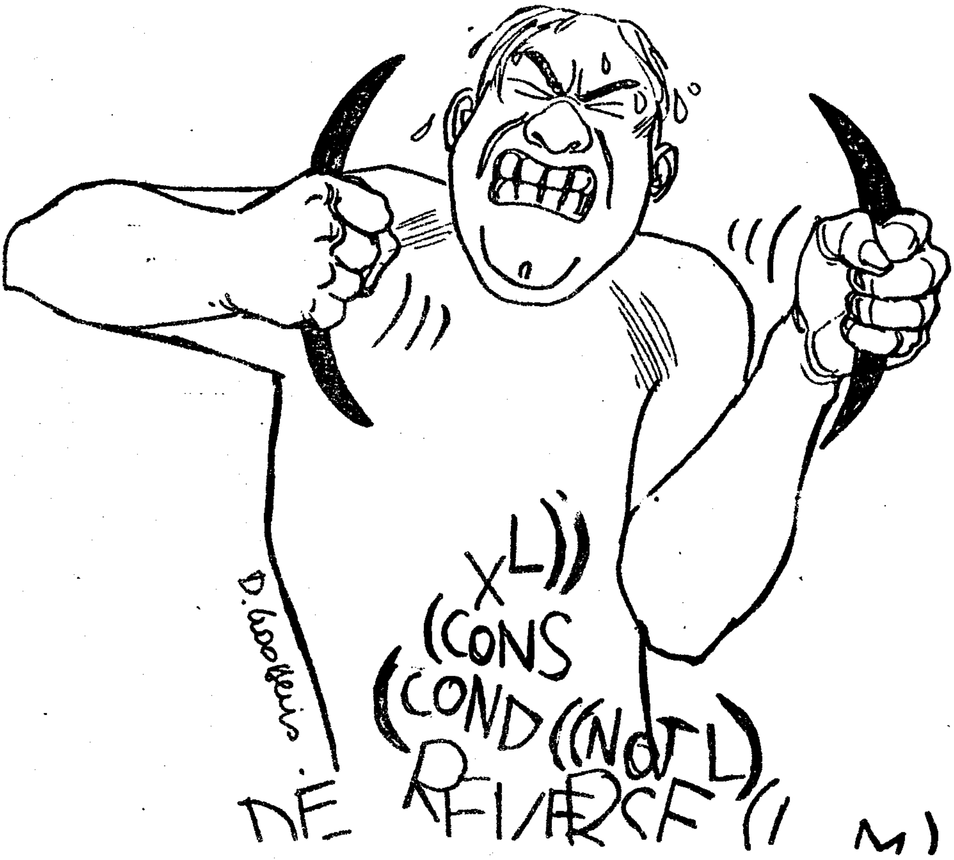
(LISP BULLETIN)



#2
July 1978

Editors : P. GREUSSAY
Dept. Informatique, Universite Paris-8-Vincennes
Route de la Tourelle, Paris 75012, FRANCE

J. LAUBSCH
Institut fur Informatik, Universitat Stuttgart
Azenbergstr. 12, 7000 Stuttgart 1, BRD

Table of Contents

from the editors 1

LISP puzzles 2

LISP books 3

Book review 4

LISP thesis' 6

Announcements for LISP systems 7

The use of LISP in Western Germany 10

Current LISP manuals 14

Useful functions 16

Technical notes 19

- A VLISP Interpreter on the VCMC1 Machine 19
- A System to Understand Incorrect Programs 27
- Iterative Interpretation of Tail-Recursive LISP Procedures 35

from the editors

Well, the (LISP BULLETIN) is back again. It was invented, a long time ago, by Daniel G. Bobrow, and a first issue was published in SIGPLAN, September 1969. The bulletin has exceptionnally deep roots and cannot be pulled free. So now here is the second issue (which explains the #2 on the cover). We think that the main purpose of the (LISP BULLETIN) is to make easier the communication between members of the LISP community.

We welcome contributions. The following topics are particularly encouraged

- announcements for new books
- book and paper reviews
- small technical papers
- useful functions
- comments
- puzzles
- announcements for new LISP systems
- bars of silver
- precious jewelry

As you may notice we have not changed our subscription rate (free), acknowledging the frequency of publication. We will try to stick to a more regular schedule, so our subscription policy may be changed in the future.

In the meantime, enjoy it.

(LISP PUZZLES

from P. GREUSSAY
This is the function SKE. What is she doing?

```
(DE ske (l r)
    (COND
       ((ATOM l) NIL)
       ((MEMQ l r) T)
       ((ske (CAR l) (CONS l r)) T)
       (T (ske (CDR l) (CONS l r)))))
```

from J. ALLEN
This is the function FOO. What is she doing?

```
(DE foo (l)
    (COND
       ((NULL l) NIL)
       ((NULL (CDR l)) l)
       (T (CONS (CAR (foo (CDR l)))
              (foo (CONS (CAR l)
                     (foo (CDR (foo (CDR l))))))))))
```

from H. SAMET
This is the function BAR. What is she doing?

```
(DE bar (x y)
   (IF (< x 2) (ADD1 y)
       (bar (SUB1 x) (bar (- x 2) y))))
```

from D. GOOSSENS
This is the function MOBY. What is she doing?

```
(DE moby (l)
    (IF (NULL (CDR l))
        (CAR l)
        (moby (CDDR (APPEND l [(CAR L)])))))
```

from H. BOLEY
Write a LISP function NOTHING without parameter such that the call (NOTHING) does 'nothing', i.e. no value [including NIL] is returned and no effect on the further interaction with LISP is noticed. So in LISP 1.6 for.example NOTHING should enable the following interaction :

```
* (SETQ X 2)
2
* (NOTHING)
* X
2
* ...  further normal LISP 1.6 interaction ...
```

SEND MORE PUZZLES)

(LISP BOOKS

ALLEN J., The Anatomy of LISP, McGraw Hill, New York 1977

BERKELEY E.C. & BOBROW D.G., (ed), The Programming Language LISP : Its Operation and Applications, Information International Inc., The M.I.T. Press, Cambridge, Mass., 1964

FRIEDMAN D.P., The little lisper, Science Research Associates Inc., 1974

MAURER W.D., The Programmer's Introduction to LISP, Mac Donald / American Elsevier Computer Monographs, 1972

Mc CARTHY J. & TALCOTT C., LISP Programming and Proving, Stanford University, march 1978

Mc CARTHY J. et al, LISP 1.5 Programmer's Manual, The M.I.T. Press, Cambridge, Mass., 1962

NAKANISHI M., Initiation to LISP, Modern Science Co., Tokyo, Japan, 1977

PETER R., Rekursive Funktionen in der Komputer-Theorie, Akademiai Kiado, Budapest, 1976

RIBBENS D., Programmation non-numerique LISP 1.5, Dunod, Paris, 1969

SIKLOSSY L., Let's Talk LISP, Prentice Hall, Englewood Cliff, New Jersey, 1976

STOYAN H., LISP Programmier-Handbuch, Akademie Verlag, Berlin, DDR, 1978

WEISSMAN C., LISP 1.5 Primer, Dickenson Publishing Company Inc., Belmont, California, 1967

WINSTON P.H., Artificial Intelligence, Addison Wesley, 1977

WRITE MORE LISP BOOKS)

Book review

JOHN ALLEN, The Anatomy of LISP,
McGRAW HILL, New York, 1977.

At last there exists a textbook for the computer scientist that gives a self-contained and in-depth treatment of "LISP" in its broadest sense. ALLEN's exposition of LISP serves as a manifestation for fundamental ideas such as: structured-programing and step-wise refinement, abstract programs and abstract data, data-driven programing, proving properties of algorithms, programing language semantics, translator implementation etc. This text will enable the student to regain the perspective he possibly lost in studying isolated topics or never gained in other so-called "introductions". Augmented by projects, "The Anatomy of LISP" is the best introduction to computer science I have seen.

After introducing symbolic expressions and a few LISP applications (mainly in algebraic manipulation), a series of languages together with their evaluators is built. ALLEN starts out with the language of polynomials, and a function to compute their value, and then proceeds step by step adding all the features usually found in programing languages, until advanced constructs like Label, closures and non-recursive control-structures. Each time the language is altered, so is its evaluator. The evaluators are rather concise and leave implementation issues for refinement to a special chapter. The semantics of a newly introduced construct is precisely defined by the new interpreter/evaluator. ALLEN's presentation strategy breaks the habit by which we acquire natural language constructs, namely through perceiving their usage in context. Breaking this habit is a healthy first step towards understanding the creation of new or more general language constructs. The working of these evaluators is clearly visualized using a representation of environments also known from WEIZENBAUM's explanation of FUNARGs [WEI].

The chapter on implementation covers solutions to the common problems and pitfalls encountered when implementing high-level programing languages. The discussion of binding is very profound and can be recommended as a first reading before studying more specialized papers such as [B&W], [STE] etc.. Most of this chapter discusses topics also relevant to other areas of software engineering: symbol-tables, table-searching, storage management and syntax-directed Input/Output.

The book ends with a series of interesting projects for students. This collection could be augmented by more typical AI-problems which would provide motivation for applying the idea of language and evaluator design as a means for problem-solving.

[B&W] Bobrow, D. and Wegbreit, B. A model and stack implementation of multiple environments, CACM, 16, 591-603.

[STE] Steele, G.L. Jr., Macaroni is better than spaghetti, Proc. of the Symposium on Artificial Intelligence and Programing Languages, ACM, 1977, 60-66.

[WEI] Weizenbaum, J. The Funarg problem explained, Intern. Seminar on Adv. Progr. Syst., Jerusalem, 1968.

(reviewed by: Joachim Laubsch)

(CURRENT LISP THESIS

CARTWRIGHT R. Jr, Formal Semantics of LISP with Applications to Program Correctness, Stanford University, Artificial Intelligence Laboratory, AIM-257, January 1975

GREUSSAY P., Contribution a la definition interpretative et a l'implementation des lambda-langages, These d'Etat, Universite Paris 7, Novembre 1977, Rapport L.I.T.P #7

LECOUFFE P., Etude et Definition d'une Machine Langage LISP, These de 3eme Cycle, Universite de Lille, Decembre 1977

LUX A., Etude d'un modele abstrait pour une machine LISP et de son implementation, These de 3eme Cycle, Universite de Grenoble, Mars 1975

NEWEY M. C., Formal Semantics of LISP with Applications to Program Correctness, Stanford University, Artificial Intelligence Laboratory, AIM-257, January 1975

SAMET H., Automatically Proving the Correctness of Translations Involving Optimized Code, Stanford University, Artificial Intelligence Laboratory, AIM-259, May 1975

TERASHIMA M., Algorithms Used in an Implementation of HLISP, Information Science Laboratory, Faculty of Science, University of Tokyo, January 1975

WERTZ H., Un Systeme de Comprehension, d' Amelioration et de Correction de Programmes Incorrects, These de 3eme Cycle, Universite Paris 6, Juillet 1978

WRITE MORE LISP THESIS)

(ANNOUNCEMENTS FOR LISP SYSTEMS

R.R. JOHNSON
Dept. of Computer Science
University of Kentucky
Lexington, Kentucky 40506

We are putting together a LISP-system on our Varian 74 mini-computer. Our primary interest is to use the micro-programming capabilities of the machine to implement some of the most used parts of the LISP-system.

S.S. MUCHNICK
Dept of Computer Science
The University of Kansas
Lawrence, Kansas 66045

An implementation effort for LISP 1.5 is now underway on an Interdata 85. The interpreter is being written in PL/85, a locally developed structured assembly language. Though the implementation is being done on a model 85 it is compatible with other Interdata 16-bit processors. The basic interpreter, storage management, and input/output packages are complete and we are currently extending our collection of SUBRs. We intend to implement the modified interpreter described in P. Greussay, Iterative Interpretation of Tail-Recursive LISP Procedures, as well.

Announcement

A New LISP System for IBM System 360/370

Over the last eighteen months we have developed in Cambridge, England a new IBM specific LISP system, with a number of advanced features. These include a built-in pretty printer for echoing of the input, which is also available to the user to print list structures. It has extensive and comprehensive tracing, backtrace and error recovery facilities, and is designed to be safe. Errors such as taking the car of an atom are trapped immediately, a feature which is also available in compiled functions.

The system has extensive numerical features, such as arbitrary precision integers and rational numbers, as well as double precision floating point numbers and finite field functions, (arithmetic modoulo a prime).

The compiler is a version of the Griss and Hearn portable LISP compiler, which produces compiled functions as a normal LISP object held in the heap. To manage this there is an efficient compacting garbage collector.

The system is value cell LISP, and is implemented in a high level language, (BCPL). It runs interpretively in 150Kbytes, and with the compiler in about 220Kbytes. Measurements have shown it to be efficient, (a little slower than Stanford LISP/360 when interpreted, faster when compiled) and it has been running successfully under OS in Cambridge for nine months. For further information contact one of:

Dr. John Fitch and Dr. Arthur Norman,
The Computer Laboratory,
University of Cambridge,
Corn Exchange Street,
Cambridge,
England.

14.1.77

JPF, ACN.

LISP for Interdata M85, 7/32

G. Persch, Gg. Winterstein

At the University of Kaiserslautern we have designed and implemented a LISP-system ID-LISP which runs on Interdata M85 and 7/32 minicomputers.

The Interdata M85 machine is a byte-oriented minicomputer with 64 K-byte working storage. It has 16 registers à 16 bit. The average time for an executable instruction is about 1 µs. The machine configuration at Kaiserslautern is for interactive use only and consists of a Hewlett Packard 264oA terminal and a Diabolo 162o for I/O, 4 disks and an Intertape Cassette unit. ID-LISP was designed in a way that it can run under the primitive operation system BOSS.

ID-LISP contains LISP 1.5 as a subset and has special features from MACLISP and INTERLISP as well. ID-LISP has its own EDITOR, PRETTYPRINT, file-handling (LOAD-SAVE) and various TRACE- and BREAK-functions. In the current version there is no paging.

The storage allocation is as follows:

hex.address	contents	size
oooo - 17oo	operating system BOSS	5.75 K-byte
17oo - 2EOO	programm-code	5.75 K-byte
2Eoo - 38oo	I/O-Buffer Hash-tables	2.5 K-byte
38oo - E8oo	11 K-LISP-cells	44. K-byte
E8oo - FFFF	stack	6. K-byte
		64. K-byte

4 bytes form a LISP-cell. Both pointers are absolute addresses. All parts of a symbol are represented in LISP-cells. For every symbol we have

(<list of bindings><function-definition><Pname><property>...)

Symbols are respresented only once. They are identified through a hash-table with Add-the-Hash-Rehash. Numbers are also stored within the LISP-storage. Small numbers (14 bit) are directly represented in the pointer. The garbage-collection-algorithm is a modification of P. Deutsch's pointer-chasing-algorithm.
At the time being the system is only available on cassettes. There is also a manual available which is written in German. For more details and further informations contact the autors at

Universität Kaiserslautern
Fachbereich Informatik
Pfaffenbergstr. 95

D-675o Kaiserslautern, FRG

The use of LISP at computer centers in Western Germany

(A summary of G.GÖRZ "Die Verwendung von LISP an wissenschaftlichen Rechenzentren in der BRD", IAB Nr 63, Universität Erlangen-Nürnberg, Rechenzentrum, Dez. 76).

LISP-systems as used on various computer systems

Computer	LISP-System	Installation
Burroughs B6700	LISP B 6700	Inf., Karlsruhe
CDC : CD 3300	LISP 1.5	U Erlangen
	LISP FINT	U Erlangen
	LISP F 1.1	U Gießen
		U Tübingen
CDC : CYBER bzw. 6000	LISP 1.5.6	TU Berlin
		RRZN Hamburg
	LISP 1.5.9	U Köln
	UTLISP 4.0	TU Berlin
		U Stuttgart
CGK : TR440	B&LISP (1973)	GMD Darmstadt
		U ERlangen
		Inf. München
		U Saarbrücken
		U Tübingen
		U Ulm
	B&LISP (1976)	GRZ, Berlin
	LISPSYSTEM	Inf. München
	LISP 440	IMMD Erlangen
	MACLISP	U Bielefeld
		U Bochum
		U Erlangen
		U Kaiserslautern
		Inf. Stuttgart
DEC : SYSTEM 10	LISP 1.6 (II)	Inf. Hamburg
	LISP 1.6 (28.7.)	U Kiel
IBM 360, 370	LISP/360 (Stanford)	U Bielefeld
		U Bonn
		MPI, Garching
		KfA, Jülich

	LISP/360 (Stanford/36, Utah-Mod.)	GfK, Karlsruhe U Münster
	LISP 1.5/CMS (Grenoble)	Inf.,TU Berlin
	LISP (bits)	GMD, St. Augustin
	LISP FINT	U Bonn
	LISP F2	U Heidelberg
Interdata M85, 7/32	ID-LISP	U Kaiserslautern
Philips Electrologica X8	LISP-X8	U Kiel
	LISP	U Regensburg
Siemens 4004 (BS1000)	LISP F2	IdS, Mannheim
Siemens 4004/151 (BS2000)	INTERLISP	IdS Mannheim
Univac 1108	1100 LISP	U Freiburg GWD, Göttingen U Karlaruhe

Applications

theorem-proving
    GRZ-Berlin, U. Kaiserslautern
program-manipulation
    GMD-St. Augustin, Informatik-Karlsruhe
natural-language processing
    Inf.-Hamburg, U-Heidelberg, U-Köln,
    Inst. f. deutsche Sprache (IdS), Inf. Stuttgart
cognitive psychology
    Phychol. Inst. Uni Hamburg, Inf. Stuttgart
others: Computer-aided instruction, REDUCE, nuclear physics.

The report contains a brief description of each LISP-system and contact addresses for further reference. To obtain it write to G.Görz, RZ d Uni. Erlangen-Nurnberg, INFRA, Martenstr. 1, 8520 Erlangen.

LISP-Reference Manuals (in german)

(1) INTERLISP - Programmierhandbuch
Institut für deutsche Sprache
Abt. Linguistische DV,
Postfach 5409, D6800 MANNHEIM 1

(2) MACLISP - Reference Manual
J. Laubsch, Inst. f. Informatik
Azenbergstr. 12, D7000 Stuttgart 1

Jerome CHAILLOUX
Universite de Paris VIII - Vincennes
Route de la Tourelle
75571 Paris Cedex 12 (France)

A VLISP System for 8-bit words micro-computers.

A new version of the VLISP system, VLISP 8, has been implemented on the 8-bit micro-computers
- Intel 8080
- Zilog 80

VLISP 8 is available for the following systems :
- MDS ISIS 1 and 2 (a 8080 based system), with the minimal configuration :
    - 32k RAM
    - a teletype
    - one floppy disk
- MOSTEK DDT80 (a Z80 based system), with the minimal configuration :
    - 16k RAM
    - 8k REPROM
    - a teletype

The interpreter occupies 8k bytes. Special attention has been paid to speed up the evaluation of forms. In particular, the interpreter does not perform any internal CONSes.
Naturally, this system owns all the new features recently introduced into the other VLISP systems, like ESCAPE LESCAPE SELF ... , as well as the iterative interpretation of tail-recursive functions calls.
This system is already used to introduce children to program in VLISP and a LOGO-like language.

SEND MORE ANNOUNCEMENTS FOR LISP SYSTEMS)

(CURRENT LISP MANUALS

CHAILLOUX J., VLISP-10 Manuel de Reference, Dept. Informatique, Universite Paris-8 Vincennes, RT 17-76, 1976

DURIEUX J.-L., TLISP IRIS 80, Universite Paul Sabatier, Toulouse, 1977

GREUSSAY P., LISP T-1600 Manuel de Reference Provisoire, Universite Paris-8 Vincennes, RT 10-75, 1975

HARALDSON A., LISP-details INTERLISP / 360-370, Uppsala University, 1975

LAUBSCH J.H., MACLISP Manual, CUU-Memo-3, Universitaet Stuttgart, 1976

LUX A., LISP IRIS-80 Manuel d'Utilisation, february 1978, Universite de Grenoble, LA CNRS no 7

MOON D.A., MACLISP Reference Manual, M.I.T. Project Mac, Cambridge, Mass., 1974

QUAM L.M. & DIFFIE W., Stanford LISP 1.6 Manual, Stanford AI Project Operating Note 28.7, Computer Science Dept., Stanford University, 1972

TEITELMAN W., INTERLISP Reference Manual, XEROX Palo Alto Research Center, Palo Alto, Ca., 1974

SEND MORE CURRENT LISP MANUALS)

INTERLISP - Programmierhandbuch

In deutscher Sprache liegt auf 340 Seiten

- ein bewährtes Lehr- und Ausbildungswerk vor, das besonders zum Selbststudium geeignet ist und das

- ein Nachschlagewerk darstellt, das für alle, die INTERLISP benutzen, unentbehrlich ist.

Inhalt:

0. LISP und INTERLISP
1. Die Syntax von LISP
2. Die Arbeitsweise von INTERLISP
3. Grundfunktionen
4. Funktionen und Programme
5. Funktionen mit funktionalen Argumenten
6. Ein/Ausgabe in LISP
7. Datentypen und zugehörige Funktionen
8. Spezielle Leistungen

Anhang A: EDIT
Anhang B: BREAK
Anhang C: Verzeichnis der beschriebenen Funktionen

Kontaktadresse: Institut für deutsche Sprache
Abteilung Linguistische Datenverarbeitung
Forschung und Entwicklung
Friedrich-Karl-Str. 12, Postf. 5409

6800 Mannheim 1
B R D

(USEFUL FUNCTIONS

A DEFINITION OF THE INVERSE QUOTE FUNCTION : @

Joachim LAUBSCH

Institut fur Informatik
Universitat Stuttgart
Azenbergstr. 12
7000 Stuttgart 1

It frequently happens that a LISP programmer wants a function to produce a data-structure or function containing constant and variable substructures. The usual solution is to program a form containing a lot of data-structure composing functions (like LIST, CONS and APPEND). The resulting expression is hard to decipher for humans unless more mnemonic constructor-functions are defined. A simple way out is to write the resulting structure with its variable substructures especially marked.

For example, return a lambda-expression which will evaluate any form in an environment where X and Y are bound to successive elements of L, and F receives the result :

```
(list 'lambda '(form)
      (append (list (list 'lambda '(X Y)
                          (list F '(eval form)))
                    L)))
```

Compare this with the inverse quote version :

```
   @(lambda (form)                ;  @ is inverse quote macro
       ((lambda (X Y)
           (=F (eval form)))      ;  = means eval the following
                                  ;  element
        ,L))                      ;  , is as = but uses all
                                  ;  elements of the value as
                                  ;  elements in the current list
```

As a second example, consider how WOODS could have returned structures without using BUILDQ

```
(S =TYPE =SUBJ (TNS= TNS) (VP (V= V)))
```

The following is the MACLISP code for @ (with the qu*1 borrowed and improved from a trace package of the MIT-AI-Lab). The function RPLACQ replaces a CONS. Feel free to include other macros using READMAC.

```
(READMAC MACRO
(NLAMBDA (F)
    ; TURNS MACRO-CHARS ON WHILE DOING THING;
    ; (READMAC <A-LIST> <THING-TO-DO>);
    ; <A-LIST> HAS PAIRS OF CHAR AND (QUOTED) FUNCTION;
    (COND
     ((CADR F)
      (LET
       (CHAR (CAAADR F) FN (CDAADR F))
       (RPLACO
        F
        'LET
        @
        ((SYNTAX (STATUS SYNTAX =CHAR)
                 FNTYP
                 (FNTYP '=CHAR)
                 OLD
                 (GET '=CHAR FNTYP))
         (PROG2 (SSTATUS MACRO =CHAR =FN)
                (READMAC = (CDADR F) , (CDDR F))
                (FUNCALL 'SSTATUS 'SYNTAX '=CHAR SYNTAX)
                (AND FNTYP (PUTPROP '=CHAR OLD FNTYP))
                ; SET STATUS AND RESET AFTERWARDS;)))))
    ; DO THING WITH READ OR READLIST;
    ((RPLACO F (CAADDR F) (CDADDR F))))))


(LET MACRO
(NLAMBDA (F)
    (COND ((CADR F) (RPLACA F 'LET1))
          ((CDDDR F) (RPLACO F 'PROGN (CDDR F)))
          ((RPLACO F (CAADDR F) (CDADDR F))))))


(LET1 MACRO
(NLAMBDA (F)
    ((LAMBDA (V)
         (COND
          ((NULL (CDDR V))
           (RPLACO F
                   (CONS 'LAMBDA (CONS (LIST (CAR V)) (CDDR F)))
                   (LIST (CADR V))))
          (V (RPLACO
              F
              (CONS 'LAMBDA
                    (CONS (LIST (CAR V))
                          (LIST
                           (CONS 'LET1 (CONS (CDDR V) (CDDR F))))))
              (LIST (CADR V))))))
     (CADR F))))
```

```
(QU* MACRO
(NLAMBDA (X)
    ; LISTS WITH EV OR EV* ARE EVALUATED;
    ; AND THEIR RESULTS WILL BE CONSED;
    ; OR APPENDED RESPECTIVELY;
    ((LAMBDA (Y) (RPLACO X (CAR Y) (CDR Y))) (QU*1 (CADR X)))))

(QU*1 EXPR
(LAMBDA (X)
    (COND
     ((NULL X) NIL)
     ((ATOM X) (LIST 'QUOTE X))
     ((EQ (CAR X) 'EV) (CADR X))
     ((OPTIM
       (COND
        ((ATOM (CAR X))
         (LIST 'CONS (LIST 'QUOTE (CAR X)) (QU*1 (CDR X))))
        ((EQ (CAAR X) 'EV*)
         (LIST 'APPEND (CADAR X) (QU*1 (CDR X))))
        ((LIST 'CONS (QU*1 (CAR X)) (QU*1 (CDR X)))))))))))

(OPTIM EXPR
(LAMBDA (X)
    ; ELIMINATES UNNECESSARY FN-CALLS;
    (SELECTQ (CAR X)
       (CONS
        ; (CONS X (LIST ---)) => (LIST X ---);
        (COND
         ((CADDR X)
          (AND (EQ (CAADDR X) 'LIST)
               (SETQ X (CONS 'LIST (CONS (CADR X) (CDADDR X))))))
         ((SETQ X (LIST 'LIST (CADR X))))))
       (APPEND
        ; (APPEND X (APPEND ---)) => (APPEND X ---);
        (COND
         ((CADDR X)
          (AND (EQ (CAADDR X) 'APPEND)
               (SETQ X (CONS 'APPEND (CONS (CADR X) (CDADDR X))))))
         ((SETQ X (CADR X)))))
       NIL)
    (AND (CATCH (MAPC '(LAMBDA (ARG)
                        (COND ((ATOM ARG) (THROW NIL))
                              ((EQ (CAR ARG) 'QUOTE))
                              ((THROW NIL))))
                   (CDR X)))
         (SETQ X (LIST 'QUOTE (EVAL X)))
         ; F IS-IN (APPEND CONS LIST);
         ; (F 'A 'B ---) => 'VALUE;
         ; WHERE VALUE = (EVAL (F 'A 'B ---));)
    X))


                                         SEND MORE USEFUL FUNCTIONS)
```

(TECHNICAL NOTES

A VLISP Interpreter on the VCMC1 Machine

May 1977

Jerome CHAILLOUX

Universite de Paris VIII - Vincennes
Route de la Tourelle
75571 Paris Cedex 12 (France)

VCMC1 is a virtual machine designed to observe "in vitro" the behaviour of VLISP interpreters. VCMC1 is actually entirely simulated in VLISP 10. We present a short description of the VCMC1 machine followed by the complete listing of the code of a VLISP interpreter. This interpreter incorporates the special feature for tail-recursion function calls.

Basically VCMC1 is a 16 bits machine. An instruction uses one, two, three or four words, and has one, two or three operands. Each operand is coded on a 4 bits field.

There are two formats for the instructions :

3 operands instructions :

```
         [op. code , 1st operand , 2nd operand , 3th operand]
bits      15 .. 12   11   ..   8   7    ..   4   3    ..   0
```

2 operands instructions :

```
         [op. code              , 1st operand , 2nd operand]
bits      15                   8   7    ..   4   3    ..   0
```

specification of the operands

AX	register 0. Holds sometimes the result of complex instructions (e.g. GET). It is used as index register.
A1	register 1.
A2	register 2.
A3	register 3.
A4	register 4.
A5	register 5.
A6	register 6.

LINK	register 7.
PC	register 8. Is the program counter.
ST	register 9. Is the stack pointer.
TST	the top of the stack.
+TST	the top of the stack after incrementation of the stack pointer.
TST-	the top of the stack. The stack pointer is decremented after the computation of the effective address.
(value)	a 16 bits value enclosed in parenthesis. This value is stored in the word just following the instruction.
(@ . address)	the value contained in the location specified. Used to denote indirection. The address is stored in the word following the instruction.
NIL	the atom NIL itself.

These operands allow several kinds of addressing :
- direct on registers AX A1 A2 A3 A4 A5 A6 LINK PC ST
- indirect on SP
- auto-increment on SP and PC
- auto-decrement on SP
- auto-increment indirect on PC.

Terminology and notation

r1	effective address of the 1st operand
r2	effective address of the 2nd operand
r3	effective address of the 3th operand
r' -> r''	move the content of the effective address r' into the word of effective address r''
r' <-> r''	exchange the contents of the effective addresses r' and r''
(CAR r)	the CAR of the effective address r
(CDR r)	the CDR of the effective address r
act1 & act2	denotes the overlap of the two actions

The instruction set

Instructions described are only those used by the interpreter listed below.
The effective addresses of the two or three operands are computed before the execution of the instructions, except in the case of conditional jumps.

transfers of data

(MOVE r1 r2)   r2 -> r1

A VLISP Interpreter on the VCMC1 Machine

(EXCH r1 r2)	r2 <-> r1
(MOVD r1 r2 r3)	r1 -> r2 & r2 -> r3
(CAR r1 r2)	(CAR r2) -> r1
(CDR r1 r2)	(CDR r2) -> r1
(RPLACA r1 r2)	r2 -> (CAR r1)
(RPLACD r1 r2)	r2 -> (CDR r1)
(MPUSH r1 r2)	r1 -> +TST ; if r2 # NIL, r2 -> +TST
(MPOP r1 r2)	TST- -> r1 ; if r2 # NIL, TST- -> r2

Transfers and branches

(MOVR r1 r2)	r2 -> r1 & TST- -> PC
(MOVJ r1 r2 r2)	r2 -> r1 & r3 -> PC
(MOVC r1 r2 r3)	r2 -> r1 & PC -> +TST & r3 -> PC
(CARR r1 r2)	(CAR r2) -> r1 & TST- -> PC
(CDRR r1 r2)	(CDR r2) -> r1 & TST- -> PC
(RPLACAR r1 r2)	r2 -> (CAR r1) & TST- -> PC
(RPLACDR r1 r2)	r2 -> (CDR r1) & TST- -> PC

Unconditional branches

(JUMP r1)	r1 -> PC
(JUMPX r1 r2)	r1 + r2 -> PC
(CALL r1)	PC -> +TST & r1 -> PC
(RETURN)	TST- -> PC

Conditional branches

(JEQ r1 r2 r3)	if r1 = r2 then r3 -> PC
(JNEQ r1 r2 r3)	if r1 # r2 then r3 -> PC
(JTNIL r1 r2)	if r1 = NIL then r2 -> PC
(JTNIL r1 r2)	if r1 # NIL then r2 -> PC
(JTLIST r1 r2)	if r1 is a pointer on a list then r2 -> PC
(JFLIST r1 r2)	if r1 is not a pointer on a list then r2 -> PC
(JTNUMB r1 r2)	if r1 is a pointer on a number then r2 -> PC
(JFNUMB r1 r2)	if r1 is not a pointer on a number then r2 -> PC

Other instructions

(UNCONS r1 r2 r3)	(CAR r1) -> r2 & (CDR r1 -> r3)
(CONS r1 r2 r3)	(r2 . r3) -> r1
(GET r1 r2)	(GET r1 r2) -> AX

Syntax of the assembler

The VLISP simulator handles lists of VCMC1 instructions, in which atomic elements are labels. It is possible to abbreviate instructions which look like

```
      (opcd op1 op2 (label))
```

into `(opcd op1 op2 . label)`
In order to decrease the size of such a list.

```
(PROGN
  (SETQ -INTERPRETER '(

         ;-------------------------------------------------------;
         ;                  V C M C    1                         ;
         ;                                                       ;
         ;                VLISP  interpreter                     ;
         ;-------------------------------------------------------;

         ;     binding of arguments for standard functions       ;
         ;                                                       ;
         ; 1SUBR : A1 <- value of the 1st argument               ;
         ; 2SUBR : A2 <- value of the 1st argument !             ;
         ;         A1 <- value of the 2nd argument !             ;
         ; NSUBR : A1 <- list of values of all the arguments     ;
         ; FSUBR : A1 <- list of all the arguments non-evaluated ;


         ; TOP-LEVEL function ;

TOPLEVEL
         (MOVC A1 ('TOPLEVEL) . PRINTA1) ; (WHILE T              ;
         (CALL . READ)                   ;    (PRINT 'TOPLEVEL)  ;
         (CALL . EVAL)                   ;    (PRINT             ;
         (MOVJ +TST (TOPLEVEL) . PRINTA1);       (EVAL (READ)))) ;

         ; PRINTA1 : because PRINT is a system NSUBR ;

PRINTA1  (CONS A1 A1 NIL)
         (JUMP . PRINT)

          ;------------------------------;
          ; functions of the interpreter ;
          ;------------------------------;

          ; GETFN : recognizes the type of the function stored in A2        ;
          ; call : (MOVJ A6 PC . GETFN) i.e. return address in A6           ;
          ; result in AX :                                                  ;
          ; AX <- 1 if 1SUBR the address of the function is stacked         ;
          ; AX <- 2 if 2SUBR      "                 "          "            ;
          ; AX <- 3 if NSUBR      "                 "          "            ;
          ; AX <- 4 if FSUBR      "                 "          "            ;
          ; AX <- 5 if LAMBDA   ((lvar) ... body ...) is stacked            ;
          ; AX <- 6 if FLAMBDA  "                 "             "           ;
          ; AX <- 7 if GAMMA    "                 "             "           ;
          ; don't destroy  A1 ! ;

 GETFN    (JTLIST A2 . GETFN5)
                                         ; the function is an atom ;
          (GET A2 ('EXPR))               ; is it an EXPR ? ;
          (JTNIL AX . GETFN1)            ; no ;
          (MOVJ A2 AX . GETFN)           ; yes : retry with the new expression ;
 GETFN1   (GET A2 ('TYPFN))              ; is it a standard function ? ;
          (JTNIL AX . GETFN3)            ; no ;
          (MOVJ +TST (*VAL* [A2]) A6)    ; yes : stack the address and return ;
 GETFN3   (CAR A2 A2)                    ; indirection on the value of the atom ;
          (JUMP . GETFN)
 GETFN5                                  ; the function is a list ;
          (UNCONS A2 AX +TST)            ; stack ((lvar) ... body ...) ;
          (JNEQ AX ('LAMBDA) . GETFN6)   ; is it a LAMBDA ? ;
          (MOVJ AX ('5) A6)              ; yes : value = 5 and return ;
 GETFN6   (JNEQ AX ('FLAMBDA) . GETFN7)  ; is it a FLAMBDA ? ;
          (MOVJ AX ('6) A6)              ; yes : value = 6 and return ;
 GETFN7   (JNEQ AX ('GAMMA) . GETFN8)    ; is it a GAMMA ? ;
          (MOVJ AX ('7) A6)              ; yes : value = 7 and return ;
 GETFN8   (MOVE TST A6)
          (MPUSH A1)                     ; in others cases ;
          (MOVC A1 A2 . EVAL)            ; re-evaluate the function ;
          (MOVD A2 A1 TST-)
          (MOVJ  A6 TST- . GETFN)

          ; EVAL : 1SUBR          A1 <- the forme to be evaluate  ;
          ; APPPLY : 2SUBR        A1 <- the list of values ready  ;
          ;                       A2 <- function to apply         ;

 EVALCAR (CAR A1 A1)                     ; (EVAL (CAR A1)) ;
 EVAL     (JTLIST A1 . EVAL1)            ; in case of a list ;
          (JTNUMB A1 TST-)               ; numbers are not evaluated ;
 QUOTE    (CARR A1 A1)                   ; the value of an atom is
                                           its C-value (i.e. its CAR) ;
 EVAL1    (UNCONS A1 A2 A1)              ; A1 <- the function,
                                           A2 <- the list of arguments ;
          (JEQ A1 ('QUOTE) . QUOTE)      ; special case for the QUOTE function ;
          (MOVJ A6 PC . GETFN)           ; find the type of the function ;
          (JUMPX AX ((EVALCAR)(EVAL2)(EVLIS) TST- (EVAL3)(APPLYF)(EVAL4)))
                      ; 1SUBR   2SUBR  NSUBR  FSUBR LAMBDA FLAMBDA GAMMA ;
```

```
EVAL2   ; for the 2SUBRs ;
        (UNCONS A1 A1 +TST)
        (CALL . EVAL)                   ; evaluate the 1st argument ;
        (EXCH A1 TST)
        (CALL . EVALCAR)                ; evaluate the 2nd argument ;
        (MOVR A2 TST-)

EVAL3   ; evaluation of LAMBDA-expressions ;
        (MOVJ +TST (APPLYL) . EVLIS)

EVAL4   ; evaluation of GAMMA-expressions ;
        (MOVJ +TST (APPLYG) . EVLIS)

APPLYC  (CONS A1 A1 NIL)                ; used by mapping functions ;
APPLY   (MOVJ A6 PC . GETFN)            ; get the type of the function ;
        (JUMPX AX ((CAR)(APPLY2) TST- TST- (APPLYL)(APPLYG)(APPLYG)))
                ; 1SUBR 2SUBR   NSUBR FSUBR LAMBDA FLAMBDA GAMMA ;
APPLY2  (UNCONS A1 A2 A1)
        (CARR A1 A1)
APPLYG  (CAR A1 A1)
        (JUMP . APPLYL)
APPLYF  (CONS A1 A1 NIL)
                                        ; APPLYL must follow ... ;

        ; general for LAMBDA/FLAMBDA/GAMMA       ;
        ; suppose : A1 <- list of values ready   ;
        ;           TST <- ((lvar) ... body ...) ;

APPLYL  (UNCONS TST- A2 A3)             ; A2 <- lvar, A3 <- body. ;

        ; test of tail-recursion ;

        (JNEQ (@ . TST) ('*TR*) . APPLYN) ; it is not in terminal position ;
        (JNEQ (@ . LINK) A3 . APPLYN)     ; it is not a recursive function ;

        ; special binding for tail-recursive function ;

REBIND  (JFLIST A2 . REBIND2)           ; larg is atomic ;
REBIND1 (UNCONS A2 A5 A2)               ; A5 <- new variable ;
        (UNCONS A1 (@ . A5) A1)         ; force the new value ;
        (JTLIST A2 . REBIND1)           ; variables left ? ;
REBIND2 (JTNIL A2 . PROGNA3)            ; real end of lvar ;
        (MOVJ (@ . A2) A1 . PROGNA3)    ; in case of LEXPR ;

        ; normal binding with preservation of the old values ;

APPLYN  (MPUSH LINK ('MARKER))          ; special mark in stack ;
BIND1   (JFLIST A2 . BIND2)
        (UNCONS A1 A4 A1)               ; A4 <- next value ;
        (UNCONS A2 A5 A2)               ; A5 <- next variable ;
        (MOVD +TST (@ . A5) A4)
        (MOVJ +TST A5 . BIND1)
BIND2   (JTNIL A2 . BIND3)              ; real end of lvar ;
        (MOVD +TST (@ . A2) A1)
        (MPUSH A2)
BIND3   (MPUSH A3)

        ; execution of the body of the function ;

        (MOVC LINK ST . PROGNA3)
*TR*    (MOVJ A6 PC . UNBIND)
        (RETURN)

        ; unbind the previous bindings ;

UNBIND  (MOVJ A5 TST- . UNBIND2)
UNBIND1 (RPLACA A5 LINK)
UNBIND2 (MPOP A5 LINK)
        (JNEQ A5 ('MARKER) . UNBIND1)
        (JUMP A6)
```

```
         ;----------------------;
         ; control functions ;
         ;----------------------;

         ; PROGN : FSUBR,  EPROGN : 1SUBR ;
         ; allows to handle the tail-recursive functions ;

PROGNA3 (MOVE A1 A3)                     ; internal (PROGN A3) ;
EPROGN
PROGN    (UNCONS A1 A1 A2)               ; next element ;
         (JFLIST A2 . EVAL)              ; there is one element ;
PROGN1   (MOVC +TST A2 . EVAL)
         (UNCONS TST- A1 A2)             ; next element ;
         (JTLIST A2 . PROGN1)            ; it is not the last element ;
         (JUMP . EVAL)                   ; it is the last element ;

         ; LIST :  FSUBR,  EVLIS : 1SUBR ;

LIST
EVLIS    (JFLIST A1 TST-)                ; nothing to do ;
         (CONS A2 NIL NIL)               ; prepare the head of the result ;
         (MPUSH A2)                      ; which is saved in the stack ;
                                         ; A2 is also the address of the
                                           last CONS-cell ;
EVLIS2   (UNCONS A1 A1 +TST)             ; next element ;
         (MOVC +TST A2 . EVAL)           ; save the remainder and
                                           evaluate the element ;
         (CONS A2 A1 NIL)                ; CONS the value ;
         (MPOP A3 A1)                    ; restore last and the remainder ;
         (RPLACD A3 A2)
         (JTLIST A1 . EVLIS1)            ; list not exhausted ;
         (CDRR A1 TST-)


         ; LESCAPE : FSUBR  ;
         ; allows to force a tail recursion ;

LESCAPE (MOVJ +TST (*TR*) . PROGN)


         ; IF : FSUBR. The most simple conditionnal function ;
         ; allows to handle the tail-recursive functions ;

IF       (UNCONS A1 A1 +TST)
         (CALL . EVAL)                   ; evaluate the predicate ;
         (UNCONS TST- A2 A3)
         (JTNIL A1 . PROGNA3)            ; else clauses ;
         (MOVJ A1 A2 . EVAL)             ; then clause ;


         ; COND : FSUBR. The most famous CONDitionnal function ;
         ; allows to handle the tail-recursive functions ;

COND     (MOVE A2 A1)
COND1    (JFLIST A2 TST-)                ; no more clauses ;
         (UNCONS A2 A1 +TST)             ; A1 <- next clause ;
         (UNCONS A1 A1 +TST)             ; A1 <- the predicate ;
         (CALL . EVAL)                   ; evaluate it ;
         (MPOP A3 A2)
         (JTNIL A1 . COND1)              ; the predicate is false ;
         (JFNIL A3 . PROGNA3)            ; evaluate the clause ;
         (RETURN)                        ; the clause is empty ;


         ; OR AND : FSUBR, logical connectors ;
         ; allows to handle the tail-recursive functions ;

OR       (UNCONS A1 A1 A2)
         (JFLIST A2 . EVAL)              ; the last element ;
         (MOVC +TST A2 . EVAL)
         (JFNIL A1 . PRET)
         (MOVJ A1 TST- . OR)

AND      (JFLIST A1 . TRUE)              ; (AND) -> T ;
AND1     (UNCONS A1 A1 A2)
         (JFLIST A2 . EVAL)              ; the last element ;
         (MOVC +TST A2 . EVAL)
         (JTNIL A1 . PRET)
         (MOVJ A1 TST- . AND1)

PRET     (MOVR A2 TST-)                  ; pop and return ;


         ; WHILE : FSUBR ;

WHILE    (MOVJ +TST A1 . WHILE2)         ; stack the whole expression ;
WHILE1   (CDR A1 TST)
         (CALL . PROGN)
WHILE2   (MOVC A1 TST . EVALCAR)         ; evaluate the test ;
         (JFNIL A1 . WHILE1)             ; it is ready for an other turn ;
         (MOVR A2 TST-)                  ; finish ;
```

```
        ;---------------------------;
        ; predicates and searches ;
        ;---------------------------;

        ; NULL NOT ATOM NUMBP LISTP : 1SUBR ;

NULL
NOT     (JTNIL A1 . TRUE)
        (MOVR A1 NIL)
ATOM    (JFLIST A1 . TRUE)
        (MOVR A1 NIL)
NUMBP   (JTNUMB A1 . TRUE)
        (MOVR A1 NIL)
LISTP   (JTLIST A1 . TRUE)
        (MOVR A1 NIL)

        ; EQ NEQ : 2SUBR ;

EQ      (JEQ A1 A2 . TRUE)
        (MOVR A1 NIL)
NEQ     (JNEQ A1 A2 . TRUE)
        (MOVR A1 NIL)

        ; EQUAL NEQUAL : 2SUBR ;

NEQUAL  (MPUSH . NOT)
EQUAL   (MOVC A6 ST . EQUAL2)             ; prepare A6 for fast return ;
        (MOVR A1 ('T))
EQUAL1  (JFLIST A2 . NAN)
        (UNCONS A1 A1 +TST)               ; cdr down A1 ;
        (UNCONS A2 A2 +TST)               ; cdr down A2 ;
        (CALL . EQUAL2)                   ; recurse on CAR ;
        (MPOP A2 A1)
EQUAL2  (JTLIST A1 . EQUAL1)              ; iterate on CDR ;
        (JEQ A1 A2 TST-)
NAN     (MOVJ ST A6 . FALSE)              ; fast return ;

TRUE    (MOVR A1 ('T))
FALSE   (MOVR A1 NIL)

        ; CAR CDR : 1SUBR ;

CAR     (CARR A1 A1)
CDR     (CDRR A1 A1)

        ; GET : 2SUBR ;

GET     (GET A2 A1)
        (MOVR A1 AX)

        ; MEMQ : 2SUBR ;

MEMQ1   (JEQ (@ . A1) A2 TST-)
        (CDR A1 A1)
MEMQ    (JTLIST A1 . MEMQ1)               ; the list is empty ;
        (RETURN)




        ;--------------------;
        ; create and modify ;
        ;--------------------;

        ; MAPC : 2SUBR, the position of the arguments is non-standard ;

MAPC    (EXCH A1 A2)                      ; A1 <- list of arguments
                                            A2 <- function ;
        (JFLIST A1 TST-)                  ; nothing to do ;
MAPC1   (UNCONS A1 A1 +TST)
        (MOVC +TST A2 . APPLYC)
        (MPOP A2 A1)
        (JTLIST A1 . MAPC1)
        (RETURN)
```

```
        ; MAPCAR : 2SUBR ;

MAPCAR  (EXCH A1 A2)                    ; A1 <- list of arguments,
                                          A2 <- function ;
        (CONS A3 NIL NIL)
        (MOVJ +TST A3 . MAPCAR2)
MAPCAR1 (MPUSH A3)
        (UNCONS A1 A1 +TST)             ; next argument ;
        (MOVC +TST A2 . APPLYC)         ; save the function ;
        (CONS A3 A1 NIL)
        (MPOP A2 A1)
        (RPLACD TST- A3)
MAPCAR2 (JTLIST A1 . MAPCAR1)
        (CDRR A1 TST-)

        ; RPLACA RPLACD : 2SUBR ;

RPLACA  (RPLACA A2 A1)
        (MOVR A1 A2)
RPLACD  (RPLACD A2 A1)
        (MOVR A1 A2)

        ; SETQ : FSUBR ;

SETQ    (UNCONS A1 +TST A2)             ; stack the name ;
        (UNCONS A2 A1 +TST)             ; stack the remainder ;
        (CALL . EVAL)                   ; evaluate the value ;
        (MPOP A2 A3)
        (RPLACA A3 A1)                  ; set the new value ;
        (JFLIST A2 TST-)                ; no more couple ;
        (MOVJ A1 A2 . SETQ)

        ; SET : NSUBR,  SETQQ : FSUBR ;

SETQQ
SET     (UNCONS A1 A2 A1)
        (UNCONS A1 (@ . A2) A1)
        (JTLIST A1 . SET)
        (MOVR A1 (@ . A2))

        ; NEXTL : FSUBR ;

NEXTL   (CAR A2 A1)                     ; A2 <- atom ;
        (CAR A3 A2)                     ; A3 <- its value ;
        (UNCONS A3 A1 A3)
        (RPLACAR A2 A3)

        ; CONS : 2SUBR ;

CONS    (CONS A1 A2 A1)
        (RETURN)

        ; REVERSE : 2SUBR ;

REV1    (UNCONS A2 A3 A2)
        (CONS A1 A3 A1)
REVERSE (JTLIST A2 . REV1)
        (RETURN)

) ; end of the SETQ  -INTERPRETER ; )
```

```
; initialisation of the indicators of the standard functions ;

(MAPC
   '(ATOM CAR CDR EPROGN EVAL EVLIS
     LISTP NULL NUMBP)
   '(LAMBDA (X) (PUT X 1 'TYPFN)))

(MAPC
   '(APPLY CONS EQ EQUAL GET MAPC MAPCAR
     MEMQ NEQ NEQUAL REVERSE RPLACA RPLACD)
   '(LAMBDA (X) (PUT X 2 'TYPFN)))

(MAPC
   '(PRINT PRIN1 TERPRI READ SET)
   '(LAMBDA (X) (PUT X 3 'TYPFN)))

(MAPC
   '(AND COND IF LESCAPE LIST NEXTL OR
     PROGN QUOTE SETQ SETQQ WHILE)
   '(LAMBDA (X) (PUT X 4 'TYPFN)))

   '-INTERPRETER)
```

A SYSTEM TO UNDERSTAND INCORRECT PROGRAMS

Harald Wertz

Universite de Paris VIII (Vincennes)
route de la tourelle
75571 Paris

Abstract :
This paper presents a system (PHENARETE) which understands and improves incompletely defined LISP programs, such as those written by students beginning to program in LISP. This system takes, as input, the program without any additional information. In order to understand the program, the system meta-evaluates it, using a library of "pragmatic rules", describing the construction and correction of general program constructs, and a set of "specialists", describing the syntax and semantics of the standard LISP functions. The system can use its understanding of the program to detect errors in it, to debug them and, eventually, to justify its proposed modifications. This paper gives a brief survey of the working of the system, emphasizing on some commented examples.

A lot of effort is actually spent on the developpement of tools to help programmers in constructing, debugging and verifying programs. Unfortunately most of these tools
- impose too much constraints on the intuitions of the programmer [cf DIJKSTRA 1976],
- are working only on a very limited subset of possible programs [cf RUTH 1974],
- are only working on correct programs [cf ARSAC 1977, IGARASHI et al 1975].

Our aim is two-fold :

-1- to make explicit the knowledge involved in constructing and debugging programs and

-2- to verify - not the correctness of programs - but their "consistency" and to provide "hints" for improving and correcting their programs to the programmer.

To this end we have build our system on four main concepts :
1 an algorithm of meta-evaluation [cf GOOSSENS 1978] to help the system to understand each of the possible paths of the program,
2 a set of "specialists", i.e. a set of procedural specifications of the syntax and the operational semantics of the standard LISP functions,

example : specialist CAR for the syntax

```
[CAR-1 (X) =>
        v (& atom (CAR X)
           & type (X) = LISTP)
        v (& S-expression (CAR X)
           & type (val (X)) = LISTP)
  else :
           modify X until CAR-1 (X) = T]
```

paraphrasing :

```
CAR expects that its argument is
   - an atom
     and the type of  the  value  of  the
     argument is a list
   - a S-expression
     and the type of the  value  of  that
     S-expression is a list
  else
     CAR has to modify the argument until
     one of these two conditions is true
```

and the specialist CAR for the semantics

```
[CAR-N =>
      arg : (X (meta-eval X))
     test : (type (val (X)) = LISTP) ->
            (type (val (X)) = ?)
                  -> hypothesize (X, type LISTP)
            T -> complain (X, type : LISTP)
   action : if (existe (CAR X)) --> (CAR X)
            else (create (CAR, X)) --> (CAR X)]
```

or in paraphrasing :

```
CAR-N

   has an argument named X, which must be evaluated
   one must verify
     if
        the type of value of the argument is a
        list, all is ok
     else
        if the type of value of the argument isn't
        known, one has to create a hypothetical
        value of type LIST for X
     else
        one has to ask the debugger to change the
        text of the program in such a way that the
        value of X becomes a list

   the value of CAR is
     if
        there exists already a CAR of X, this CAR
     else
        one has to create a symbolic value for X,
        the CAR of which will be the desired value
```

These specialists are the agents of the meta-evaluation and they represent the system's knowledge about the programming language used.

3 a set of "pragmatic rules" describing general program constructs and methods to repair inconsistencies

example :

rule of the dependence of a loop of the predicate =>

if no variable of the exit-test is modified inside the loop, then the loop is independent of the exit-test and, its execution is non-terminating or the loop will never be executed.

The set of these rules expresses the system's general knowledge about the well-formedness of programs and about the correction of errors;

4 during the analysis of a program, PHENARETE contructs some description – an internal representation – of the program under the form of "cognitive atoms". These may be considered as the nodes of a network-like representation of the program actually analysed.

The system accepts every LISP program conforming to the following restrictions :

- partitioning of the names of variables, funtions and labels;
- all funtion calls must be "call by name";
- the unique functional arguments admitted are explicit lambda-expressions.

We call this subset of LISP : extended first order LISP.

To use PHENARETE, the user has to give to the system only the text of the draft version of the program he wants to write, without any additional information like input/output assertions, commentaries, plans etc. The system will try to understand what the user wanted to do, and, if necessary, modify the text of the programm.

To give some feeling of the working of the system, let us examine some examples in detail :
Our first example is a (very) erroneous version of the well known REVERSE function. Here is the actual input to the system :

```
?  (P '(DE REV L1 L2 COND ULL L22 A1 T RVE A1 ONS CRA A1 A2))
```

PHENARETE will first correct the spelling errors :

```
   ERREUR:
                NOM --> (? ULL --> NULL)
   ERREUR:
                NOM --> (? L22 --> L2)
   ERREUR:
                NOM --> (? A1 --> L1)
   ERREUR:
                NOM --> (? A1 --> L1)
   ERREUR:
                NOM --> (? RVE --> REV)
   ERREUR:
                NOM --> (? RVE --> REV)
   ERREUR:
                NOM --> (? A1 --> L1)
   ERREUR:
                NOM --> (? ONS --> CONS)
```

```
ERREUR:
                NOM --> (? CRA --> CAR)
ERREUR:
                NOM --> (? A1 --> L1)
ERREUR:
                NOM --> (? A2 --> L2)
```

After having very well corrected the spelling errors, PHENARETE proceeds to a first analysis where she uses only her syntactic knowledge. The result of this first analysis is a "syntactically correct" LISP program (i.e. a programm accepted by any smart LISP interpreter or compiler) :

PROPOSITION 1 :

```
(DE REV (L1 L2)
    (COND
        ((NULL L2) L1)
        (T (REV L1 (CONS (CAR L1) L2)))))
```

These first improvements have eliminated all the syntactic errors. Anyway, there subsist two semantic errors :

-1 in the recursive call of rev, the first argument L1 is not modified. This creates an infinit recursion.

-2 even with a modification of L1 in the recursive call, the recursion won't stop either since the stop-test has as argument L2, a list which grows longer and longer in the run of the succesive recursive calls.

PHENARETE can not disambiguate this function - she does not know anything of the intentions of the programmer - so she gives two different propositions :

PROPOSITION 2 :

```
(DE REV (L1 L2)
    (COND
        ((NULL L2) L1)
        ((NULL L1) L2)
        (T (REV (CDR L1) (CONS (CAR L1) L2)))))
```

```
                AT LEAST YOUR FUNCTION SEEMS OK TO ME.
```

In this first proposition, PHENARETE supposed the stop-test given to be true, but that the user omitted a second stop-test for the case where the second argument is not NULL at the initial call of REV.

PROPOSITION :

```
(DE REV (L1 L2)
   (COND
      ((NULL L1) L2)
      (T (REV (CDR L1) (CONS (CAR L1) L2)))))
```

AT LEAST YOUR FUNCTION SEEMS OK TO ME.

In this second proposition, PHENARETE supposed that the user inadvertently inverted the arguments of the stop-test, so she inverts the two arguments L1 and L2.
Of the two corrected versions of the initial draft-program PHENARETE is assured that they will stop and deliver a result when running.

Our second example is an extremly "simplified" version of the equally well known function FACTORIAL. Here she is :

```
? (DE FACT N TIMES N FACT N)
```

As in the previous example, PHENARETE will first translate this unparentesized expression into an well parentesized one :

PROPOSITION 1 :

```
(DE FACT (N) (TIMES N (FACT N)))
```

This first proposition is a syntactically correct program, but semantically it is not very correct :

-1 at the recursive call N is not modified. This is the same kind of error as in the previous example, exept the argument here is of numeric type.

-2 there is no stop-test at all, so there are two (!) reasons to make the recursion infinit.

Remember that PHENARETE doesn't know the intentions of the programmer, so she must detect these errors without any additional information : all she can use in the further analysis are the semantic specialists and the pragmatic rules.
So let us look at her proposition :

PROPOSITION :

```
(DE FACT (N) (COND
   ((LE N 0) 1)
   (T (TIMES N (FACT (SUB1 N))))))
```

AT LEAST YOUR FUNCTION SEEMS OK TO ME.

This corrected version is actually a correct version of the factorial-program. The performance is really astonishing knowing that the system works completely automatically whithout asking any question to the user and without any information about the supposed intention.

One last (uncommented) example :

```
?  (DE ADDIT M N  ((ZEROP N) M)
                  (T (ADDIT SUB1 M ADD1 N)))

    PROPOSITION 1 :

   (DE ADDIT (M N)
       (COND
          ((ZEROP N) M)
          (T (ADDIT (SUB1 M) (ADD1 N)))))


    P R O P O S I T I O N :
   (DE ADDIT (M N)
       (COND
          ((ZEROP N) M)
          ((LE M 0) N)
          (T (ADDIT (SUB1 M) (ADD1 N)))))

                 AT LEAST YOUR FUNCTION SEEMS OK TO ME.
```

Presently we are working on some extensions as to find automatically the intentions and the goals of given pieces of code. We would also like to adjoin to PHENARETE a module permitting to explain the reasoning of the system. This would be a great help to the user.
The system is running on PDP-10, uses about 25k word memory, is implemented in VLISP [CHAILLOUX 1976, GREUSSAY 1977], and is used by about 1000 students in our university.
A more detailed description may be found in [WERTZ 1978].

references

ARSAC J., (1977), La construction de programmes structures, Dunod-Informatique, Paris

CHAILLOUX J., (1976), VLISP-10 manuel de reference, Dept. Informatique, Universite Paris 8, RT-17-76

DIJKSTRA E.W., (1976), A Discipline of Programming, Prentice-Hall, Inc., Englewood Cliffs, N.J.

GOOSSENS D., (1978), A System For Visual-Like Understanding of LISP Programs, Proc. AISB/GI Conference, Hamburg, RFA, July 17-19, 1978

GREUSSAY P., (1977), Contribution a la Definition Interpretative et a l'Implementation des Lambda-langages, These, Universite Paris 7.

IGARASHI S., LONDON R.L. & LUCKHAM D.C., (1975), Automatic Program Verification 1 : Logical Basis and its Implementation, Acta Informatica, vol. 4, pp. 145-182.

RUTH G.R., (1974), Analysis of Algorithm Implementations, M.I.T., MAC-TR-130.

WERTZ H., (1978), Un systeme de comprehension, d'amelioration et de correction de programmes incorrects, These de 3eme cycle, Universite Paris 6.

ITERATIVE INTERPRETATION OF TAIL-RECURSIVE LISP PROCEDURES

Patrick GREUSSAY

Departement d' Informatique
University of Vincennes

September 1976

ABSTRACT

The design of a LISP interpreter that allows tail-recursive procedures to be interpreted iteratively is presented at the machine-language level. Iterative interpretation means that, without any program transformations, no environments and continuations will be stacked unless necessary.

We apply a specific modification within a traditional stack-oriented version of LISP interpreter, without any non-recursive control structure. The design is compatible with value-cells as well as a-lists LISP processors.

We present a complete modified interpreter written itself in LISP and an informal proof that it meets its requirements.

1.0 INTRODUCTION

It is well-known that tail-recursive procedures (TR for short) are formally equivalent to iterative procedures [1]. Unfortunately, when interpreted, TR code behaves as ordinary recursive code, and we do not obtain the benefit of the formal equivalence. We need a way to make the formal equivalence also a practical one.

The problem of the computation of TR procedures can be stated in terms of program transformations, or in terms of interpreters specially designed to handle them properly.

Static program transformations [4,10] from TR to iterative programs are mainly used on compiler-oriented LISP systems, and therefore do not allow full access to interpreter-oriented tools of debugging, breaks, traces etc. Moreover, as they are name-sensitive, they cannot handle procedures with circular types.

Interpreter-oriented processing of TR procedures uses non-recursive control structures like message-passing as in Hewitt's ACTORS system[7,8] or generalized FUNARG devices with static a-lists, as in Sussman's SCHEME [12]. Also delayed evaluation of CONS [6] has been proposed in which partial building of a structure is triggered by invocations to the decomposition primitives CAR and CDR applied to this virtual structure.
As interesting as they are, there do not seem to be any evident ways to adapt such methods to ordinary recursive stack-oriented LISP interpreters.

In contrast, we propose a simple way to process TR procedures iteratively, without any program transformations or non-recursive control structures. Our scheme can be used with a-lists as well as value-cells stack-oriented LISP interpreters.

2.0 TAIL RECURSIVE SCHEMATA

In [1], a TR schema in iterative form is defined as a recursion equation

f(x1,...,xn) = g(f,x1,...,xn,h1,...,hm)

where g is a conditional expression defining f in terms of the functions h1,...,hm; g is said to be iterative if f occurs exactly in terms of g in the form THEN f(...) or ELSE f(...).
For example, this is the recursive form of the well-known program for addition (NOTE 1)

```
(de plus (x y) (if (= x 0) y
                   (add1 (plus (sub1 x) y))))
```

and this is the corresponding iterative form

```
(de plus (x y) (if (= x 0) y
                   (plus (sub1 x) (add1 y))))
```

NOTE 1 : (if c e1 e2 ... en) is the LISP equivalent to the form
if c *then* e1 *else* e2; ... ; en *fi* .

We must generalize the previous definition of iterative schema to any named λ-expression such that

(1) f = (λ(x1...xn) ... (f a1...an)) (NOTE 2)

i.e. the last term of the λ-expression body is a call of this λ-expression.

(2) f = (λ(x1...xn) ... (if c (f a1...an) ...))
and
f = (λ(x1...xn) ... (if c e1 e2 ... (f a1...an)))

i.e. the THEN-part or the last term of the ELSE-part of an if-form is a call of this λ-expression.
Nested if-forms are valid under this schema, e.g.

f = (λ(x1...xn) ... (if c1 (if c2 ... (if cm (f a1...an) ...) ...)))

(3) f = (λ(x1...xn) ... (cond ... (c e1 ... em-1 (f a1...an)) ...))

Note that the condition c, even if it consists solely of the constant T, must be mentionned explicitely, in contrast with for example INTERLISP [13] style of writing CONDs.

3.0 RETURN CONTINUATIONS

We notice that these schemata share a common property: all of them are instances of forms interpreted by the LISP system internal function PROGN.

These forms will be interpreted recursively if PROGN is defined as follows.
Let us suppose the variable EXP is the name of a register which contains the form to be evaluated and the result after the evaluation. TEMP is a working register, SAVE and RESTORE push and pop respectively their argument onto a stack, REC pushes a return continuation (NOTE 3), and UNREC restores the current continuation from the stack.

```
PROGN = temp←exp;
        while not(null(temp))
              do save(temp); exp←car(temp);
                 rec EVAL; temp←restore();
                 temp←cdr(temp)
              od
        unrec();
```

NOTE 2: This part of the definition means that we allow non-terminating procedures.

NOTE 3: Here we use the continuation concept [11] the same way as in [9], where a continuation is just a list of instructions to be executed.

On the contrary no return continuation will be stacked at an instance of a TR call of our iterative λ-expressions if PROGN is designed as:

```
PROGN = while not(null(cdr(exp)))
                do save(exp); exp←car(temp);
                   rec EVAL; exp←restore();
                   exp←cdr(exp)
                od
           exp←car(exp); jumpTo EVAL;
```

The last clause of a PROGN argument (a list of expressions to be evaluated) will be passed directly to EVAL, which obtains the definitive control of the continuation.

If LISP TR procedures had no arguments, the interpreter would automatically handle calls of forms like

```
f = ( λ() (if c e1 e2 ... en-1 (f)))
```

as

```
while not c do eval(e2); ...; eval(en-1) od;
eval(e1);
```

Then the program writer could design its procedures in the most natural way, without paying attention to what are necessary or unnecessary recursions [14]. But, as LISP procedures generally use argument passing, we need a way to omit unnecessary saving of environments (NOTE 4) by the internal LISP system function APPLY, in the case of TR procedures.

4.0 THE HANDLING OF ENVIRONMENTS

A redundant environment is defined as a new environment caused by the call of a TR procedure, the old environment being unnecessarily saved, in spite of the fact that it will be never used again.

To avoid redundant environments, we have to modify APPLY in the following manner.
When APPLY has discovered that it must handle a λ-expression, first it examines the stack at a definite place to see if the same λ-expression has been called before. If this is the case, it does not save the current environment onto the stack, and just binds every variable of its formal arguments list to its value, then gives the body of the λ-expression to PROGN. If this is not the case, then APPLY saves the current environment (which means that the λ-expression is called for the first time, as far as APPLY can see) onto the stack, then saves the list which represents the λ-expression being called, then builds a new environment as before, and finally saves a return continuation to a part of APPLY which restores environments. The control is then given to PROGN.

The definite element which APPLY examines is then the next-to-last item saved in the stack.
This part of APPLY can be designed in the following manner (the field CVAL being the value-cell of LISP variables) :

```
PART-OF-APPLY = if car(exp)=λ then
                   if STACK[TOP-1]=exp then
                      for-each x in cadr(exp) do
                               x.CVAL←car(arglist);
                               arglist←cdr(arglist)
                               od
                      exp←cddr(exp); jumpTo PROGN
                   else
                      save(sentinel);
                      for-each x in cadr(exp) do
                               save(x.CVAL); save(x);
                               x.CVAL←car(arglist);
                               arglist←cdr(arglist)
                               od
                      save(exp); exp←cddr(exp); rec PROGN;
                      restore();
                      while STACK[TOP]≠ sentinel do
                            x←restore();
                            x.EVAL←restore()
                      od
                      restore(); unrec()
                   fi
                fi
```

NOTE 4: By environment is meant an association of variables with values. In LISP, saved environments can take the form of a-lists, where the value of every variable is found (with the possibility of multiple instances of the same variable), or can take the form of value-cells, in which case the value associated with the variable is unique and immediately accessible, old associations being saved on the stack.

5.0 EXAMPLES OF TR-PROCEDURES

Here are some examples to illustrate programming in TR style, with the modified interpreter.

(1) An iterative Ackermann function :

```
(de ack (x y) (a x y nil))

(de a (x y p) (cond
                  ((= x 0) (if p (a (car p) (add1 y) (cdr p))
                                 (add1 y)))
                  ((= y 0) (a (sub1 x) 1 p))
                  (T (a x (sub1 y) (cons (sub1 x) p)))))
```

(2) Building a list of factorials :

```
(de factlist (n) (g n 1 (list 1)))

(de g (n x r)
    (if (= x n) r
           (g n (add1 x)
                (cons (times (add1 x) (car r)) r))))
```

(3) Building a factorial procedure with circular type :

```
(setq g '((λ (x y f)
              (if (= x 0) y
                     ((car f) (sub1 x) (times x y) f)))))
```

to obtain factorial n we call `((car g) n 1 g)`

Our modified interpreter appears to be "name-insensitive" and can run this example iteratively, which cannot be handled by program transformations.

(4) Notice that forms like

```
((λ (x) (x x)) '(λ (x) (x x)))
```

being run iteratively do not cause overflow of the stack.

6.0 THE MODIFIED INTERPRETER

Here is the complete modified LISP interpreter (NOTE 5). It is written itself in LISP in the machine language style of Sussman's SCHEME [12]. We use a global environment and we do not use any recursive features.

```
(de run () (setq pc 'toplevel) (loop))

(de loop () (while t (apply pc nil)))
```

We use a non-terminating control loop which runs the "next" procedure, in which the non-modified LISP internal function *apply* is called over and over again. The variable pc plays the role of a program counter.
Here is the top-level loop :

```
(de toplevel () (setq link nil stack nil) (save 'top1)
                (setq exp (read) pc 'eval))

(de top1 () (print exp) (setq pc 'toplevel))
```

Here are the "pipe-lined" procedures eval, evlis and apply :

```
(de eval () (cond
        ((numberp exp) (unrec))
        ((atom exp) (setq exp (car exp)) (unrec))
        (T (setq hdexp (car exp) exp (cdr exp) pc 'eval1))))

(de eval1 () (cond
         ((listp hdexp) (save hdexp) (setq pc 'evlis))
         ((or (get hdexp 'expr) (get hdexp 'subr))
          (save hdexp) (setq pc 'evlis))
         ((setq temp (get hdexp 'fexpr))
          (setq arglist (list exp) exp temp pc 'apply))
         ((get hdexp 'fsubr) (setq pc hdexp))
         (T (setq hdexp (car hdexp)))))


(de evlis () (setq built nil pc 'evlis1))

(de evlis1 () (if (null exp)
                  (setq arglist (reverse built) exp (restore) pc 'apply)
                  (save exp) (save built) (save 'evlis2)
                  (setq exp (car exp) pc 'eval)))

(de evlis2 ()
        (setq built (cons exp (restore)) exp (cdr (restore)) pc 'evlis1))
```

NOTE 5 : This interpreter is in fact a simplification of the one running under the name of VLISP at the University of Vincennes [2,5] on a PDP 10 and a T1600 computer.

```
(de apply () (cond
        ((atom exp) (cond
              ((setq temp (or (get exp 'expr) (get exp 'fexpr)))
               (setq exp temp))
              ((or (get exp 'subr) (get exp 'fsubr))
               (setq pc exp))
              (T (setq exp (car exp)))))
        ((or (eq (car exp) (setq temp 'λ))
             (eq (car exp) (setq temp 'γ)))
         (setq λ-γ-exp exp)
         (if (eq temp 'γ) (setq arglist (car arglist)))
         (if (eq (cadr stack) λ-γ-exp)
             (rebind (cadr λ-γ-exp) arglist)
             (bind (cadr λ-γ-exp) arglist) (save λ-γ-exp) (save 'apply3))
         (setq exp (cddr λ-γ-exp) pc 'progn))
        (T (save arglist) (save 'apply2) (setq pc 'eval))))
```

A form (γ(x1...xn) e1 e2 ... en) is like a λ-expression but when applied to an argument which is a list, it distributes the elements of this list over the formal arguments x1...xn . This is very efficient for handling multiple values recursive procedures.

```
(de apply2 () (setq arglist (restore) pc 'apply))

(de apply3 () (restore) (unbind) (unrec))
```

The internal procedure *unbind* restores old environments, *rebind* simply builds a new environment without saving the current one in contrast to *bind* which saves the old environment.

Next we come to the "sequencer" procedure *progn* :

```
(de progn () (if (cdr exp) nil (save exp) (save 'progn1))
             (setq exp (car exp) pc 'eval))

(de progn1 () (setq exp (cdr (restore)) pc 'progn))
```

Finally, as an illustration of control procedure of FSUBR type, we give the code for *if* :

```
(df if () (save exp) (save 'if1) (setq exp (car exp) pc 'eval))

(de if1 () (if exp (setq exp (cadr (restore)) pc 'eval)
                   (setq exp (cddr (restore)) pc 'progn)))
```

7.0 CHECKING-RULES

We must now devise a means of insuring that our interpreter meets its requirements. We shall use "checking-rules" of the form

S1{P1}P2:S2

where S1 is the state of the stack when entering procedure P1, S2 is the state of the stack when leaving procedure P1, and P2 is the name of the next procedure to enter. When P2 is the label "retcont" it means that P1 has no next procedure to enter, so a recursive return to the head of the stack has to be performed.

Examination of the interpreter yields the following rules, which constitute in a sense an abstract version of the interpreter, the enter and exit states of the stack playing the role of an history [3].

```
α{eval}retcont:α ∨ eval1:α
α{eval1}evlis:hdexp:α ∨ apply:α ∨ fsubr:α ∨ eval1:α

hdexp:α{evlis}evlis1:hdexp:α
hdexp:α{evlis1}eval:evlis2:built:exp:hdexp:α ∨ apply:α
built:exp:hdexp:α{evlis2}evlis1:hdexp:α

α{apply}apply:α ∨ subr:α ∨ fsubr:α
                  ∨ progn:α ∨ progn:apply3:λ-γ-exp:oldbindings:α
                  ∨ eval:apply2:arglist:α
arglist:α{apply2}apply:α
λ-γ-exp:oldbindings:α{apply3}retcont:α

β{progn}eval:progn1:exp:β ∨ eval:β
exp:β{progn1}progn:β

β{if}eval:if1:exp:β      (NOTE 6)
exp:β{if1}eval:β ∨ progn:β
```

Using the checking-rules, we can show that the interpreter handles TR programs correctly.

let foo = (λ (x1 ... xn) e1 ... em)

with em = (foo a1 ... an) .

First of all we must show that the state of the stack is the same when evaluating em and when entering progn with exp = (e1 ... em) .

NOTE 6 : Recall that *if* is of FSUBR type.

Let exp = (e1 ... em) with α{progn}.

Suppose m = 1, we have

α{progn}eval:α and therefore α{eval}.

If m > 1 we have

α{progn}eval:progn1:exp:α

and if the evaluation of the head of exp does not enter into an infinite loop, we shall obtain

exp:α{progn1}progn:α

followed by

α{progn} , now with the length of exp being m-1 □.

Further, we must show that when

apply3:foo:oldbindings:α{eval}

then exp = em , i.e. it is only when evaluating em that we can find foo as the next-to-top item in the stack.

Suppose exp = ek with k ≠ m. The state of the stack when entering eval will be

progn1:(ek ... em):β{eval}

and (ek ... em) being a tail of foo cannot be equal to foo □.

Finally we must show that, if there is not an infinite loop when evaluating one of the ei , 1 ⩽ i ⩽ m, the old bindings will be restored.

As before, if exp = em, with

apply3:foo:oldbindings:α{eval}

when eval returns, the state of the stack is

foo:oldbindings:α{apply3}retcont:α

and apply3 restores the environment immediately preceding the first call of foo □.

8.0 CONCLUDING REMARKS

We have proposed a LISP interpreter in which TR code behaves at run-time as efficiently as well-written iterative code, with the extra benefit of avoiding explicit side-effects as well as manual or automatic program transformations.
Another advantage is that it is insensible to renaming, e.g. if we have

```
(de foo (x) ... (foo (g x)))
```

and we perform

```
(put 'fie (get 'foo 'expr) 'expr)
```

then the call (fie a) will be interpreted exactly the same way as a call of foo.

The modification we have proposed does not depend on particular implementations of environments. This one more encouragement to write programs in recursive style, particularly since our modification can be applied very easily with no apparent drawbacks to any LISP interpreter.

Ed. Note: The whole thing has been improved since 1976. It can run the same way mutual co-recursive procedures, and also what I call "enveloped" tail-recursions, as in

```
(DE foo (x)
    (IF (ZEROP x) 0
        (+ x (foo (SUB1 x)))))
```

ACKNOWLEDGEMENTS

Many discussions and advices from J. CHAILLOUX have been very helpful, as well as the material help he provided during the preparation of this report.

REFERENCES

1. J. McCARTHY : "Toward a mathematical science of computation."
Proc. IFIP 1962 21-28

2. CHAILLOUX J. : "VLISP 10" RT 17-76
Computer Sc Dept. University of Vincennes. France.

3. M. CLINT : "Program Proving : Coroutines"
Acta Informatica 2, 50-63 (1973)

4. DARLINGTON J., BURSTALL R.M. : "A system which automatically improves programs." (1973)
Proc. of 3d International Joint Conference on Artificial Intelligence.
Stanford. 537-542

5. GREUSSAY P. : "Descriptions compactes d'interprètes implémentables."
Programming Symposium. Paris. Avril 1976.
B. Robinet ed. 281-297

6. FRIEDMAN D.P., WISE D.S. : "Output Driven Interpretation of Recursive Programs."
TR n°50. Indiana University. July 1976.

7. C. HEWITT : "Behavioral semantics of nonrecursive control structures"
Proc. Programming Symposium. Paris. 1974.
B. Robinet ed. Springer-Verlag. 385-407

8. C. HEWITT : "Viewing control structures as patterns of passing messages"
Working Paper 92. April 1976. MIT AI Lab.

9. REYNOLDS J.C. : "Definitional interpreters for higher-order programming languages."
Proc of 1972 ACM Nat. Conf. Boston. 1972. 717-740

10. RISCH T.: "REMREC : A Program for automatic recursion removal in LISP."
Uppsala University. 1973

11. STRACHEY C. : "A mathematical semantics which can deal with full jumps"
Séminaires IRIA "Théorie des algorithmes, des langages et de la programmation" ed. M. NIVAT. Mai 1973. 175-191

12. SUSSMAN G.J., STEELE Jr G.L. : "SCHEME : An interpreter for extended lambda-calculus."
MIT AI Lab. AI Memo n° 349. Dec 1975

13. TEITELMAN W. : "Interlisp Reference Manual."
XEROX Palo Alto Research Center. 1974

14. WIRTH N. : "Algorithms + Data Structures = Programs"
Prentice Hall. 1976

SEND MORE TECHNICAL NOTES)