Help

Version 1.0 (January 91)

A Scheme-like Lazy Lisp dialect

e-mail: schiex@cert.fr, schiex@irit.fr

Table of Contents

This software is actually freeware, but any donation (under any useful form), e-mail, s-mail or encouragement, is greatly appreciated by the author. I do not want to work (during week-ends) on something that is left unused except by four or five people. Thanks !

For queries and bug reports write or e-mail to the author:

Thomas SCHIEX
Centre d'Etudes et de Recherche de Toulouse (ONERA)
2, Av Edouard Belin
BP 4025
31055 TOULOUSE CEDEX
FRANCE
e-mail: schiex@cert.fr

No responsibilities accepted for bugs, but please let me know so I may try to fix them.

For your information, Help is mainly written in 680x0 assembly language. Porting it to C, or any "portable" language is not even considered (at this time).

The archive file is organized as follows:

Paresseux: The HELP interpreter. The needed memory may be changed via the Finder "Get Info".
Docs: as the name says…
Examples: as the name says… One may find various theorical examples from lambda-calculus, a constraint solver ("test and generate", no forward checking or arc-consistency)…
Divers: (miscellaneous) contains 1) resedit templates ressouces to insert in your Resedit program to ease Help ressources editing, 2) the symbolic code generated by the Help compiler for fibonacci (fib), 3) its conversion in real assembly lang. (foreign.a), 4) the language machine code (Foreign) and 5) a file that load and links everything (Fib-init). 6) A simple stepper for Help is in the "Stepper" file (load it from help and use step syntactic form).
Compilo: a compiler kernel for Help (compiles to 680x0 code, symbolic syntax). Probably not understandable by the vulgum pecus, but given as an example…
Readme: a simple readme file (TEXT)
start: a file needed by Help, loaded upon booting. Must stay in the Help folder.

Have fun !

5 The manual

Let us recall first that Help is, before anything else, a Lisp interpreter. Anybody having a good knowledge of Lisp, or even better Scheme ,may in a first step, immediately try the binaries, but beware of lazyness ! If you want to use Help and change its configuration, consider reading §5.5.

5.1 An overview

5.1.1 Semantics

- ▼ Help is a statically[1] scoped language (identifier scope is lexical) as are Scheme, Algol… Each identifier occurrence is associated with a lexically visible binding of this identifier.
- ▼ Help is non-sctrict, relying on call by need (also referred as "lazyness") for **every** parameter passing as in Lazy Miranda, Hope…
- ▼ Help is dynamically typed (types are latent), i.e. types are associated with values not variables. This is usual in Lisp, APL, Snobol…
- ▼ Help closures (or procedures) are full citizens that may be dynamically created, gathered in any data structure (eventually infinite data structures)… Help shares this quality with every "functionnal" language (such as Hope, Miranda, Daisy…).
- ▼ Help objects (closures, evironments, numbers) have unlimited extent. The memory management software will simply collect any object that may not be referenced by Help user (usually referred as the "GC" or "Garbage Collector"). The same type of system is used in every Lisp, APL, Prolog . It has been proposed for Ada, but no implementation i know has included it.
- ▼ Help does not supports iteration, only recursion (lazyness and iteration do not originate from the same world). Therefore, the interpreter efficiently handles terminal recursions. This allows these recursions to operate with a fixed stack size consumption.

5.1.2 Syntax

- ▼ Help uses parenthesized for describing code and data. This syntax, whose simplicity is one of the main advantage is used (with some exceptions) by every Lisp dialect[2].

5.1.3 Notation and terminology

When Help semantics is yet undefined, we will consider that the value returned is an "undefined" value. In this case, you should not rely on the peculiar value any implementation may return. You should notre that an unspecified value is different for the "unspecified error" which is a precisely specified value denoted by the symbol "`?`".

In the following, the examples will always be presented in following typewriter style: `exemples`. The symbol "➟" used in these examples should be read as "whose value is printed as…". Because of Help lazyness, there is an big difference between the internal representation (holding "suspended values") and the external representation. This distinction is important. Infinite objects

[1]Du moins, c'est la cas dans le cadre d'une utilisation normale. Il est possible d'élargir la portée d'une variable (Cf § 3.5.1.1).

[2]Les premiers évaluateurs Lisp utilisaient une syntaxe différente, distinguant programmes et données (M Expressions opposées aux S Expressions actuelles).

are usually limited by the interpreter's printer. Dots (...) are used to denote the fact that the data structure is not completly printed. Example:

```
(letrec [(x (cons 1 x))] x) ➟ (1 1 1 1 1 1 1 …)
```

The § 5.3 and 5.4 are composed of a sequence of definition. Each definition presents a closure or a syntactic form of the language and begins by a header specifying a model (how is the closure/syntactic form used), its type (the closures are separated in 2 types: `ProcN` whose arity is ficed, and `NProc` whose arity is not fixed), its arity (minimal arity for `NProc`), its arguments types (*`arguments`* style). If there is no restriction on the arguments type, the word *`any`* shall be used. The following types will be considered:

number --------------------------	fixsize integer, "bignum" or floating point
integer -------------------------	fixsize integer, "bignum"
fix -------------------------------	fixsize integer
posfix ---------------------------	positive fixsize integer
bignum --------------------------	"bignum"
floating ------------------------	floating point
smallnum -----------------------	a number that is not a bignum
list ------------------------------	conses or () (the empty list)
cons -----------------------------	conses
vector ---------------------------	vectors, also called "cells"
environ-------------------------	environment
applicable---------------------	closure or *fixpos*
symbol --------------------------	symbol, error, constant,keyword...
ident----------------------------	variable or constant identifier
identv---------------------------	variable identifier
keyword ------------------------	syntactic keyword
error ----------------------------	error
bit-array ----------------------	bit-array
io-unit-------------------------	input/output unit
any ------------------------------	anything you may want

Example:

(**`cell=?`** *`vector posfix`*)	Closure:`ProcN`	2

says that **`cell=?`** is a fixed arity closure (arity is equal to 2), whose arguments are a vector and a number which should be a fixsize positive integer.

5.2 Readable and non readable objects

At any time, a great deal of different objects exist in the memory (the heap). Most of these objects, whose external rpresentation is of matter to the user, are also readable (i.e., it is possible to give a syntactic description of the object that may be given to the Help "reader". This reader will then create an internal representation of it). For pragmatic or feasability reasons, some objects have no external representations or eventually, have an external representation which is not readable.

The reader makes no distinction between upper or lower cases (except inside strings and for symbols interned with the“backslash” macro-char). So, the strings `Foo`, `fOO` denote a single symbol. The space character (ASCII 32) eand carriage return(ASCII 13) are delimiters.

In the following, we will distinguish characters sequences (characters separated by delimiters (Cf § 5.2.7)) and strings (an object type in Help).

5.2.1 The numbers (fixnums, bignums and floatings)

Rationnals and complexs are not yet available in Help. Moreover, there are strong restrictions on bignums use.

5.2.1.1 The integers

An integer is always read in the current base. This base is fixed with the closure `ibase` described later. The character sequences that are interpreted as numbers depend on this base. If the base is lower or equal to 10 (in decimal), the character between “`0`” and the base less one will be considered as digits. If the base is greater than 10 (It is limited to 36 internally), digits are extended using “`a`”...”`z`”. So, in hexa, The sequence “`ff`” represents an integer (not a symbol) whose value is 255 (decimal).

Let `CarNum`, be the set of characters representing digits.

```
number::=               number_unsigned          |
                        +number_unsigned         |
                        -number_unsigned
number_unsigned::=      CarNum*
```

Examples: `123 +123 -123 +235987459862345` (base 10 at least)
`12af` (base 16) -`foobar` (base 29)

5.2.1.1.1 Fixsize integers

When there is an internal representation as a “fixsize integer”, the reader will automatically use it. The value of a fixsize integer is between -2 147 483 648 and 2 147 483 647.

5.2.1.1.2 Bignums

Every integer that may not be internally represented as a fixsize integer will be read as a “bignum”. Pay attention that “bignums” handling uses much stack space.

Allocation of temporary “bignums” on the stack during bignum computation implies that bignum size is limited:

1- by the heap size;
2- by the stack size !

So, it is possible to get a “full stacks” error using a terminal recursion for Fibonacci if you use bignums (try `(fib 50000)`) …(NB: there is currently a bug in bignums multiplication)..

5.2.1.2 floatings

Every character sequence (that is not inside a string or a symbol interned via the “backslash” macro-char) holding a “.” will be considered as expressing a floating point.

Motorola IEEE “extended-precision floating point” standard is actually used. It allows to express numbers between $1.9*10^{-4951}$ and $1.1*10^{4932}$. There are between 19 and 20 significative digits. It is also possible to denote forbidden operations (divide by zero…) generating “NaNs[3]” or infinity. These NaNs are not readable but are printable and used by Help (in fact your 68882).

The reading may be done using scientific notation (character “`e`” or “`f`” between mantissa and exponent) or classic (mantissa only). The floating is always read in decimal. The mantissa, as the exponent may follow a “`+`” or a “`-`”.

Examples:
```
1.123456789123456789
1.1234f-2001
-233.2e+63
```

5.2.2 The lists (conses)

List expression relies on parenthesis and “conses”[4] (as in Lisp):

- The characters “`(`“, “`)`” et “`|`” sare for list expression and are delimiters;
- The empyu list will be denoted by a special symbol denoted by “`()`”;
- A cons \x(`car` | `cdr`) is represented by: `( <car> | <cdr> )` where `<car>` is the representation of “`car`”…;
- There is a simplified notation (when the `cdr` is a cons or the empty list): the pairs “`| (`“ and the corresponding “`)`” may be omitted . So, one may write: `(1 | ())` or `(1)`.

Examples:
```
(1 | 2)
(1 2 3 4)
(1 | (2 | ()))
(1 (2 3 4) (5 (6)) 1)
```

5.2.3 The cells (vectors)

A vector (or cell) will be represented using special delimiters “`[`“ and “`]`”. A three elements cell `e1`, `e2`, `e3` will be represented by `[<e1> <e2> <e3>]`

[3]NaN= Not a Number. Retourné lors de la lecture d’une séquence de caractères ne formant pas un nombre flottant.

[4]structure de données comportant deux champs non typés appelés (pour des raisons historiques) `CAR` et `CDR` (prononcé “coudair).

Examples: `[]`

`[1 2 3]`

`[[1 2][2 3]]`

5.2.4 Bit-arrays

The delimiter “`%`” is used to express bit-arrays. The sequence that follows should be composed of “`0`” or “`1`”.

Examples: `%`

`%11011000110110001101100011011000`

5.2.5 The strings

The character “`"`“ is used to express strings. Every character following a “`"`“ upto the next “`"`“ swill be included in the string (including control chars, such as carriage return, linefeed...)

5.2.6 Symbols (symbols, constants, errors...)

Every sequnce that is not in one of the previously defined syntactic domain and that does not hols a “`:`” will express a simple symbol.

It is possible to force the reader to read a symbol for any sequence of character using the macro-character “`\`”. So, reading the sequence“`\12\`” will create an interned symbol whose name will be “12”.

Moreover, symbols are organized along a tree (as the packages tree in Le_Lisp) for the user use. A simple symbol will have “`()`” as a father. It is possible to denote a son of a symbol (simple or not) by inserting a “`:`” and the son’s name. So:

`a` ----------------------------- symbol “`a`”, father “`()`”

`a:b` -------------------------- symbol “`b`” , father “`a`”, , father “`()`”

`a:b:c:d:e`-----------------symbol “`e`” , father “`d`”...

Symbols ahave many different uses in Help:

▼ symbols--------------------------------------- **`'a`**
▼ variable--------------------------------------- **`a`**
▼ keywords------------------------------------- (**lambda** (x) (1+ x))
▼ macro-expressions------------------------ (defmacro **useless** (lambda (t) '()))
▼ errors-- **`?:var-undef`**
▼ constants------------------------------------- (**1+** 1)

IThere is no specific micro-syntax for each of these. The user may usually give the type of a symbol using specific closures or syntactic forms.

Upon startup, several symbols are already defined.

5.2.6.1 Keywords:

The following symbols are reserved keywords:

```
lambda cond define =! begin step nomemo
defmacro quote let letrec bindings warn
```

5.2.6.2 Errors:

The following table give the list of predefined errors . Usually, (this is not compulsory) an error is a son of the symbol" `?`".

Symbole	Definition
`?`	Undefined error
`?:too-args`	Too many arguments
`?:few-args`	Too few arguments
`?:bad-type`	Badly types arguments
`?:bad-expr`	Bad expression
`?:syn-keyw`	Keyword use forbidden here
`?:syntx-er`	General syntax error
`?:mem-full`	The heap is full
`?:no-apply`	The object us not applicable
`?:indx-out`	Index out of bounds
`?:strange!`	A strange error has occurred
`?:overflow`	Overflow
`?:cb-break`	Break
`?:lispstck`	The Lisp stack has been destroyed
`?:contstck`	The control stack has been destroyed
`?:varundef`	Undefined variable
`?:stckfull`	Piles pleines (collision)
`?:maxlengt`	Max length reached
`?:io-error`	I/O error
`?:eof-error`	End of file error
`?:dead-cont`	Chronologic continuation is dead

5.2.6.3 Constants:

Many constants are predefineds. Most of them denote closures and will be described in §5.4.

The symbol "`()`" is a constant whose value is itself that denotes the empty list;
Thesymbol "`†`" is a constant whose value is itself that denotes the boolean TRUE;
Thesymbol "`ƒ`" is a constant whose value is itself that denotes the boolean FALSE;

Pay attention that the symbol "`()`" is considered as TRUE in HELP. Only the boolean "`ƒ`" denotes FALSE.

5.2.7 Unreadable objects

Among all the objects existing in memory, some are “unreadable”. These are the:

- ▼ Environments (printable);
- ▼ Closures (printable);
- ▼ Code (partially);
- ▼ I/O Units (unprintable);
- ▼ Suspended forms (unprintable);

5.2.8 Special characters and delimiters

The following table gives the special characters, whether they are delimiters, ad their corresponding (if they exits) delimiters and role. When a number is given, It is the ascii code of the character. These characters are modifiable by the user (Cf §4.2.6).

C	Délim	Corresp	Role
(	Yes	) |	list, cons
)	Yes	(|	list, cons
|	Yes	( )	delimier between CAR and CDR in a cons
[	Yes	]	start of vector
]	Yes	[	end of vector
%	Yes	delim.	start of bit-array
“	Yes	“	start and end of string
;	Yes	13	start of comment on a single line
{	Yes	}	start of comment
}	Yes	{	end of comment
32	Yes		space: delimiter
13	Yes		carriage return: delimiter
\	Yes	\	force interning

5.3 The primitive expressions in Help and derived forms

Every program Help is made of a sequence of definition expressed through `define`.

(**define** <ident> <any>)	Syntactic form	x

Allows to define `<ident>` as denoting `<any>`.

(**define** <closdef> <body>)	Syntactic form	x

allows to define a closure. `<body>` is a sequnce of expression (body of the closure). `<closdef>` is a a list whose CAR should be an identifier (name of the closure defined) . The CDR should be a formal parameters “list” (Cf. `lambda`)

5.3.1 Litteral references (constants, “quoted” symbols ...)

(**quote** <any>)	Syntactic form	x

`(' <any>)`	Syntactic form	`x`

Conclusion

Return the object whose external representation is `<any>`. It is used to reference litterals in the Help code. `(quote <x>)` may be abbreviated to `'<x>`.

```
(quote (+ 1 2))     ➟ (+ 1 2)
```

`<constant>`	Syntactic form	x

Return the objetc whose external representation is `<constant>` (constant objects). It is used to reference litterals in the Help code.

```
†      ➟ †
1      ➟ 1
```

5.3.2 Variables references

`<ident>`	Syntactic form	x

The value returned is the value found at the location which is bound to the identifier in the current environment. If the variable is not bouns, the error`?:varundef` is returned.

5.3.3 Closure application

`(<operator> <operand1>…)`	Syntactic form	x

Closure application is simply expressed by writing the operator (closure) and the operands between parenthesis. The `<operator>` is evaluated, but the `<operand1>…` will be evaluated only if they are "used" (call by need).

```
(+ 1 2)                          ➟ 3
((0 (list - +)) 32 10)     ➟ 22
```

Many closures are predefined. The user may define new closures through the syntactic form `(lambda…)`. Please note that fixsize numbers are actually considered as closures. The positive fixnums select the "nth" element in a sequence (bit array, list, vector…). Negative fixnums return the "-nth" cdr of a list.

```
(0 '[a b c])               ➟ a
(-3 '(z y x w v u))        ➟ (w v u)
```

5.3.4 Closure creation

`(`**`lambda`**` <formals> <body>)`	Syntactic form	x

Returns a closure holding (among others) the environment that existed when the lambda expression was defined (captured environment). When the closure will be applied to effective arguments, its `<body>` will be evaluated in the capturated environment extended with the bindings of the `<formals>` to fresh locations holding the effective arguments. The value returned is the value of the last expression evaluated. If the `<body>` is empty, error `?` is returned.

`<formals>` should take one of the following form:

- `(<ident1> … <identn>)`: The closure will have fixed arity “n”. When applied, each parameter will be bound to each argument.
- `<ident>`: The closure will have variable arity. When applied the formal will be bound to a newly allocated list of the arguments.
- `(<ident1> … | <identn>)`: The closure will have variable arity, (n-1) arguments at least. The last formal parameter will be bond to a newly allocated list of exceding arguments.

`<body>` is a sequence of expression.

```
(lambda(x) x)                  ➟ {Closure (x) x in ()}
((lambda x x) 1 2 3)                  ➟ (1 2 3)
```

5.3.5 Conditional

The syntactic form COND is only there for efficiency (considering its frequent use). A fully compiled release of Help would allow the suppression of this form.

(**cond** <clause1>…)	Syntactic form	x

Each `<clause>` is a sequence of two expressions, the last `<clause>` may be restricted to a sequence of one expression.

The first expression of the first `<clause>` is evaluated. If it returns a FALSE value (`ƒ`), then the following clause is handled (if any, oterwise it returns `ƒ`). If it returns an error, then this error is returned, else the vlaue of the second expression in the `<clause>` is returned (if any, else first expression value is returned).

```
(cond (=? 2 (1+ 1)) “Yes” “No way”)     ➟ Yes
(cond (>? -3 0) -3
     (<? -3 -5) (- 0 -3)
     (+ 2 2))                          ➟ 4
```

5.3.6 Assignment

(**=!** <ident> <any>)	Syntactic form	x

The location whose `<ident>` is bound to will receive the value of `<any>` in the current environment. This Syntactic form is one of the “non functionnal” feature of Help.

```
(=! x 2)            ➟ 2
(=! z (cons 1 z))   ➟ (1 1 1 1 1 1 1 …)
```

5.3.7 Non-memoïzing suspension

(**nomemo** <boolean> <any>…)	Syntactic form	x

The forms `<any>` are evaluated one after the other. According to `<boolean>` value, every suspension that could be created by these forms will be a definitive (true) or memoïzed suspension... One should note that definitive (i.e. not memoïzed) suspensions are nevertheless easily “memoïzable” from “above”. If a suspension s1 created by `nomemo` is itself classically suspended (getting s2) , any access to the suspension s2 will induce s1 evaluation and the memoïzation of its value in s2 forbidding any new evaluation of s1 via s2.

```
(define z
  (nomemo † (cons (print “CAR”)
                  (print “CDR”))))
z                                              ➟ (? | ?)
                                                    prints “CAR” et “CDR”
z                                              ➟ (? | ?)
                                                    prints “CAR” et “CDR”
(define zz (list z z))
zz                                             ➟ ((? | ?) (? | ?))
                                                    prints “CAR”, “CDR” (twice)
zz                                             ➟ ((? | ?) (? | ?))
                                                    prints nothing…
```

5.3.9 Environment

(**bindings**)	Syntactic form	x

Returns the current environment. If it is the top level environment, the symbol `()` is returned. This form can NOT be a closure because the body of the closures are evaluated in the captured environment, but not in the current environment wich is inaccessible for them.

```
(bindings)                          ➟ ()
((lambda(x) (bindings)) 1)          ➟ {Env: «x=1» Then ()}
```

5.3.10 Defining macros

(**defmacro** <any>)	Syntactic form	x

Allows the definition of macros The syntax is identical to `define` syntax. When interpreting, any macro call is physically replaced (via `car=!`) by the value returned upon the first evaluation (displacing macros). When the compiler is used, macros are expanded when compiling. It is therefore necessary to define macros before their use.

```
(defmacro (mapause m)
  `(begin (prin “Pause à “)
          (print ‘,m)
          (pause)))
(define (test n)(mapause test)(1+ n))
```

Après a exécution:

```
test               ➟ {Closure:((n) (begin (prin “Pause à “)
                                          (print ‘test)
                                       (pause)) (1+ n)))…}
```

5.3.10 Defining external functions

(**defext** <fic><seg><nom><str> | <par>)	Syntactic form	x

Allows the importation of external functions (assembly langage for example, see in the “Divers” folder). The external code (written under MPW) must follows conventions given in the §4.5.1.3.1.2, qnd must be linked to the LibHelp.o library…(see folder “Divers”).
The parameters are:

- ▼ `<fic>`-------------------------file name (CODE ressources);
- ▼ `<seg>`-------------------------name of the segment containing the closure code;
- ▼ `<nom>`-------------------------name for the closure to be defined
- ▼ `<str>`-------------------------stricness bit vector (Cf `setstrict`)
- ▼ `<par>`-------------------------parameters of the external closure (as for `define`)

5.3.11 Creating bindings

(**let** <bindings> <body>)	Syntactic form	x

The `<body>` is evaluated in the current environment extended with the `<bindings>`. The scope of the bindings is limited to the `<body>`.

`<bindings>` is a **vector** of lists that can take the following forms:

- `(<ident> <any>)`: `<ident>` will be (lazily) bound to the value of `<any>`.
- `(<closdef> <corpsclos>)`: allows to define a local closure (non recursive closure because of the scope of the bindings) more easily than by using a lambda expression. `<corpsclos>` is a sequence of any expressions, `<closdef>` is a list whose CAR should be an identifier (name of the closure defined) and whoseCDR should be a "list" of formal parameters (Cf. `lambda`).

```
(let [(x 1.1)
     ((double n)(+ n n))]
    (double x))                          ➟ 2.200000000000000e+00
```

(**letrec** <bindings> <body>)	Syntactic form	x

The syntax is the same as for the `let` but the scope of the bindings created is equal to the lexical union of the `<bindings>` and the `<body>`. It is possible (as for the `let`) to define local functions (that can, here use recursion). Because of lazyness, there is no problem (as there are in Scheme by the way) in defining objetcs that immediately refer to identifier bound by the letrec .

```
(let [(r (fib 20))                  ;error en Scheme
     ((fib n)                       ;(fib undefined function)
       (cond (<? n 2) 1
             (+ (fib (1- n))(fib (- n 2)))))]
    r)                              ➟ 10946
```

5.3.12 Sequence

(**begin** <any1>…)	Syntactic form	x

The `<any1>` are evaluated one after the other…. Allows to group evaluations so that they may all be executed "at the same time". This is especially useful when interacting with the user.

```
(let [(x (begin (prin "Entrez x")(read)))]
    (traiter x))                         ➟ …according to traiter
```

5.3.12 Debugging

The error handling system of Help, because of the lazyness (that allows to build objects partially), does not stop the evaluation upon error. According to the "evaluation mode", an error may:

- return an error object which is passed to the current continuation (ƒ mode);
- return an error object which is passed to the current continuation and print an error message on the `stder` input/output unit (() mode);
- put the Help evaluator in a debug mode († mode).

(**pause**)	Syntactic form	x

Simply stop the cuurent evaluation and give the user the opportunity to use a new READ-EVAL-PRINT loop with a prompt whose form will be "`{n}†` " where n is the depth of `pause`s. The loop will be abandonned as soon as the symbol `†` is evaluated (therefore, and because of the prompt appearance, it is enough to type the ENTER key to exit the loop). Any action is possible during the loop . The value returned is the symbol `?` (undefined error).

(**warn** <ident> <any1>…)	Syntactic form	x

Evaluate the `<any1>…` in the evaluation mode specified by `<ident>` (Cf 5.3.12). One should note that suspended form that may be build will efectively be evaluated in the specified mode upon access, even if one as gone out of the `warn` expression (suspended form do capture evaluation mode).

The debugger is actually a simple READ-EVAL-PRINT (Cf `pause`) that may be abandonned (to resume execution) by reading the symbol `†` (true). The expression and the value that (probably) are the reason for the error are printed. The closures and syntactic forms `bindings`, `envar`, `stack`… allow the user to look at the environment, the stack, values… The closure `break` will bring the user back to TopLevel. One should note that evaluating (under debugger) may force suspensions that should not have been forced and therefore may modify the program behaviour. A not fully functionnal program may be greatly disturbed by such inspections.

```
    (warn ƒ (+ 0 'a))                ➟     ?:bad-type
```
et aucun message d'error

(**step** <boolean> <any1>…)	Syntactic form	x

According to `<boolean> value`, will (or not) evaluate the `<any1>…` step by step. One should note that suspended form will be evaluated step by step even if the control is out of the `step` scope (suspensions capture the current evaluation mode, including step or not mode). At each step, the function `step?` (user defined) is called with the form being evaluated and the current environment. If the value returned is flase (`ƒ`), step by step evaluation does not actually take effect for this form. Otherwise, the closure `stepin` is called with he form being evaluated and the current environment. The value returned is then given along with the value of the stepped through form to the `stepout` (user defined) closure. Predefined `step?`, `stepin` and `stepout` closures are proposed in the "Stepper" file. The main purpose of the boolean argument is to desactivate stepping locally.

Example:

```
    (step † (+ 1 2))                 ➟     3  and the following session (for example):
    -> "(+ 1 2)"{1}†
    -> "+"{1}†
    <- "{Closure:{Code 680xx for +} Env:()}"{1}†
    -> "1"{1}†
    <- "1"{1}†
    -> "2"{1}†
    <- "2"{1}†
    <- "3"{1}†
    { = 3 }
```

The step by step mode is really very useful and powerful using the `step?`, `stepin` and `stepout` closures (and to the `(step f …)` form that allows local steppin' desactivation).

For example, using the following definition for `step?`:

```
(define (step? f e)
  (not (number? f)))
```

The previous step by step evaluation becomes:

```
(step † (+ 7 12))          ➟       19
```
and the following session (for example):

```
1 -> "(+ 7 12)"{1}†
2 -> "+"{1}†
2 <- "{Closure:{Code 680xx for +} Env:()}"{1}†
1 <- "19"{1}†
{ = 19 }
```

5.4 Predefined closures in Help

5.4.1 Booleans

The two constants † and ƒ denote booleans. Nevertheless, in ANY test actually made by Help, any value that is not equal to ƒ is considered as true (except for errors, in this case the test fails and returns the error itself). Pay attention the the EMPTY LIST is actually true !

(**not** *any*)	Closure:ProcN	1

Returns † (true) if *any* is equal to ƒ, otherwise returns ƒ.

```
(not †)                    ➟ ƒ
(not 3)                    ➟ ƒ
(not (list 1 2))     ➟ ƒ
(not ƒ)                    ➟ †
(not '())            ➟ ƒ
```

(**boolean?** *any*)	Closure:ProcN	1

Returns † (true) if *any* is equal to † or ƒ. Returns ƒ otherwise.

```
(boolean? †)         ➟ †
(boolean? 3)         ➟ ƒ
```

5.4.2 Equivalence predicates

The three equivalence predicates eq?, =? and equal? define three equivalence relations (reflexive, symetric transitive) on the Help objects. The relation defined by eq? is included in the relation defined by =? itself included in the relation defined by equal?.

(**eq?** *any1 any2*)	Closure:ProcN	2

Returns † (true) if the two objetcs are actually the same one (i.e exist at the same location in memory). Returns ƒ otherwise. Because of their unicity, this predicate is especially useful for symbols.

```
(eq? 'a 'a)          ➟ †
```

```
(eq? 1 2)          ➟ ƒ
(eq? 1 1)          ➟ undefined, probably false
```

(**neq?** *any1 any2*)	Closure:`ProcN`	2

Returns ƒ (false) if the two objetcs are actually the same one (i.e exist at the same location in memory). Returns † otherwise. Because of their unicity, this predicate is especially useful for symbols.

```
(neq? 'a 'a)        ➠ ƒ
(neq? 1 2)          ➠ †
(neq? 1 1)          ➠ undefined, probably true
```

(**=?** *any1 any2*)	Closure:`ProcN`	2

Returns † (true) if both objects are the same (i.e same address in memory) or if they have the same content (according to `eq?` for data structures (eg. conses, vectors) referencing other objects). Returns ƒ otherwise. This predicate is especially useful for numbers, bit-arrays and strings and in some cases for vectors or conses (lists or cells of symbols…).

```
(=? 'a 'a)          ➠ †
(=? 1 2)            ➠ ƒ
(=? 1 1)            ➠ †
(=? %01 %01)        ➠ †
(=? '[a a] '[a a])  ➠ †
```

(**<>?** *any1 any2*)	Closure:`ProcN`	2

Returns ƒ (false) if both objects are the same (i.e same address in memory) or if they have the same content (according to `eq?` for data structures (eg. conses, vectors) referencing other objects). Returns † otherwise. This predicate is especially useful for numbers, bit-arrays and strings and in some cases for vectors or conses (lists or cells of symbols…).

```
(<>? 'a 'a)                 ➠ ƒ
(<>? 1 2)                   ➠ †
(<>? 1 1)                   ➠ ƒ
(<>? %01 %01)               ➠ ƒ
(<>? '(a | b) '(a | b))     ➠ ƒ
```

(**equal?** *any1 any2*)	Closure:`ProcN`	2

Returns † (true) if both objects are equivalent (in the sens of `=?`) or, otherwise, if their content is equivalent (in the sens of `equal?`). Returns ƒ otherwise.

```
(equal? 'a 'a)              ➠ †
(equal? 1 1)            ➠ †
(equal? '(a b) '(a b))  ➠ †
```

(**nequal?** *any1 any2*)	Closure:`ProcN`	2

Returns ƒ (false) if both objects are equivalent (in the sens of `=?`) or, otherwise, if their content is equivalent (in the sens of `equal?`). Returns † otherwise.

```
(nequal? 'a 'a)         ➠ ƒ
(nequal? 1 1)           ➠ ƒ
(nequal? '(a b) '(a b)) ➠ ƒ
```

5.4.3 Lists and conses

A cons (or pointed pair) is an heterogenous data structures made of two fields called (for good old reasons i won't bother you with...) CAR and CDR (pronounce could'er). This structures are actually mutable. Accessing the fiels is done through "numerical selectors".

The main use of conses is to represent lists. A list is defined as being either the empty list `()` or a cons whose CDR is a list.

The empty list `()` is a special object actually denoted by a constant symbol. It contains no element and its length is 0.

(**list?** *any*)	Closure:`ProcN`	1

Returns *ƒ* (false) if *any* is not a list (i.e. a cons or the symbol `()`, dotted lists are considered as lists). Returns † otherwise.

```
(list? '())          ➟ †
(list? '(1 2 3))     ➟ †
(list? 3)            ➟ ƒ
```

(**cons?** *any*)	Closure:`ProcN`	1

Returns † (true) if *any* is a cons. Returns *ƒ* otherwise.

```
(cons? '())          ➟ ƒ
(cons? '(1 2 3))     ➟ †
(cons? 3)            ➟ ƒ
```

(**atom?** *any*)	Closure:`ProcN`	1

Returns *ƒ* (false) if *any* is a cons. Returns † otherwise.

```
(atom? '())          ➟ †
(atom? '(1 2 3))     ➟ ƒ
(atom? 3)            ➟ †
```

(**cons** *any1 any2*)	Closure:`ProcN`	2

Returns a newly allocated cons whose CAR contains *any1* value and whose CDR contains *any2* value. The cons is different [5] of any existing object.

```
(cons 1 2)           ➟ (1 | 2)
(cons 'a '())        ➟ (a)
(cons '(a b) 'c)     ➟ ((a b) | c)
```

(**car=!** *cons any*)	Closure:`ProcN`	2

Allows to write tha value of*any* in the CAR of a *cons*. Because of lazyness, unexpected effects may be observed. This closure is therefore very dangerous. It may disappear from future releases.

```
(car=! '(1 2) 3)     ➟ (3 2)
(car=!  %1010 3)     ➟ ?:bad-type
```

(**cdr=!** *cons any*)	Closure:`ProcN`	2

[5] in the sens of eq?, there is no "hash-consing".

Allows to write tha value of*any* in the CDR of a *cons*. Because of lazyness, unexpected effects may be observed. This closure is therefore very dangerous. It may disappear from future releases.

```
(cdr=! '(1 2) 3)     ➟ (1 | 3)
(cdr=!  %1010 3)     ➟ ?:bad-type
```

(**null?** *any*)	Closure:`ProcN`	1

Returns ƒ (false) if *any* value is not equal to the empty list. Returns † otherwise.

```
(null? '())          ➠ †
(null? '(1 2 3))     ➠ ƒ
```

(**list** *any …*)	Closure:NProc	0

Returns the list of the arguments values.

```
(list 'ts (+ 24 3) '65)    ➠ (ts 27 65)
(list)                     ➠ ()
```

(**length** *list*)	Closure:ProcN	1

Returns the length of *list*. The length of a list is defined by:

- ▼ the length of the empty list is 0;
- ▼ the length of a cons is 1 plus the length of its CDR.

```
(length '())                       ➠ 0
(length '(1 2 3))                  ➠ 3
(length '(mcl (1 2 3) (a b)))      ➠ 3
(length 1)                         ➠ ?:bad-type
```

(**append** *list1 any*)	Closure:ProcN	2

Returns a list made of the elements of the first list*list1* followed by the elements of the list *any*. The second arg. (any) may be something else than a list.

```
(append '(1 2 3) '(4 5 6)) ➠ (1 2 3 4 5 6)
(append '() '(a b))        ➠ (a b)
(append '(a b) '(c | d))   ➠ (a b c | d)
```

(∞ *any*)	Closure:ProcN	1

Returns an infinite list whose elements are all equal to the value of *any*.

```
(∞ 1)          ➠ (1 1 1 1 1 1…)
(∞ (+ 2 3))    ➠ (5 5 5 5 5 5…)
```

(… *number*)	Closure:ProcN	1

Returns a n infinite list made of the integers after *number*.

```
(… 1)          ➠ (1 2 3 4 5 6…)
(… (- -2 1))   ➠ (-3 -2 -1 0 1 2…)
```

5.4.4 Symbols

The symbols main property is that two identical symbols (micro-syntax) are actually identical in memory (in the sens of eq?). This property is very useful to represent variables, constant identifiers, keywords. Help also use symbols to denote errors.

(**symbol?** *any*)	Closure:ProcN	1

Returns † (true) if *any* is a symbol, otherwise returns the object *any*.

(**intern** *symbol any*)	Closure:ProcN	2

Returns a symbol whose name is *any* value (that should be a string or a symbol) and whose father (in the symbols hierarchy) is *symbol*.

```
(intern 'père "fils")      ➟ père:fils
(intern 'père 'fils)       ➟ père:fils
```

5.4.5 Numbers

According to their position, numbers may be interpreted as closures (numerical selectors) or as numerical data. This double point of view may be considered as a semantical weakness but is of great ease (very useful from a pragmatical point of view).

For numerical selectors see §5.3.

(**+** *number number*)	Closure:ProcN	2

Returns the sum of its arguments values. If one of the numbers is a floating, the result will be floating.

```
(+ 10 10)           ➟ 20
(+ 1 2.312)         ➟ 3.312000000000000e+00
(+ 9999999999 1)    ➟ 10000000000
(+ 1 'a)            ➟ ?:bad-type
```

(**-** *number number*)	Closure:ProcN	2

Returns the difference of its arguments values. If one of the numbers is a floating, the result will be floating.

```
(- 10 10)           ➟ 0
(- 2.312 1)         ➟ 1.312000000000000e+00
(- 9999999999 1)    ➟ 9999999998
(- 1 'a)            ➟ ?:bad-type
```

(**1+** *number*)	Closure:ProcN	1

Returns one plus its argument value. If the number is a floating, the result will be floating.

```
(1+ 10)             ➟ 11
(1+ %)       ➟ %00000000000000000000000000000001
(1+ 'a)             ➟ ?:bad-type
```

(**1-** *number*)	Closure:ProcN	1

Returns its argument value minus one. If the number is a floating, the result will be floating.

```
(1- 10)            ➟ 9
(1- 'a)            ➟ ?:bad-type
```

`(* number1 number2)`	Closure:`ProcN`	2

Returns the product of its arguments values. If one of the numbers is a floating, the result will be floating.

```
(* 10 23)     ➟ 230
(* 'a 2)      ➟ ?:bad-type
(* 1e10 2)    ➟ 2.000000000000000e10
```

(**/** *smallnum1 smallnum2*)	Closure:`ProcN`	2

Returns the result from the division of its arguments values. If one of the numbers is a floating, the result will be floating. If both numbers are integers, the result will be an integer.

```
(/ 23 10)           ➟ 2
(/ 'a 2)            ➟ ?:bad-type
(/ 23 10.0)         ➟ 2.300000000000000e0
```

(**modulo** *fix1 fix2*)	Closure:`ProcN`	2

Returns the remainder of the division of *`fix1`* value by *`fix2`* value.

```
(modulo 23 10)      ➟ 3
(modulo 'a 2) ➟ ?:bad-type
```

(**<?** *number1 number2*)	Closure:`ProcN`	2

Returns *ƒ* if *`number1`* value is not strctly less than *`number2`* value. Returns *`number1`* value otherwise.

```
(<? 23 10)          ➟ ƒ
(<? 'a 2)           ➟ ?:bad-type
(<? 12 20)          ➟ 12
```

(**>?** *number1 number2*)	Closure:`ProcN`	2

Returns *ƒ* if *`number1`* value is not strctly more than *`number2`* value. Returns *`number1`* value otherwise.

```
(>? 23 10.0)                      ➟ 23
(>? 'a 2)                  ➟ ?:bad-type
(>? 20 12345678987654321)  ➟ ƒ
```

(**zero?** *number*)	Closure:`ProcN`	1

Returns *ƒ* if *`number`* value is not equal to zero. Returns *`number`* otherwise.

```
(zero? 0)           ➟ 0
(zero? 'a)          ➟ ?:bad-type
```

(**float** *number*)	Closure:`ProcN`	1

Converts *`number`* value to floating.

```
    (float 0)            ➟ 0.00000000000e+0
```

(**cos** *number*)	Closure:`ProcN`	1

Returns the cosinus (floating) of *`number`* value in radians.

(**sin** *number*)	Closure:`ProcN`	1

Returns the sinus (floating) of *`number`* value in radians.

(**tan** *number*)	Closure:`ProcN`	1

Returns the tangent (floating) of *number* value in radians.

(**acos** *number*)	Closure:`ProcN`	1

Returns the arc-osinus (floating) of *number* value in radians.

(**asin** *number*)	Closure:`ProcN`	1

Returns the arc-sinus (floating) of *number* value in radians.

(**atan** *number*)	Closure:`ProcN`	1

Returns the arc-tangent (floating) of *number* value in radians.

(**cosh** *number*)	Closure:`ProcN`	1

Returns the hyperbolic cosinus (floating) of *number* value.

(**sinh** *number*)	Closure:`ProcN`	1

Returns the hyperbolic sinus (floating) of *number* value.

(**tanh** *number*)	Closure:`ProcN`	1

Returns the hyperbolic tangent (floating) of *number* value.

(**atanh** *number*)	Closure:`ProcN`	1

Returns the hyperbolic arc-tangent (floating) of *number* value.

(**log** *number*)	Closure:`ProcN`	1

Returns the neperian logarithm (floating) of *number* value.

(**exp** *number*)	Closure:`ProcN`	1

Returns the exponential (floating) of *number* value.

(**sqrt** *number*)	Closure:`ProcN`	1

Returns the square root (floating) of *number* value.

(**ibase** *fixpos*)	Closure:`ProcN`	1

Modifies lthe current reading numerical base of the reader. The *`fixpos`* value should be between 2 and 36. Returns its argument value. Cf §5.2.1.

```
(ibase 10)          ➟ 10
```

5.4.6 Closures

The closures are made of a “code” that is waiting for a sequence of expressions and from an environment. For optimisation, each closure contains a 16 bits bit-array to express the closure strictness with respect to the 15 first arguments (15 first bits) and the following one (last bit). This bit-array is automatically filled by the compiler (to some extent only), but not at all by the interpreter.

(**apply** *applicable list*)	Closure:`ProcN`	2

Applies the closure or the numerical selector *`applicable`* to the list of argument *`list`*.

```
(apply 1 '((a)))           ➟ a
(apply + '(1 2))           ➟ 3
```

(**getcode** *closure*)	Closure:`ProcN`	1

Returns the code (interpreted or compiled) of the *`closure`*.

```
(getcode 1+)               ➟ {Code 680xx for 1+}
(getcode (lambda(x)x))     ➟ ((x) x)
```

(**getenv** *closure*)	Closure:`ProcN`	1

Returns the *`closure`* captured environment.

```
(getenv 1+)                ➟ ()
(getenv
  (let [(x 2)]
     (lambda(x)x)))        ➟ {Env: «x=2» Then ()}
```

(**getstrict** *closure*)	Closure:`ProcN`	1

Returns the bit-array associated to the *`closure`* that gives the closure strictness (see above).

```
(getstrict cons)           ➟ %00000000000000000000000000000000
(getstrict +)              ➟ %11000000000000000000000000000000
```

(**setstrict** *closure bitarray*)	Closure:`ProcN`	2

Allows the user to give the strictness of the *`closure`* . May lead to important (10-20%) improvements in execution time and memory usage if strict.

5.4.7 Macros

(**expand** *list*)	Closure:`ProcN`	1

The *`list`* should be a “quoted” macro call. Returns the result of the macro expansion.

```
    (expand '(quasiquote (a (unquote b))))  ➟ (list 'a b)
```

5.4.8 Cells (or vectors)

The vectors are heterogenous structures indexed by integers. These structures (as conses) are mutable. Because of lazyness, mutation is potentially dangerous.

When accessed using “nemerical selectors”, vectors are indexed from 0 (first element) to the vector length minus one. the length of a vector is obtained using closure `blength` and then by substracting one from the result.

(**cell?** *any*)	Closure:`ProcN`	1

Returns ƒ (false) if *any* is not a vector, otherwise returns the vector itself.

```
(cell? 23)          ➟ ƒ
(cell? [1 2 3])     ➟ [1 2 3]
```

(**cell** *any* …)	Closure:`NProc`	0

Returns a vector whose content is *any* ….

```
(cell 0 1 2)                ➟ [0 1 2]
(cell 'd 'e 1 '(a) [1])     ➟ [d e 1 (a) [1]]
```

(**makecell** *fixpos*)	Closure:`ProcN`	1

Returns a vector having size *fixpos* and whose fields are all initialized to `?`.

```
(makecell 2)                ➟ [? ?]
```

(**cell=!** *vector fixpos any*)	Closure:`ProcN`	3

Writes `any` in the vector, position `fixpos`. Returns the vector. This non-funtionnal closure has unexpected effects when used (because of Help lazyness).

```
(cell=! '[a b c] 0 0)       ➟ [0 b c]
(cell=! [0 1 2] 3 1)        ➟ ?:indx-out
```

5.4.9 Environments

The Syntactic form `bindings` allos one to access the current environnement. Some closures allows then to handle these environments, or to create new ones. Numerical selectors allows one to access environment contents (non empty env.). If N is the `blength` of the environment, the following indexes are valid:

- Index 0: the lower environment (previous frame);
- from 1 to (N-2)/2: the values of the variables bound in the environment;
- from (N-2)/2+1 to N-1: the identifiers of the variables bound.

Note that the compiler may rely on so-called “short-environments” that do not contains names variables, only their values.

(**environment?** *any*)	Closure:`ProcN`	1

Returns ƒ (false) if *any* is not an environment, otherwise returns the environment itself. as the Syntactic form `bindings` returns `()` if the current enviroement is the toplevel environment, the empty list is considered to be an environment.

```
(environment? 23)                          ➟ ƒ
(environment? '())                         ➟ ()
(environment (let [(x 2)] (bindings)))     ➟ {Env: «x=2» Then ()}
```

(**binding=?** *ident environ*)	Closure:`ProcN`	2

Returns the value of *`ident`* in *`environ`* . If the variable is undefined in the environment, The error `?:varundef` is returned.

```
(binding=? '1+ '())                        ➟ {Code6800x for 1+}
(binding=? 'x (let [(x 2)] (bindings)))  ➟ 2
(binding=? 'a '())                         ➟ ?:varundef
```

(**binding=!** *ident environ any*)	Closure:`ProcN`	3

Modifies the value of *`ident`* in *`environ`*. Returns the value of *`environ`*.

```
(binding=! '1+ '() 2)                      ➟ ()
(binding=! 'a '() 2)                       ➟ ()
```

(**makeenv** *ident1 …*)	Closure:`NProc`	0

Returns an environment where the *`ident`* are all bound to `?`. This environment is automatically linked to the current environnement (the lower frame will be the current env).

```
(makeenv 'a 'b)      ➟ {Env: «a=?» «b=?» Then ()}
(makeenv 1)          ➟ ?:bad-type
```

(**envar** *environ*)	Closure:`ProcN`	1

Returns a vector that contains all the identifiers of the environment *`environ`*. Especially useful during debugging to have a look at an anvironment without forcing all the values referenced in the environment.

```
(envar (makeenv 'a 'b))    ➟ [a b]
```

5.4.10 Bit-arrays

Bit arrays are especially useful to represent sets (and in AI for every “propositionnal logic oriented” software: a clause is simply two sets of litterals, i.e. two bit-arrays). The set-operations are efficiently managed (intersection, union…). Pay attention that bit-set operations are destructive for efficiency. A functionnal behaviour may be obtained using closure `bcopy`.

Bit-array reading is obtained through “numerical selectors” (index starting at 0).

(**bitarray?** *any*)	Closure:`ProcN`	1

Returns ƒ (false) if *`any`* is not a bit-array, otherwise returns the array itself.

```
(bitarray? 23)       ➟ ƒ
(bitarray? %)➟ %
```

(**makebitarray** *posfix*)	Closure:ProcN	1

Returns a bit-array whose length is at least equal to *posfix* , all bits cleared.

```
(makebitarray 0)     ➟ %
(makebitarray 10)    ➟ %00000000000000000000000000000000
```

(**bitand!** *bitarray1 bitarray2*)	Closure:ProcN	2

Logical AND between *bitarray1* and *bitarray2*. The result is stored in *bitarray2*. If the arrys have not the same siwze, the operation is restricted to the smaller argument. This operation is destructive, use `bcopy` if you need a “functionnal” behaviour.

```
(bitand! %0011 %0101)      ➟ %00010000000000000000000000000000
```

(**bitor!** *bitarray1 bitarray2*)	Closure:`ProcN`	2

Logical OR between *bitarray1* and *bitarray2*. The result is stored in *bitarray2*. If the arrys have not the same siwze, the operation is restricted to the smaller argument. This operation is destructive, use `bcopy` if you need a “functionnal” behaviour.

```
(bitor! %0011 %0101)       ➟ %01110000000000000000000000000000
```

(**bitxor!** *bitarray1 bitarray2*)	Closure:`ProcN`	2

Logical XOR between *bitarray1* and *bitarray2*. The result is stored in *bitarray2*. If the arrys have not the same siwze, the operation is restricted to the smaller argument. This operation is destructive, use `bcopy` if you need a “functionnal” behaviour.

```
(bitor! %0011 %0101)       ➟ %01100000000000000000000000000000
```

(**bitnot!** *bitarray*)	Closure:`ProcN`	1

Logical NOT of *bitarray*. The result is stored in *bitarray*. This operation is destructive, use `bcopy` if you need a “functionnal” behaviour.

```
(bitnot! %01) ➟ %10111111111111111111111111111111
```

(**bitcount** *bitarray*)	Closure:`ProcN`	1

Returns the number of bits set in *bitarray*. Execution time in the order of the number of bits set.

```
(bitcount %0101)    ➟ 2
```

(**bitfind** *bitarray*)	Closure:`ProcN`	1

Returns the position of the first bit set in *bitarray* (if any). Returns *f* otherwise.

```
(bitfind %0001)     ➟ 3
(bitfind %)         ➟ f
```

(**bitset!** *bitarray fixpos*)	Closure:`ProcN`	2

Sets (to 1) the bit at the position *fixpos* of *bitarray* . Returns the bit-array. This operation is destructive, use `bcopy` if you need a “functionnal” behaviour.

```
     (bitset! %0 0)       ➟ %10000000000000000000000000000000
```

(**`bitclr!`** *`bitarray fixpos`*)	Closure:`ProcN`	2

Clears (to 0) the bit at the position *`fixpos`* of *`bitarray`* . Returns the bit-array. This operation is destructive, use `bcopy` if you need a “functionnal” behaviour.

```
(bitclr! %1 0)       ➟ %00000000000000000000000000000000
```

(**bitchg!** *bitarray fixpos*)	Closure:`ProcN`	2

Flip the bit at the position *`fixpos`* of *`bitarray`* . Returns the bit-array. This operation is destructive, use `bcopy` if you need a “functionnal” behaviour.

```
(bitchg! %0 0)       ➟ %10000000000000000000000000000000
```

(**zero?** *bitarray*)	Closure:`ProcN`	1

Returns *f* if one of the bits of *`bitarray`* is set. Returns *`bitarray`* otherwise.

```
(zero? %00010)       ➟ f
(zero? %000) ➟ %00000000000000000000000000000000
```

5.4.11 Entrées-Sorties

All input/output is done through so-called “input/output units”. These units may denote a file or a text window. As in C, the variables `stdi`, `stdo` et `stder` denote the units (resp.) for input, output and error. Upon startup, `stdi` and `stdo` are bound to the “current selected window” and `stder` to the “Transcript” window. It is possible to modify the content of this variables to read/write to files...

(**read**)	Closure:`ProcN`	0

Reads a Help expression in the input-output unit in `stdi`. Returns the internal representation of the expression read.

(**print** *any*)	Closure:`ProcN`	1

Print the external representation of the object *`any`* on the current output i/o unit (contained in the variable `stdo`) and then prints a carriage return. One should note that Help printer will force every suspended form in the object (so as to print the object in its completeness). If you need to print an object without forcing its content, use closure `printdebug`. Returns `?`.

(**prin** *any*)	Closure:`ProcN`	1

Print the external representation of the object *`any`* on the current output i/o unit (contained in the variable `stdo`). One should note that Help printer will force every suspended form in the object (so as to print the object in its completeness). If you need to print an object without forcing its content, use closure `printdebug`. Returns `?`.

(**prinlength** *fixpos*)	Closure:`ProcN`	1

The max. printing length (in number of printed objects) is set to `fixpos`. Especially useful to print (partially) infinite objects. Returns the value of its argument.

(**prindepth** *fixpos*)	Closure:`ProcN`	1

The max. printing depth (in number of printed objects) is set to `fixpos`. Especially useful to print (partially) infinite objects. Returns the value of its argument.

(**openi** *string*)	Closure:`ProcN`	1

Opens a file for reading only. The path is given in *string* . It must be specified using ":" to separate folders. The startup folder is the "default" folder. Returns an input-output unit associated to the file.

```
(openi "HD:Help:bob")      ➟ «IO-Unit»
```

(**openo** *string*)	Closure:ProcN	1

Opens a file for writing only. The path is given in *string* . It must be specified using ":" to separate folders. The startup folder is the "default" folder. Returns an input-output unit associated to the file.

```
(openo "HD:Les:deux-pierre(s)")   ➟ «IO-Unit»
```

(**close** *iounit*)	Closure:ProcN	1

Closes the file associated to the *iounit* . Returns `?`. Every attempt to read/write to a closed i-o unit will return a `?:bad-type` error.

(**prinio** *any iounit*)	Closure:ProcN	2

IPrint the external representation of the object *any* on the output i/o unit `iounit`. One should note that Help printer will force every suspended form in the object (so as to print the object in its completeness). If you need to print an object without forcing its content, use closure `printdebug`. Returns `?`.

(**readio** *iounit*)	Closure:ProcN	1

Reads a Help expression in the input-output unit `iounit`. Returns the internal representation of the expression read. Returns error `?:eof-error` when end of file is reached.

(**flushio** *iounit*)	Closure:ProcN	1

Flush the "buffers" of *iounit*. Especially useful when the io-unit is associated to a window (multiple writers).

(**load** *string*)	Closure:ProcN	1

Open the given file, read and evaluates all its content, then close the file.

5.4.12 Erreurs et gestion d'errors

(**printdebug** *any*)	Closure:ProcN	1

Prints, on the io-unit in `stder`, the external representation of the value of *any* without forcing any of the suspensions in the object (then prints a Carriage return). A suspension is printed as a tuple made of the suspended code and the captured environment. Returns `?`.

`(printdebug (cons a b))` affichera `({Susp: a in ()} | {Susp: b in ()})`

(**error** *error any*)	Closure:`ProcN`	2

Raise error *error* with message *any*. According to the current “error handling mode”, it will result in message printing, value return or debugger call.

(**error?** *any*)	Closure:`ProcN`	1

Returns † if the value of*any* is an “error” (Cf the closure `type`). Returns ƒ otherwise.

```
(error? 1)          ➟     ƒ
(error? (1+ 'a))    ➟     †
```

(**explain** *error*)	Closure:`ProcN`	1

Returns the error message associated to *error*.

```
(explain '?:varundef)      ➟ "Variable non définie"
```

5.4.13 Control

Closures for control are few in Help. The introduction of escapes or explicit continuation (à la Scheme) brings few comfort in a non-parallel lazy frame. Nevertheless, the closure `force` allows to give to these facility all their original power, the primitive `call/ep` is used to express escapes (efficeint, but very limited use); the closure `call/cc` allows to capture the current continuation (à la Scheme).

(**call/ep** *closure*)	Closure:`ProcN`	1

Gives to the *closure* (that should accept one argument) a closure of type ProcN, arity 1 that (when called) will allow to escape from the continuation and will give its argument value to tha `call/ep` continuation (as call/cc in Scheme). Nevertheless, this is only a “chronological continuation ” whose extent is limited[6] (stack overwriting); therefore, it is useful only for “escapes” (catch/throw, tag/exit facilities).

```
(call/ep (lambda(k)(k 1) 2))     ➟ 1
```

(**call/cc** *closure*)	Closure:`ProcN`	1

Gives to the *closure* (that should accept one argument) a closure of type ProcN, arity 1 that (when called) will allow to escape from the continuation and will give its argument value to tha `call/ep` continuation (as call/cc in Scheme). This continuation as unlimited extent but may need much memory and execution time (stacks are simply copied).

```
(call/cc (lambda(k)(k 1) 2))     ➟ 1
```

(**force** *any*)	Closure:`ProcN`	1

[6]c’est le seul objet Help ayant une durée de vie limitée.

From a fonctionnal point of view, it is identity. However, it recursively forces all the suspensions in the value of *any*. It is essentially useful to simulate the transformation of Help suspensions in “futures” of a parallel machine and also to give back some dirty tricks (affectation, escapes…) all their “functionnalities”…

```
(=! x 2)                            ➟ 2
(=! x (force (cons 1 x)))        ➟ (1 | 2)    ;but not (1 1 1 1 1 1 1 1...)
(force
  (call/cc
    (lambda(k)
      (cons 1 (k ‘Sortir)))))    ➟ Sortir     ;et non (1 | <teratos>)
```

(**if** *any1 any2 …*)	Closure:`NProc`	0

Because of Help lazyness, the conditionnal may be implemented as a closure in Help. The closure `if` is the embodiement as a closure of the syntactic form `cond` (that is kept only for efficiency reasons !).

```
(if (eq? ‘a ‘b) 1
   (=? 1 2)    2
   3)                       ➟ 3
```

(**eval** *any environ*)	Closure:`ProcN`	2

In the interpreted release of Help (if a realistic compiler may exist one day), this closure evaluate the form *quelconque* in the environment *environ*. In a compiled release, it should compile the form *any* in the environment *environ* and then execute the code compiled. One should note that if *any* is already compiled (`CODE` type object) , the code is immediatly executed in the environment given. This allows an efficient `EVAL` execution if the forms have been precompiled (for multiple evaluation, exemple: constraints in a CSP like constraint solver).

```
(eval 1 ‘())                                 ➟ 1
(eval (cons x x) ((lambda(x) (bindings)) ‘a)) ➟ (a | a)
```

(**or** *any1…*)	Closure:`NProc`	0

Evaluates the *any1…* forms one after the other up to the point when one the value returned is not false or end of the *any1…* Returns the value of the last form evaluated.

```
(or (null? ‘a) (null? ‘()))            ➟ †
(or (number? ‘a) (number? 1.1))  ➟ 1.1000000000000000e00
```

(**and** *any1…*)	Closure:`NProc`	0

Evaluates the *any1…* forms one after the other up to the point when one the value returned is false or end of the *any1…* Returns the value of the last form evaluated.

```
(and (null? ‘a) (null? ‘()))      ➟ ƒ
(and (number? 10) (number? 1.1)) ➟ 1.1000000000000000e00
```

5.4.14 System

Here are all the closures that have some accointance with memory management, operating system…

(**type** *any*)	Closure:`ProcN`	`1`

Returns the type (contained in the “tag” of the referenced block) of *`any`* value as a number (*`fix`*). The following table, or simple application of the closure `type` to an object gives the type interpretation:

integer fixed size	`1`	floating point number	`2`
bit-array	`3`	string	`4`
input/output unit	`5`	variable identifier (symbol)	`6`
constant identifier (symbol)	`7`	error (symbol)	`8`
macro (symbol)	`9`	primitive key-word (symbol)	`10`
compiled or assembly code	`11`	closure	`12`
cons	`13`	vector	`14`
indirection[7]	`15`	environment	`16`
short environnement	`17`	bignum	`19`
memoïzable suspension[8]	`20`	non-memoïzable suspension[9]	`21`

[7]type invisible pour l’utilisateur.

[8]type invisible pour l’utilisateur.

[9]type invisible pour l’utilisateur.

```
(type 'a)                        ➟ 6
(type (+ 9999999999 1))          ➟ 19
```

(**coerce** *any fixpos*)	Closure:`ProcN`	2

Physically changes the type of *`any`* in *`fixpos`* . Use `type` closure to get existing type. This function may foul the GC (Garbage COllector or “Glaneur de Cellules”), that will free unexisting or referencable blocks leading to hard errors(Bus Error, Adress Error…). Harmless type conversion are possible between elements of the following group of types (note that coerce only changes the type of the object, not its content).

- ▼ `1 2 3 4 19----------` numbers et bit-arrays
- ▼ `6 7 8 9 10----------` identificateurs…
- ▼ `16 17 14------------` environnements et vectors

(**runtime**)	Closure:`ProcN`	0

Returns the time in 1/60 seconds since system boot.

(**chrono** *any*)	Closure:`ProcN`	1

Returns a three element vector. The first one is the value of *`any`*. The next one is the excution time (in seconds, ±1/60 second) to evaluate the form *`any`* (Pay attention to lazyness, a `print` or a `force` may be useful to get a full evaluation) and the last one is the time used for Garbage Collection (Glanage de Cellules). Note that `chrono` automatically calls the compacifier GC BEFOREevaluating the form *`any`* to get a significant “best” time (the time to make this GC is obviously not taken into account in the results given…). If the *`any`* form uses the GC via the closures `compgc` or `masgc`, the result retourned will almost ignore the time taken for these GC.

```
(chrono (fib 20))    ➟ [10946 5.4000000000e+0 0.000000000000e+0]
```

(**masgc**)	Closure:`ProcN`	0

Calls the memory manager to execute a “Mark and Sweep” type GC. This GC is very efficient, but does not suppress memory fragmentation. Nevertheless it will suppress every “indirection” blocks that could have been installed by forced closures or via the closure `replace`. Isolated garbage blocks with size 1 or 2 are not collected. Symbols whose value is undefined and which are not referencable ARE collected. The value returned is `?`.

(**compgc**)	Closure:`ProcN`	0

Calls the memory manager to execute a compacifying modified“Break Table” GC. This GC is slower, but suppresses fragmentation. It also collects all “indirection” type blocks (that could have been created via suspension forcing or `replace` closure). Symbols whose value is undefined and which are not referencable ARE collected. The value returned is `?`.

(**blength** *any*)	Closure:`ProcN`	1

Returns the memory used by *any* in long words (32 bits). rhe memory used by the “tag” associated to every object (1 LW) is not taken into account.

```
(blength 'a)        ➟ 6
(blength 2)         ➟ 1
```

(**bcopy** *any*)	Closure:`ProcN`	1

Returns a copy of the *any* . This is a simple surface copy (copy of the object itself, not of the objects referenced by it. It should not be employed on objects whose unicity is garanteed by the system (symbols…).

```
(bcopy 1)           ➟ 1
(eq? (bcopy x) x)   ➟ ƒ  ;(suppose value(x) ≠ symbol)
(=?  (bcopy x) x)   ➟ †   ;idem
```

(**replace** *any1 any2*)	Closure:`ProcN`	2

Replace physically every occurrence (in the sens of `eq?`) of *any1* by *any2*. This closure uses the indirections to operate. Use it with MUCH care on symbols (especially ƒ, †…). Its utility is yet to be determined, but is seems powerful…

```
(define x '(a b c)) ➟ (a b c)
(replace 'b 'k)     ➟ k
x                   ➟ (a k c)
```

(**≈**)	Closure:`ProcN`	0

Obtenu au clavier par Option-X. Returns la dernière valeur obtenue au top-level.

```
(openi "palmipède") ➟ «IO-Unit»
(readio (≈))        ➟ (define (palmipede v)…)
```

(**where** *any*)	Closure:`ProcN`	1

Returns the memory address of *any* in a fix size integer (32 bits).

(**find** *any*)	Closure:`ProcN`	1

Returns the list of every identifier whose value in the global environment is equal (in the sens of `eq?`) to the value of *`any`*. especially useful to find the symbol containing a closure or code…

```
(define x 1+) ➟ {Closure: {Code 6800xx for 1+} in {Env: ()}}
(find x)      ➟ (x 1+)
```

5.5 Help interface

We will describe the MacIntosh implementation. Note that:

- This implementation is usable only with 68020 or better microprocessors, a 68881 (or better) floating point unit is needed to use floating points (you may also use software emulation packages if you lack the 6888x). MMU is yet left unused.
- This implementation is not yet complete. The compiler is not really usable, the development environment is yet very poor (no printing, files limited to 32Ko…).

5.5.1 Configuration

The Help interpreter can run with very few memory (less than 500 Ko) if you are ready to cope with many GC. One Mo is a nominal value. 4 Mo is really comfortable for most uses. Help runs under Multifinfer, but current release won't really cope with System 7 (A bug probably coming from MPW 3.0 will bomb when QUITTING Help, therefore you may use Help under 7.0, but Quit = Reboot).

The memory "asked for" by Help is simply the memory size indicated in the "Get information" (Command-I) under the Finder. One can change it easily. By default, this size is 512 Ko and should therfore be modified if possible.

To modify other features of Help, A set of resedit "TEMPLATES" ressources are given (insert these ressources in your Resedit copy) . They allow the user to simply modify Help interesting ressources .

5.5.1.1 The ressource CONF (Id 0, "Configuration")

Contains four data, user modifiables:

- ▼ Font Id: choose the default font of Help editor. The initial value (22) is for Courier (fixedwidth font, for better indent).
- ▼ Font Size: Choose the default size for the font. Default: 9.
- ▼ Block Visu: choose the duration (in 1/60 seconds) of lexical block visulisation (parenthesis match…) when using mouse click or cursor arrows. Initial value: 6 for 1/10 second.
- ▼ Stack Memory %Age: choose how much memory you want to give to stacks. Initial value:10% (should be enough for almost any application).

5.5.1.2 Ressources STC#

They contain "C" strings. The first one(Id 0, "SynF Names") holds the name of every syntactic form of the language. These names may be modified to user convenience (You will have to modify every help source also…).

The next one (Id 1, "Erreurs") cholds alternatively the error message and the error name of every help error. Again, can go wild modifying error message (no side-effects) or error names (but pay attention to sources taht do refer to error names).

5.5.1.3 Ressource STCL (Id 0, "Startup File")

It simply contains the name of the startup file, loaded by Help when starting. (Initial value: Start”). It is possibel to use a full path (See closures `openi` et `openo`).

5.5.1.4 Ressource STCN (Id 0, “Closures Names”)

It contains informations related to predefined closures. Four fields for each closure:

- ▼ Strictness: a number in hexa that should be interpreted as a 16 bit bit-array. The weak 15 bits gives strictness of the closure with respect to the first 15 args. The last one (the higher one) gives the stricness for any supplementary argument. It is therefore “easy” to suppress most lazyness from help by setting EVERY strictness to $FFFF and by using the macro `defkap and kappa` (instead of `define` and `lambda`, see Start file).

- ▼ Arity: Gives the arity (or minimal arity) of the closure. User modification is neither required nor advised !

- ▼ Type: Gives the type of the closure (fixed arity:0, variable arity: 1). User modification is neither required nor advised !

- ▼ The string: Gives the name of the symbol that will denote the closure. If you modify this one, you will have to modify every sources that refer to this closure name ! (including Start file !).

5.5.1.5 Ressource CART (Id 200, “Reader conf”)

Contains 256 bytes that give the reader the type of each ASCII codes. You may therefore modify the charcters for lists, vectors... See § 5.2.8. You will again have to modify any source that may use these chars.

5.5.1.6 Ressources WIND

To perfectly match your screen, the sizes and position of default Help windows are user-modifiable. You may set:

- ▼ Size and position of the “Transcript” window, ID 1000;
- ▼ Sizes and positions of the five edit windows, used cyclicly by Help at each window creation (ID 1001 to 1005).

5.5.2 Using the editor

The editor intensively relies on ROM routines and inherit their limitations. The most important one being that no text should be longer than 32Ko. Undefined[10] things may occur when this limit is reached

The “Transcript” window receives very error message (it is associated to the default error input/output unit denoted bythe symbol “`stder`”). It has every limitation of other windows (32 Ko limit) so think to clean it from time to time (Command-A, Backspace. Forgetting to clean it will simply lead to memory loss and weird printing... no bomb).

[10]Wthout any danger, i.e. you should be able to save anyway, then split the file using a decent text editor.

Every window is an edit and evaluate window. The scheme used rely on the validation key used (and is really great to use IMHO). Two different modes:

- ▼ The carraige return key is for editing. It will simply change from one line to the following one, with automatic indentation and “(“, “[“ or “{“ matching.
- ▼ The ENTER key(numerical keyboard) is for evaluation. If a sequence of chars has already been selected, this sequence will be evaluated. Otherwise, the previous S-expression will be automatically selected and evaluated (try it, you will understand immediatly).

In this case (ENTER key) it is possible to change the READ-EVAL-PRINT behaviour:

- ▼ If you press shift-ENTER instead of ENTER, no value will be printed. This is VERY useful in the frame of lazy evaluation (no suspension will be forced by the printer).
- ▼ If you press Option-ENTER instead of enter, the value will be printed on the error input-output default unit (denoted by the symbol `stder`). This is especially useful to evaluate part of (or a whole) file without damaging it. Example: to load a file, select open, type command-A (select all) then Option-ENTER. Note that values are always printed between “comment chars” ({ and }).

The mouse “click” allows to easily check the “matching” of the “`()`”, “`[]`” and “`{}`”. If their is no match, you will hear a beep. Another useful trick: if you click with the Option key pressed, the matching is verified, shown and SELECTED. You will then be able to cut, paste... it. The double click selects a “word”.

5.5.2 The valuator and the “bugs”...

Help has been used for much, much time, and very few bugs are known (in fact, only ONE, in bignum multiplication). In case of bug during evaluation (The mouse pointer will look like a small Macintosh) or during a GC (Mouse pointer in Sweep, mark, or Compacify 1,2 or 3 state):

- ▼ If you use a debugger, simply GO to the address contained in the D6 register (type “`G D6`” under MacsBug). You will return to toplevel in the best possible conditions (stacks and main Help machine registers set to default values). Nevertheless, the system may be unstable, especially if Help or macintosh heap are damaged. You are advised to save and exit (or reboot).
- ▼ Otherwise, The “System Error Manager” will try to draw a simple dialog with an error message and two buttons. If the Mac Heap is damaged...This drawing may be partial. Simply rememberthat LEFT button is “Reboot” and RIGHT is “Resume” (it will resume Help as if you had a debugger and had typed “`G D6`” (If the screen appearance has been damaged by the dialog, simply zoom and unzoom one of Help windows). Nevertheless, the system may be unstable, especially if Help or macintosh heap are damaged. You are advised to save and exit (or reboot).

Bibliography

[Abelson 85] : Harold Abelson et Gerald Jay Sussman avec Julie Sussman
Structure and Interpretation of Computer Programs
M.I.T. Press, Cambridge, 1985

[Aho 83]: A. Aho, J. Hopcroft et J. Ullmann
Structures de données et Algorithmes
InterEditions 87 (orig. Addison & Wesley 83)

[Aho 86]: A. Aho, R. Sethi et J. Ullmann
Compilateurs: Principes, Techniques et Outils
InterEditions 89 (orig. Addison & Wesley 86)

[Allen 78]: John Allen
Anatomy of LISP
McGraw-Hill Inc., 1978

[Allison 86]: Lloyd Allison
A Practical introduction to denotational semantics
Cambridge University Press, 1986

[Ashcroft 85]: Edward A. Ashcroft, William W. Wadge
Lucid, the Dataflow Programming Language
Academic Press, 1985

[Avenhaus 90] J. Avenhaus & K. Madlener
Term Rewriting and Equationnal Reasoning
Elsevier Science Publishers - North Holland, 1990

[Barendregt 84] Barendregt H.P.
The Lambda calculus, Its syntax and semantics
North Holland, 1984

[Bloss 88] A. Bloss, P. Hudak, J. Young
Code Optimisations for Lazy Evaluation
Lisp ans Symbolic Computation ,Vol. 1, N° 2,p 147-164 (1988)

[Briot 86] J.P. Briot, P. Cointe & E. Saint-James
Réécriture et récursion dans a closure
Journées Langages Orientés Objet - p90-100

[Cayrol 85] : Cayrol Michel
Conception, Formalisation et Expérimentation d'un modèle pour le traitement d'objets finis ou infinis dénombrables.
Thèse d'Etat , Université Paul Sabatier, 1985

[Cayrol 87] : Schiex Thomas, Cayrol Michel
Psil: L'infini en programmation
AFCET-RFIA 1987

[Cayrol 92] Cayrol Michel, Palmade Olivier, Schiex Thomas
A fixed point Semantics for the ATMS·
Journal of Logic and Computation (to appear), 1992

[Chailloux 80]: Jérôme Chailloux
Le Modèle VLisp: Description, Implémentation et Evaluation
Thèse de troisième cycle, Université P. et M. Curie (Paris VI), 1980

[Chailloux ??]: Jérôme Chailloux & ??

Manuel Le_Lisp version 15.21
INRIA - 19??

[Clinger 82]: William Clinger
NonDeterministic Call by Need is Neither Lazy Nor by Name
ACM Symposium on Lisp and Functionnal Programming, 1982

[Clinger 87]: William Clinger, Jonathan Rees (Editors)
Revised^3 Report on the Algorithmic Language Scheme
M.I.T. Artificial Intelligence Memo.

[CM2 87]: Thinking Machines Company
Connection Machine - Model CM-2 - Technical Summary
Thinking Machines technical report HA87-4 , 1987

[Cohen 83]: Comparison of Compacting Algorithms for Garbage Collection
Jacques Cohen & Alexandru Nicolau
ACM Transactions on Prgramming Languages and Systems Vol5, N°4, Octobre 1983 - p532-553

[Cousineau 89]: Guy Cousineau et Gérard Huet
The CAML Primer - Projet Formel
INRIA-ENS- 1989

[Dybvig 90]: R. Kent Dybvig & Robert Hieb
A New Approach to Procedures with Variable Arity
Lisp & Symbolic computation, Vol.3, N°3, p229-244 (1990)

[Field 88]: Anthony J. Field, Peter G. Harisson
Functionnal Programming
Addison Wesley Publishing company - 1988

[Gabriel 85]: Richard P. Gabriel
Performance and Evaluation of Lisp Systems
The MIT Press - 1985

[Girardot 85]: Jean Jacques Girardot
Les langages et les systèmes LISP
EdiTests, 1985

[Halstead 85]: Halstead R.H.
MultiLisp: A Language for concurrent symbolic computation.
ACM Transactions on Prgramming languages and systems 7(4),
(p 501-538) (Octobre 1985).

[Hillis ??]: Hillis ??
The Connection machine

[Hindley 86]: J. Roger Hindley & Jonathan P. Seldin
Introduction to Combinators and λ-Calculus
Cambridge University Press, 1986
Addison Wesley Publishing Company, 1968

[Jaulent 87]: P. Jaulent & L. Baticle
µ-processeurs 68020, 68030 et leurs coprocesseurs.
Eyrolles (1987)

[Knuth 68]: Donald E. Knuth
The Art of Computer Programming. Vol.1. Fundamental Algorithms
Addison Wesley Publishing Company, 1968

[Mac 85-86]: Apple Computer Inc.
Inside MacIntosh, Vol I à V
Addison Wesley Publishing Company, 1985-86

[Schiex 87] : Schiex Thomas
Psil: Manipulation d’objets infinis dénombrables
Rapport de D.E.A., Université Paul Sabatier, 1987

[Schiex 88] Schiex Thomas
Psil et la Connection Machine
Rapport pour le C.N.R.S et le Conseil Régional (1988)

[Schiex 89] : Schiex Thomas
Psil: A héritier de Scheme
BIGRE: Special Issue : “Putting Scheme to work” , 1989

[Schiex 91] Schiex Thomas
Interprétation et Compilation d'un dialecte paresseux de Scheme: Help
Phd. Thesis
Université Paul Sabatier, Toulouse, France

[Steele 90] Guy L. Steele Jr.
Common Lisp: the language (2nd edition)
Digital Press - 1990

[Tarski 55] Tarski A.
A Lattice-theorical FixPoint Theorem and its Applications
Pacific J. Math. (p285-309), 1955

Examples

```
{Function composition}
{••••••••••••••••••••}
(define (rond f g)
       (lambda x (f (apply g x))))

{integers}
{••••••••}
(define n (… 0))

{all the fibonacci's}
{•••••••••••••••••••}
(define (fibn n1 n2) (cons n1 (fibn (+ n1 n2) n1)))
(define fibl (cons 1 (cons 1 (map (∞ +) fibl (-1 fibl)))))

{A strange suite defined by a fixpoint}
{•••••••••••••••••••••••••••••••••••••}
(define (entrelace l1 l2)
     (cons (0 l1) (cons (0 l2) (entrelace (-1 l1) (-1 l2)))))
(define biz (entrelace (… 0) biz))

{readin flow}
{•••••••••••}
(define (in)(cons (read)(in)))
(define input (in))

{Factorielle CPS…}
{••••••••••••••••}
(define (fact x k)
    (cond (zero? x)(k 1)
    (fact (- x 1)(lambda(n)(k (* x n))))))
(define factl (cons 1 (map (∞ *) (… 1) factl)))

{Eratosthene crible}
{••••••••••••••••••}
(define (erat l)
    (cons (0 l)
          (erat (diff (-1 l)
                      (map (∞ *) (∞ (0 l)) l)))))

{Decomposition in prime numbers}
{••••••••••••••••••••••••••••••}
(define (dec n l)   ; l should be the prime numbers list
 (cond (=? n 1) ()
       (zero? (modulo n (0 l))) (cons (0 l) (dec (/ n (0 l)) l))
       (< n (* (0 l)(0 l))) (list n)
       (dec n (-1 l))))

{Church numerals}
{•••••••••••••••}
(define plus
 (lambda(n1)
        (lambda(n2)
               (lambda(f)
                      (lambda(x)
                             ((n1 f) ((n2 f) x)))))))

(define zero (lambda(f) (lambda(x) x)))

(define mul
 (lambda(n1)
        (lambda(n2)
               (lambda(f)
                      (lambda(x)
                             ((n1 (n2 f)) x))))))
(define suc
 (lambda(n)
        (lambda(f)
               (lambda(x) ((n f)(f x))))))

(define a (suc zero))
(define deux (suc a))
(define trois (suc deux))
(define quatre ((mul deux)deux))

(define exp
 (lambda(n1)
        (lambda(n2)
               (lambda(f)
                      (lambda(x)
                             (((n2 n1) f) x))))))

{The paradoxical combinators}
{•••••••••••••••••••••••••••}

;by Church Y0
(define (Y0 g)
 ((lambda(x) (G (x x)))(lambda(x)(G (x x)))))

{The fixed point for fixed point combinator}
(define (G y)
 (lambda(f) (f (y f))))

;by Turing Y1=Y0 G
(define Y1
 ((lambda(a)
   (lambda(b) (b ((a a) b)))) (lambda(a)
                                (lambda(b) (b ((a a) b))))))
;another WEIRD fpc by Klop
(define £
(lambda(a)
 (lambda(b)
  (lambda(c)
   (lambda(d)
    (lambda(e)
     (lambda(f)
      (lambda(g)
       (lambda(h)
        (lambda(i)
         (lambda(j)
          (lambda(k)
           (lambda(l)
            (lambda(m)
             (lambda(n)
              (lambda(o)
               (lambda(p)
                (lambda(q)
                 (lambda(s)
                  (lambda(t)
                   (lambda(u)
                    (lambda(v)
                     (lambda(w)
                      (lambda(x)
                       (lambda(y)
                        (lambda(z)
                         (lambda(r)
(r (((((((((((((((((((((((((
            (t h)i)s)i)s)a)f)i)x)e)d)p)o)i)n)t)c)o)m)b)i)n)a)t)o)r)
                          ))))))))))))))))))))))))))))
```

Conclusion

```
(define $ ((((((((((((((((((((((((
         (£ £)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)

;another one, "à la Klop" in a French release
(define £
(lambda(a)
 (lambda(b)
  (lambda(c)
   (lambda(e)
    (lambda(f)
     (lambda(g)
      (lambda(h)
       (lambda(i)
        (lambda(j)
         (lambda(k)
          (lambda(l)
           (lambda(m)
            (lambda(n)
             (lambda(o)
              (lambda(p)
               (lambda(q)
                (lambda(r)
                 (lambda(s)
                  (lambda(t)
                   (lambda(u)
                    (lambda(v)
                     (lambda(w)
                      (lambda(x)
                       (lambda(y)
                        (lambda(z)
                         (lambda(d)
(d (((((((((((((((((((((((((
             (H e)l)p)E)s)t)T)e)r)r)i)b)l)e)m)e)n)t)F)l)e)m)m)a)r)d)
                           ))))))))))))))))))))))))))))

;factorial funtionnal, to use with Y combinators
(define (FF f)
 (lambda(x)
   (cond (zero? x) 1
         (* (f (1- x)) x))))

{computation with unknowns: replace the leafs of a tree by the maximum leaf}
{••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••}
(define (rpm l)
 (letrec [((f n rest&maxc)
              (cons (cons max (0 rest&maxc))
                    (cond (>? n (-1 rest&maxc)) n (-1 rest&maxc))))
          (rest (reduce f '(() | 0) l))
          (max (-1 rest))]
         (0 rest)))
```

Index