

# Nano-Lisp

## The Reference Manual

*Francis Sergeraert*

### 1 The environment.

--	← 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
{environment ⇒ <SYSTEM ENVIRONMENT>}	← top of the system environment
--	
{itype-itype ⇒ <ITYPE ITYPE>}	← bottom of the system environment

The terminology used in this section is quite essential to understand the Nano-Lisp documentation. We apologize for the great number of *italicized* words or expressions; such items should be more or less considered as *definitions*.

The *environment* is essentially the *memory* of the Nano-Lisp machine. The environment is a *stack* of objects called *bindings*. A binding is a pair {symbol ⇒ Nano-Lisp object}. The symbol is the *source* of the binding and the Nano-Lisp object is the *target*. If some conditions are satisfied, the object is the *value* of the symbol, that is, the object returned (output) by the evaluator when the symbol is input.

The environment is divided in three parts, the *system* environment, the *global* environment and the *local* environment. The system environment is a set of 28 pairs binding *system symbols* to the 28 objects of itype itype, boolean, special and system: 9 itype objects (itype descriptors), 2 boolean-objects (true and false), 14 special objects (predefined essential functional objects) and 3 system-objects (predefined “auxiliary” functional objects). The system environment cannot be modified. Furthermore no binding in the global and local environments can cite a system symbol.

A symbol is *present* in the environment if it is the source of some binding of the environment.

The global and local environments are empty at the beginning of a Nano-Lisp session. The size of the global environment can only increase. The size of the local environment is always empty at top level, changes *during* an evaluation work and is again empty when the interpreter waits at top level.

The global environment is modified by the evaluation of a `(set ...)` statement if the symbol pointed out by the first (definitive) argument is not present in the local environment. If the symbol is even not present in the global environment, a new binding is *added* to it, the target of which is the second definitive argument of the `(set ...)` statement. If the symbol is already present in the global environment, the old target is lost and replaced by the second definitive argument.

Each time a lambda- or macro-object is *invoked* (because being the value of the first element of an evaluated list), the local environment which *was* the current one when the functional object *was* constructed is *restored*. Furthermore the evaluator adds to this local environment new bindings coming from the parameters of the functional object and from the (definitive) corresponding arguments. When the list invoking this functional object is evaluated, the old local environment is restored: the various local environments so look as a stack where a new element is added each time a lambda- or macro-object is invoked. This tricky organization, unknown in usual imperative languages, is at the origin of the functional power of Lisp languages, inherited from  $\lambda$ -calculus.

A lambda-object is constructed by the `lambda` special-object when it appears as the value of the first element of an evaluated list, usually because this list is `(lambda ...)`. A macro-object is constructed by the `macro` special-object when it appears as the value of the first element of an evaluated list, usually because this list is `(macro ...)`. Then the local current environment is saved in the lambda- or macro-object.

A binding of the local environment can possibly be modified by the evaluation of a `(set ...)` statement if this binding is the highest one for the symbol pointed out by the `(set ...)` statement.

The environment can be examined by the `(environment x)` statement where the letter (that is, the symbol) `x` is the letter 's', 'g' or 'l'. Then the system, global or local environment is displayed. This statement returns the system-object `<SYSTEM ENVIRONMENT>` which will therefore be the only displayed object if the corresponding environment is empty.

During a stepped evaluation, the stepper allows the user to ask for the global environment (`g` command) or the local environment (`l` command).

```
NL > (environment l) ==>
<SYSTEM ENVIRONMENT>
NL > (environment g) ==>
<SYSTEM ENVIRONMENT>
NL > (set 'a 'global-value-for-a) ==>
GLOBAL-VALUE-FOR-A
NL > (environment l) ==>
<SYSTEM ENVIRONMENT>
NL > (environment g) ==>
G-ENV -> A -> GLOBAL-VALUE-FOR-A
<SYSTEM ENVIRONMENT>
```

```

NL > ((lambda (a) a) 'local-value-for-a) ==>
LOCAL-VALUE-FOR-A
NL > a ==>
GLOBAL-VALUE-FOR-A
NL > ((lambda (a)
      ((lambda (a) (environment 1))
       'local-value-2-for-a))
      'local-value-1-for-a) ==>
L-ENV -> A -> LOCAL-VALUE-2-FOR-A
L-ENV -> A -> LOCAL-VALUE-1-FOR-A
<SYSTEM ENVIRONMENT>
NL > (step ((lambda (a)
            ((lambda (a) a)
             'local-value-4-for-a))
           'local-value-3-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: ((LAMBDA (A)
          ((LAMBDA (A) A) (QUOTE LOCAL-VALUE-4-FOR-A)))
         (QUOTE LOCAL-VALUE-3-FOR-A))
<RET> s q :l ==>
<RET> s q :g ==>
G-ENV -> A -> GLOBAL-VALUE-FOR-A
<RET> s q : ==>
STGO-3: (LAMBDA (A) ((LAMBDA (A) A) (QUOTE LOCAL-VALUE-4-FOR-A)))
<RET> s q :s ==>
STBK-3: <LAMBDA ((A)
                 ((LAMBDA (A) A) (QUOTE LOCAL-VALUE-4-FOR-A)))>
STGO-3: (QUOTE LOCAL-VALUE-3-FOR-A)
<RET> s q :s ==>
STBK-3: LOCAL-VALUE-3-FOR-A
STGO-3: ((LAMBDA (A) A) (QUOTE LOCAL-VALUE-4-FOR-A))
<RET> s q :l ==>
L-ENV -> A -> LOCAL-VALUE-3-FOR-A
<RET> s q :g ==>
G-ENV -> A -> GLOBAL-VALUE-FOR-A
<RET> s q : ==>
STGO-4: (LAMBDA (A) A)
<RET> s q :s ==>
STBK-4: <LAMBDA ((A) A)>
STGO-4: (QUOTE LOCAL-VALUE-4-FOR-A)
<RET> s q :s ==>
STBK-4: LOCAL-VALUE-4-FOR-A
STGO-4: A ==>
<RET> s q :l ==>
L-ENV -> A -> LOCAL-VALUE-4-FOR-A
L-ENV -> A -> LOCAL-VALUE-3-FOR-A
<RET> s q :g ==>
G-ENV -> A -> GLOBAL-VALUE-FOR-A
<RET> s q : ==>
STBK-4: LOCAL-VALUE-4-FOR-A

```

```

STBK-3: LOCAL-VALUE-4-FOR-A
STBK-2: LOCAL-VALUE-4-FOR-A
STEPPING disabled.
LOCAL-VALUE-4-FOR-A
NL >

```

The following example illustrates a situation where a local environment encapsulated inside a lambda-object is later modified.

```

NL > ((lambda (a)
      (progn (print a)
             (set 'f (lambda () a))
             (set 'a 'new-value-for-a)))
      'first-value-for-a) ==>
PRINT -> FIRST-VALUE-FOR-A
(NEW-VALUE-FOR-A <LAMBDA (NIL A)> FIRST-VALUE-FOR-A)
NL > (f) ==>
NEW-VALUE-FOR-A
L >

```

When the symbol `f` is bound, the value of the symbol `a` is the symbol `first-value-for-a`. A lambda-object is made in which this binding is encapsulated. Later this binding is modified, but this fact is known when this lambda-object is invoked through `f`. This possibility is extended further in the following example where two lambda-objects are created sharing the same environment; the first one modifies it; the second one reads it.

```

NL > ((lambda (a)
      (progn (set 'g (lambda () a))
             (set 'f (lambda () (set 'a (cons 'x a))))))
      '()) ==>
(<LAMBDA (NIL (SET (QUOTE A) (CONS (QUOTE X) A)))>
 <LAMBDA (NIL A)>)
NL > f ==>
<LAMBDA (NIL (SET (QUOTE A) (CONS (QUOTE X) A)))>
NL > g ==>
<LAMBDA (NIL A)>
NL > (g) ==>
()
NL > (f) ==>
(X)
NL > (f) ==>
(X X)
NL > (g) ==>
(X X)
NL >

```

## **2 Symbols.**

### **2.1 Itype system symbols.**

### **2.2 Boolean system symbols.**

### **2.3 Special system symbols.**

### **2.4 System system symbols.**

## **3 Nano-Lisp objects.**

### **3.1 Nano-Lisp predefined objects.**

### **3.2 Error-objects.**

### **3.3 Lists.**

### **3.4 Lambda- and macro-objects**

## **4 The special-objects.**

There are 14 special-objects, each one located by a symbol of the system environment. Such a binding cannot be modified; no binding for such a symbol can be installed in the global and local environment.

The list of system symbols bound to special objects is the following; a rough description of its use is given for each one; the precise and complete description is the subject of the corresponding subsection.

- `progn`: compound statement;
- `if`: conditional statement;
- `while`: iterative statement;
- `quote`: prevent object evaluation;
- `eval`: double object evaluation;
- `itype`: determine an implementation type;
- `error`: error statement;
- `set`: modify the environment;
- `lambda`: create a functional object;

- `macro`: create a macro object;
- `equal`: compare two objects;
- `cons`: add an element to a list;
- `first`: extract the first element of a list;
- `rest`: extract the first sublist of a list.

A special-object has trivial evaluation. Usually a special-object is used when the evaluation of the first element of a list returns it; then its functional ability is used. In the last situation, the special-object uses arguments which are the other objects of the list the evaluation of which invoked the special object. According to the special object, the arguments of a special object are evaluated or not before being used.

```
NL > progn ==>
<SPECIAL PROGN>
NL > (eval progn) ==>
<SPECIAL PROGN>
NL > (eval (eval progn)) ==>
<SPECIAL PROGN>
NL > (set 'f progn) ==>
<SPECIAL PROGN>
NL > (f 'a 'b 'c) ==>
(C B A)
NL > (set 'f quote)
<SPECIAL QUOTE>
NL > (f a b c)
<ERROR "In the QUOTE-statement:
      (F A B C)
      the argument number should be 1">
NL > (f a)
A
NL > (set 'f 'quote)
QUOTE
NL > (f a)
<ERROR "The value:
      QUOTE
      of the first component of the list:
      (F A)
      is not a functional object">
NL >
```

## 4.1 The PROGN special-object.

The `progn` special-object uses any number of arguments. Each one is evaluated in turn and each result is pushed in a result list which is finally returned by the `progn` special-object. If the evaluation of an argument returns an error-object, then the `progn` special-object returns immediately this error-object; in such a case the result list is lost.

```
NL > progn ==>
```

```

<SPECIAL PROGN>
NL > (itype 'progn) ==>
<ITYPE SYMBOL>
NL > (itype progn) ==>
<ITYPE SPECIAL>
NL > (progn 'a 'b 'c) ==>
(C B A)
NL > (progn) ==>
()
NL > (progn (print 'a) (error hello) (print 'b)) ==>
PRINT -> A
<ERROR "User Error -> HELLO">
NL >

```

The results of argument evaluation appear to be in reverse order. Frequently the user is in fact interested by the last result which therefore can be reached using the first special-object.

```

NL > (first (progn (set 'l '(boys)) (set 'element 'hello)
                 (set 'l (cons element l)))) ==>
(HELLO BOYS)
NL > (first (progn)) ==>
<ERROR "In the FIRST statement:
      (FIRST (PROGN))
      the value of the argument is an empty list">
NL >

```

COMMON-LISP NOTE: The Nano-Lisp `progn` is slightly different from the Common-Lisp one. On one hand the Common-Lisp `progn` returns only the result of the evaluation of the last argument; on the contrary the Nano-Lisp `progn` returns all the results pushed in a list, therefore in reverse order. The Nano-Lisp `progn` can logically work without argument and then (logically) returns the empty list. The Common-Lisp `progn` works in the same way in this case, but because of the definition of the Common-Lisp `progn`, this is not really coherent! This can be important in a macro-expansion. The Nano-Lisp `(progn ...)` is roughly equivalent to the Common-Lisp `(reverse (list ...))`. The Common-Lisp `(progn ...)` is roughly equivalent to the Nano-Lisp `(first (progn ...))`.

## 4.2 The IF special-object.

The `if` special-object needs three arguments. The first one is called the *condition*, the second one is the *positive branch*, the third one is the *negative branch*.

The `if` special-object firstly evaluates the condition argument. The result must be a boolean-object, otherwise an error-object is immediately returned.

If the result of the condition is the `true` boolean, the positive branch is evaluated, and the object returned by this statement is returned by the `if` special-object; the negative branch is not evaluated.

If the result of the condition is the `false` boolean, the negative branch is evaluated, and the object returned by this statement is returned by the `if` special-object; the positive branch is not evaluated.

```

NL > if ==>
<SPECIAL IF>
NL > (itype 'if) ==>
<ITYPE SYMBOL>
NL > (itype if) ==>
<ITYPE SPECIAL>
NL > (if 'a 'b) ==>
<ERROR "In the IF-statement:
      (IF (QUOTE A) (QUOTE B))
      the argument number should be 3">
NL > (if 'a 'b 'c)
<ERROR "In the IF-statement:
      (IF (QUOTE A) (QUOTE B) (QUOTE C))
      evaluating the condition produced
      A
      which is not a boolean">
NL > (if (equal 'a 'a) (print 'yes) (print 'no)) ==>
PRINT -> YES
YES
NL > (if (equal 'a 'b) (print 'yes) (print 'no)) ==>
PRINT -> NO
NO
NL >

```

If the condition returns an error-object, no branch is evaluated and the error-object is immediately returned.

```

NL > (if (error hello) (print 'a) (print 'b)) ==>
<ERROR "User Error -> HELLO">
NL >

```

### 4.3 The WHILE special-object.

The while special-object needs two arguments. The first one is called the *condition*, the second one is the *body*.

The while special object initializes an empty result list and repeats the following process until it is interrupted.

The condition is evaluated. If the result is an error-object, the while process terminates and this error-object is returned; the result list is lost. If the result is not a boolean-object, the while process terminates and an error-object is immediately returned; the result list is lost. If the result is the false boolean, the while process terminates and returns the result list. If the result is the true boolean, the while process continues.

If so, the body is evaluated. If the result is an error-object, the while process terminates, this error-object is immediately returned and the result list is lost. Otherwise the result is pushed in the result list, the condition is evaluated, and so on.

```

NL > while ==>
<SPECIAL WHILE>
NL > (itype 'while) ==>
<ITYPE SYMBOL>
NL > (itype while) ==>

```



```

<ITYPE SPECIAL>
NL > (while 'a 'b 'c) ==>
<ERROR "In the WHILE-statement:
      (WHILE (QUOTE A) (QUOTE B) (QUOTE C))
      the argument number should be 2">
NL > (while (error hello) a) ==>
<ERROR "User Error -> HELLO">
NL > (while 'a 'b)
<ERROR "In the WHILE-statement:
      (WHILE (QUOTE A) (QUOTE B))
      evaluating the condition produced:
      A
      which is not a boolean">
NL > (while true (error hello)) ==>
<ERROR "User Error -> HELLO">
NL > (set 'l '(a b c d)) ==>
(A B C D)
NL > (while (if (equal l '()) false true)
      (set 'l (rest l))) ==>
(() (D) (C D) (B C D))
NL > (while (if (equal l '()) false true)
      (set 'l (rst l))) ==>
()
NL >

```

The results of successive body evaluations appear to be in reverse order. More readable statements can be written if a not function is defined:

```

NL > (set 'not (lambda (boolean) (if boolean false true))) ==>
<LAMBDA ((BOOLEAN)
      (IF BOOLEAN FALSE TRUE))>
NL > (set 'l '(a b c d)) ==>
(A B C D)
NL > (while (not (equal l '())) ==>
      (set 'l (rest l)))
(() (D) (C D) (B C D))
NL > (while (not (equal l '())) ==>
      (set 'l (rest l)))
()
NL >

```

#### 4.4 The QUOTE special-object.

The quote special-object needs one argument which is not evaluated and is immediately returned. This special-object is used to prevent evaluation. The list (quote object) can be abbreviated as 'object.

```

NL > quote ==>
<SPECIAL QUOTE>
NL > (itype 'quote) ==>
<ITYPE SYMBOL>
NL > (itype quote) ==>
<ITYPE SPECIAL>
NL > a ==>

```

```

<ERROR "The symbol A is unbound">
NL > (quote a) ==>
A
NL > 'a ==>
A
NL > (quote a b) ==>
<ERROR "In the QUOTE-statement:
      (QUOTE A B)
      the argument number should be 1">
NL >

```

## 4.5 The EVAL special-object.

The `eval` special-object needs one argument. The provisional argument is evaluated and the result is the definitive argument. This special object again evaluates the definitive argument and returns the result. So that the `eval` special-object is essentially a double evaluator.

```

NL > eval ==>
<SPECIAL EVAL>
NL > (itype 'eval) ==>
<ITYPE SYMBOL>
NL > (itype eval) ==>
<ITYPE SPECIAL>
NL > (set 'a 'b) ==>
B
NL > (set 'b 'c) ==>
C
NL > (eval ''a) ==>
(QUOTE A)
NL > (eval ''a) ==>
A
NL > (eval 'a) ==>
B
NL > (eval a) ==>
C
NL > (eval (eval a)) ==>
<ERROR "The symbol C is unbound">
NL > (eval (eval (eval ''a))) ==>
B
NL > (eval a b) ==>
<ERROR "In the EVAL-statement:
      (EVAL A B)
      the argument number should be 1">
NL >

```

In fact the `eval` special-object is not completely defined in the previous explanations. There is a subtle gap about the environment which must be used to make the evaluator work. And here is an important difference between Nano-Lisp and Common-Lisp. In Nano-Lisp, if the evaluator works for the `eval` special-object, it keeps the global *and local* environments. On the contrary, in Common-Lisp, the evaluator evaluates the definitive argument in the current global environment but the null local environment. This important question will be studied in minute detail in the section “From

Nano-Lisp to Common-Lisp". If this is not perfectly understood, subtle bugs will be encountered in complicated programs using all the functional abilities of Lisp.

The simplest statement where the difference appears is the following where a global binding and a local one are defined for the symbol `b` when the `eval` special object works.

```
NL > (set 'a 'b) ==>          ;; Nano-Lisp experience.
B
NL > (set 'b 'a-global-value-for-b) ==>
A-GLOBAL-VALUE-FOR-B
NL > (eval a) ==>
A-GLOBAL-VALUE-FOR-B
NL > ((lambda (b) (eval a)) 'a-local-value-for-b) ==>
A-LOCAL-VALUE-FOR-B          ;; The local value of b is returned.
NL >
```

An analogous experience with *Common-Lisp*:

```
> (set 'a 'b) ==>
B
> (set 'b 'a-global-value-for-b) ==>
A-GLOBAL-VALUE-FOR-B
> (eval a) ==>
A-GLOBAL-VALUE-FOR-B
> ((lambda (b) (eval a)) 'a-local-value-for-b) ==>
A-GLOBAL-VALUE-FOR-B          ;; The global value of b is returned.
>
```

## 4.6 The `ITYPE` special-object.

The `itype` special-object needs one argument. The provisional argument is evaluated and the result is the definitive argument. If the definitive argument is an error-object, this object is returned. Otherwise the `itype` special-object returns the implementation type of the definitive argument. There are exactly nine possible itypes: `itype`, `error`, `boolean`, `symbol`, `list`, `lambda`, `macro`, `special` and `system`.

```
NL > itype ==>
<SPECIAL ITYPE>
NL > (itype 'itype) ==>
<ITYPE SYMBOL>
NL > (itype itype) ==>
<ITYPE SPECIAL>
NL > (itype a b) ==>
<ERROR "In the ITYPE-statement:
      (ITYPE A B)
      the argument number should be 1">
NL > (itype a)
<ERROR "The symbol A is unbound">
NL > (itype 'a) ==>
<ITYPE SYMBOL>
NL > (itype ''a) ==>
<ITYPE LIST>
NL >
```

## 4.7 The ERROR special-object.

The error special-object allows the user to create and return an error-object containing a simple error message. This special-object needs one argument which is not evaluated. The external form of the error-object is:

```
<ERROR "User Error -> x-x-x">
```

where x-x-x is the external form of the (non-evaluated) argument:

```
NL > error ==>
<SPECIAL ERROR>
NL > (itype 'error) ==>
<ITYPE SYMBOL>
NL > (itype error) ==>
<ITYPE SPECIAL>
NL > (error (example of error message)) ==>
<ERROR "User Error -> (EXAMPLE OF ERROR MESSAGE)">
NL >
```

If at any level of an evaluation process an error object is created, this object acts as a black hole: no more computation is done, and the error message is returned to the top-level and displayed:

```
NL > (progn
      (if (progn
            (while (print (error (a deep error)))
                  doesnt-matter))
            hi
            bye)) ==>
<ERROR "User Error -> (A DEEP ERROR)">
NL >
```

## 4.8 The SET special-object.

The set special-object allows the user to modify the current environment. It needs two (provisional) arguments which are evaluated and become the definitive arguments. The first definitive argument must be a symbol, otherwise an error object is returned. If the symbol is a system symbol, an error-object is also returned. The second definitive argument can be any Nano-Lisp object. If this object is an error-object, the environment is not modified and the error-object is returned. Otherwise a binding is installed from the symbol which is the first definitive argument towards the object which is the second one. If a binding for this symbol already exists, the highest one is modified; otherwise a new binding is added on the *global* environment.

```
NL > set ==>
<SPECIAL SET>
NL > (itype 'set) ==>
<ITYPE SYMBOL>
NL > (itype set) ==>
<ITYPE SPECIAL>
NL > (set a b c) ==>
```

```

<ERROR "In the SET-statement:
      (SET A B C)
      the argument number should be 2">
NL > (set 'a b) ==>
<ERROR "The value:
      (QUOTE A)
      of the first argument of the SET-statement:
      (SET (QUOTE (QUOTE A)) B)
      is not a symbol">
NL > (set 'set a) ==>
<ERROR "In the SET-statement:
      (SET (QUOTE SET) A)
      the symbol SET is a system symbol">
NL > (set 'a b) ==>
<ERROR "The symbol B is unbound">
NL > (set 'a 'b) ==>
B
NL > a ==>
B
NL >

```

If no previous binding is defined for the symbol, a new one is created and added on the global environment:

```

NL > a ==>
<ERROR "The symbol A is unbound">
NL > (set 'a 'value-for-a) ==>
VALUE-FOR-A
NL > a ==>
VALUE-FOR-A
NL >

```

If a binding already exists, the highest one is modified and the corresponding old value is lost:

```

NL > a ==>
VALUE-FOR-A
NL > (set 'a 'new-value-for-a) ==>
NEW-VALUE-FOR-A
NL > a ==>
NEW-VALUE-FOR-A
NL >

```

If two (or more) bindings are already defined, only the highest one is modified. An example where a global binding and a local one are already defined:

```

NL > (set 'a 'global-value-for-a) ==>
GLOBAL-VALUE-FOR-A
NL > ((lambda (a)
      (progn
        (environment 1)
        (environment g)
        (set a 'new-local-value-for-a)
        (environment 1)
        (environment g)))

```

```

      'local-value-for-a) ==>
L-ENV -> A -> LOCAL-VALUE-FOR-A
G-ENV -> A -> GLOBAL-VALUE-FOR-A
L-ENV -> A -> NEW-LOCAL-VALUE-FOR-A
G-ENV -> A -> GLOBAL-VALUE-FOR-A
(<SYSTEM ENVIRONMENT> <SYSTEM ENVIRONMENT> NEW-LOCAL-VALUE-FOR-A
 <SYSTEM ENVIRONMENT> <SYSTEM ENVIRONMENT>)
NL >

```

This is again an important difference with Common-Lisp where the `set` function modifies only the *global* environment:

```

> (set 'a 'global-value-for-a) ==>      ;;; Common-Lisp experience.
GLOBAL-VALUE-FOR-A
> a ==>
GLOBAL-VALUE-FOR-A
> ((lambda (a)
    (progn
      (print a)
      (set 'a 'new-local-value-for-a-???)
      (print a)))
  'local-value-for-a) ==>
LOCAL-VALUE-FOR-A   ;;; first print
LOCAL-VALUE-FOR-A   ;;; second print
LOCAL-VALUE-FOR-A   ;;; returned by lambda-progn-print
> a ==>
NEW-LOCAL-VALUE-FOR-A-???   ;;; the global environment is modified.
>

```

Finally let us verify that a new *so created* binding is installed on the global environment and not on the local one:

```

NL > a ==>
<ERROR "The symbol A is unbound">
NL > ((lambda (b)
    (progn
      (set 'a 'value-for-a)
      (environment l)
      (environment g)))
  'value-for-b) ==>
L-ENV -> B -> VALUE-FOR-B
G-ENV -> A -> VALUE-FOR-A
(<SYSTEM ENVIRONMENT> <SYSTEM ENVIRONMENT> VALUE-FOR-A)
NL > a ==>
VALUE-FOR-A
NL >

```

A consequence of these definitions is the fact that a global binding can be *created* only by a `set` special-object. On the contrary, a local binding can be created only when the evaluator invokes a lambda-object and makes it work on some arguments, but such a binding can be modified by the `set` special-object. Another consequence is that there is at most one global binding for a symbol; on the contrary several (stacked) different local bindings for a symbol are possible:

```

NL > (set 'a 'global-value-for-a) ==>

```

```

GLOBAL-VALUE-FOR-A
NL > ((lambda (a)
      ((lambda (a)
         (progn
           (environment g)
           (environment l)
           (print a))))
      'local-value-2-for-a))
      'local-value-1-for-a) ==>
G-ENV -> A -> GLOBAL-VALUE-FOR-A
L-ENV -> A -> LOCAL-VALUE-2-FOR-A
L-ENV -> A -> LOCAL-VALUE-1-FOR-A
PRINT -> LOCAL-VALUE-2-FOR-A
      (LOCAL-VALUE-2-FOR-A <SYSTEM ENVIRONMENT> <SYSTEM ENVIRONMENT>)
NL >

```

## 4.9 The LAMBDA special-object.

The lambda special-object allows the user to construct “user-defined” functional objects. The lambda special-object needs two arguments which are not evaluated. The first argument is the parameter-list which must be a list of different non-system symbols. The second argument is the body of the lambda statement; any Nano-Lisp object can be the body.

The lambda special-object constructs a lambda-object which keeps three informations:

1. The parameter-list;
2. The body;
3. The local environment.

The informations 1 and 2 are displayed in the external form of the lambda-object. The information 3 is not displayed but plays an essential role in functional programming.

```

NL > lambda ==>
<SPECIAL LAMBDA>
NL > (itype 'lambda) ==>
<ITYPE SYMBOL>
NL > (itype lambda) ==>
<ITYPE SPECIAL>
NL > (lambda 'a) ==>
<ERROR "In the LAMBDA-statement:
      (LAMBDA (QUOTE A))
      the argument number should be 2">
NL > (lambda a hello) ==>
<ERROR "In the lambda-statement:
      (LAMBDA A HELLO)
      the parameter-list should be a list">
NL > (lambda ((a)) hello) ==>
<ERROR "In the lambda-statement:

```

```

      (LAMBDA ((A)) HELLO)
      the parameter-list should be made of symbols">
NL > (lambda (first) hello) ==>
<ERROR "In the LAMBDA-statement:
      (LAMBDA (FIRST) HELLO)
      there is a system symbol in the parameter-list">
NL > (lambda (a b a) hello) ==>
<ERROR "In the LAMBDA-statement:
      (LAMBDA (A B A) HELLO)
      two symbols are equal in the parameter-list">
NL > (set 'second (lambda (list) (first (rest list)))) ==>
<LAMBDA ((LIST) (FIRST (REST LIST)))>
NL > (second '(a b c d)) ==>
B
NL >

```

A simple experience showing the local environment is kept in the created lambda-object is the following:

```

NL > (set 'appender
      (lambda (element)
        (lambda (list) ;; this lambda statement will create a
          (cons element list)))) ==> ;; lambda-object
<LAMBDA ((ELEMENT) ;; which will keep the binding (element -> ?)
      (LAMBDA (LIST) (CONS ELEMENT LIST)))>
NL > (set 'a-appender (appender 'a)) ==>
<LAMBDA ((LIST) ;; The binding (element -> a) is not displayed,
      (CONS ELEMENT LIST))>
NL > (a-appender '(b c)) ==> ;; but kept INSIDE the lambda-object
(A B C) ;; and LATER used.
NL > (step (a-appender '(b c))) ==>
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: (A-APPENDER (QUOTE (B C)))
<RET> s q : ==>
STGO-3: A-APPENDER
<RET> s q : ==>
STBK-3: <LAMBDA ((LIST)
      (CONS ELEMENT LIST))>
STGO-3: (QUOTE (B C)) ;; provisional argument
<RET> s q :s ==>
STBK-3: (B C) ;; definitive argument
STGO-3: (CONS ELEMENT LIST)
<RET> s q :l ==>
L-ENV -> LIST -> (B C) ;; local environment when the
L-ENV -> ELEMENT -> A ;; lambda-object is invoked
<RET> s q :q ==>
STEPPING disabled
STBK-2: (A B C)
(A B C)
NL > element ==>

```



```

<ERROR "The symbol ELEMENT is unbound">
NL > (a-appender '(bb cc)) ==>
(A BB CC)
NL >

```

The following example shows a case where the body of a lambda-object is not an input-object.

```

NL > (set 'contorted
      (eval
       (progn
        (progn (progn itype-itype false lambda)
              quote)
        '()
        lambda))) ==>
<LAMBDA
  (NIL (<SPECIAL QUOTE> (<SPECIAL LAMBDA> <FALSE> <ITYPE ITYPE>)))>
NL > (contorted)
(<SPECIAL LAMBDA> <FALSE> <ITYPE ITYPE>)
NL >

```

Maybe it is a good exercise to explain this small session, but it is certainly not a good programming style.

## 4.10 The MACRO special-object.

## 4.11 The EQUAL special-object.

The equal special-object needs two arguments which are evaluated before being used. Then they are compared.

```

NL > equal ==>
<SPECIAL EQUAL>
NL > (itype 'equal) ==>
<ITYPE SYMBOL>
NL > (itype equal) ==>
<ITYPE SPECIAL>
NL > (equal 'a 'b 'c) ==>
<ERROR "In the EQUAL-statement:
      (EQUAL (QUOTE A) (QUOTE B) (QUOTE C))
      the argument number should be 2">
NL > (equal a b) ==>
<ERROR "The symbol A is unbound">
NL > (equal 'a 'b) ==>
<FALSE>
NL > (equal 'a 'a) ==>
<TRUE>
NL > (set 'l '(a b c d)) ==>
(A B C D)
NL > (set 'not (lambda (boolean) (if boolean false true))) ==>
<LAMBDA ((BOOLEAN)
      (IF BOOLEAN FALSE TRUE))>
NL > (first (progn (while (not (equal l '()))

```

```

                (set 'l (rest (print l))))
            'the-end)) ==>
PRINT -> (A B C D)
PRINT -> (B C D)
PRINT -> (C D)
PRINT -> (D)
THE-END
NL >

```

It is not possible in Nano-Lisp to modify an existing list, so that there is no matter for the presence of two different comparison functions like in Common-Lisp (`eq` and `equal`) according to the desired comparison (machine address or shape); in particular it is not possible to construct a circular list; see the *Nano-Lisp to Common-Lisp Handbook*.

## 4.12 The CONS special-object.

The `cons` special-object needs two arguments which are evaluated before being used; the second definitive argument must be a list; the `cons` special object returns a list where the first definitive argument is added in front of the second one. If a symbol was bound to the list, this binding is not modified. If the list was a part of another list, this list is not modified.

```

NL > cons ==>
<SPECIAL CONS>
NL > (itype 'cons) ==>
<ITYPE SYMBOL>
NL > (itype cons) ==>
<ITYPE SPECIAL>
NL > (cons a b c) ==>
<ERROR "In the CONS-statement:
      (CONS A B C)
      the argument number should be 2">
NL > (cons a b) ==>
<ERROR "The symbol A is unbound">
NL > (cons 'a 'b) ==>
<ERROR "In the CONS statement:
      (CONS (QUOTE A) (QUOTE B))
      the value of the second argument:
      B
      should be a list">
NL > (cons 'a '(b c)) ==>
(A B C)
NL > (cons 'a '()) ==>
(A)
NL > (set 'l '(b c d)) ==>
(B C D)
NL > (cons 'a l) ==>
(A B C D)
NL > l ==>
(B C D)
NL > (cons 'a (rest l)) ==>
(A C D)
NL > l ==>

```

```
(B C D)
NL >
```

### 4.13 The FIRST special-object.

The `first` special-object needs one argument which is evaluated before being used; the definitive argument must be a non-empty list whose first element is returned. The argument list is not modified.

```
NL > first ==>
<SPECIAL FIRST>
NL > (itype 'first) ==>
<ITYPE SYMBOL>
NL > (itype first) ==>
<ITYPE SPECIAL>
NL > (first a b) ==>
<ERROR "In the FIRST-statement:
      (FIRST A B)
      the argument number should be 1">
NL > (first a)
<ERROR "The symbol A is unbound">
NL > (first 'a) ==>
<ERROR "In the FIRST statement:
      (FIRST (QUOTE A))
      the value of the argument:
      A
      should be a list">
NL > (first '()) ==>
<ERROR "In the FIRST statement:
      (FIRST (QUOTE ()))
      the value of the argument is an empty list">
NL > (set 'l '(a b c d)) ==>
(A B C D)
NL > (first l) ==>
A
NL > l ==>
(A B C D)
NL >
```

### 4.14 The REST special-object.

The `rest` special-object needs one argument which is evaluated before being used; the definitive argument must be a non-empty list whose tail (that is, the same list where the first element is removed) is returned. The argument list is not modified.

```
NL > rest ==>
<SPECIAL REST>
NL > (itype 'rest) ==>
<ITYPE SYMBOL>
NL > (itype rest) ==>
<ITYPE SPECIAL>
NL > (rest a b) ==>
<ERROR "In the REST-statement:
```

```

      (REST A B)
      the argument number should be 1">
NL > (rest a)
<ERROR "The symbol A is unbound">
NL > (rest 'a) ==>
<ERROR "In the REST statement:
      (REST (QUOTE A))
      the value of the argument:
      A
      should be a list">
NL > (rest '()) ==>
<ERROR "In the REST statement:
      (REST (QUOTE ()))
      the value of the argument is an empty list">
NL > (set 'l '(a b c d)) ==>
(A B C D)
NL > (rest l) ==>
(B C D)
NL > l ==>
(A B C D)
NL >

```

## **5 LIST Evaluation.**

## **6 The Nano-Lisp EVALUATOR.**

## **7 The Nano-Lisp READER.**

## **8 The Nano-Lisp PRINTER.**

-0-0-0-0-0-0-0-