these definitions. There are many others which will be explained only when we will consider the Lisp organization of functional programming, without any equivalent in Pascal-like languages.

Examining an environment can also be done during a stepped evaluation, thanks to the step commands '`g`' (global environment) and '`l`' (local environment). Look at the stepped evaluation of `(second list2)`; we restart from scratch and each time something happens about environments, the good command is used.

```
NL > (environment g)  ==>
<SYSTEM ENVIRONMENT>            ;;; void global-env.
NL > (set 'list2 '(a b c d))  ==>
(A B C D)
NL > (environment g)  ==>
G-ENV -> LIST2 -> (A B C D)     ;;; one binding in global-env.
<SYSTEM ENVIRONMENT>
NL > (set 'second (lambda (l) (first (rest l))))  ==>
<LAMBDA ((L)
         (FIRST (REST L)))>
NL > (environment g)  ==>
G-ENV -> SECOND -> <LAMBDA ((L)     ;;; two bindings
                            (FIRST (REST L)))>
G-ENV -> LIST2 -> (A B C D)
<SYSTEM ENVIRONMENT>
NL > (step (second list2))  ==>
STEPPING enabled.
<RET> (Return or Enter) -> step this form;
s (skip)                -> no-step-eval this form;
q (quit)                -> quit stepping;
g (global)              -> displays the user global environment;
l (local)               -> displays the local environment.
STGO-2: (SECOND LIST2)  ==>
```

```
<RET> s q :l  ==>       ;;; to look at the current local-env which is
<RET> s q :  ==>        ;;; void
STGO-3: SECOND
<RET> s q :  ==>
STBK-3: <LAMBDA ((L)
                 (FIRST (REST L)))>
STGO-3: LIST2  ==>
<RET> s q :l  ==>       ;;; the local-env is still void
<RET> s q :  ==>
STBK-3: (A B C D)
STGO-3: (FIRST (REST L))  ==>
<RET> s q :l  ==>
L-ENV -> L -> (A B C D)  ;;; now it is not void
<RET> s q :  ==>
STGO-4: FIRST
<RET> s q :  ==>
STBK-4: <SPECIAL FIRST>
STGO-4: (REST L)
<RET> s q :  ==>
STGO-5: REST
<RET> s q :  ==>
STBK-5: <SPECIAL REST>
STGO-5: L
<RET> s q :  ==>
STBK-5: (A B C D)
STBK-4: (B C D)
STBK-3: B
STBK-2: B
STEPPING disabled.
B
NL > (environment l)  ==>
<SYSTEM ENVIRONMENT>         ;;; now the local-env is void
NL >
```

Pay attention to the precise moment when the local environment is updated to make the lambda-object work. When the `list2` symbol (provisional argument) *is going* to be evaluated, the local environment is still void. When the definitive argument is computed (the list `(A B C D)`), *then* the local environment is modified; the binding {`l` ⇒ `(A B C D)`} is added, so that just a moment later when the value of the symbol `L` is asked, the evaluator answers the definitive argument. This communication process between the local world of the lambda-object and the exterior world is the usual one.

To be continued…

References

[1] Alonzo Church. *The calculi of lambda-conversion*. Princeton University Press, 1941.

[2] Jean-Louis Krivine. *Lambda-calcul, types et modèles*. Masson, 1990.

[3] J. McCarthy. *A basis for a mathematical theory of computation*. In *Computer programming and formal systems*, North Holland, 1963.

[4] G. Revesz. *Lambda Calculus, combinators and functional programming*. Cambridge University Press, 1988.

-o-o-o-o-o-o-o-