Le_Lisp,a portable and efficient Lisp system

J. Chailloux, M. Devin, J.M. Hullot

INRIA

CENTRE DE ROCQUENCOURT

Rapports de Recherche

N° 319

LE_LISP, A PORTABLE AND EFFICIENT LISP SYSTEM

Jérôme CHAILLOUX
Matthieu DEVIN
Jean-Marie HULLOT

Juillet 1984

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(3) 954 90 20

LE_LISP, a Portable and Efficient LISP System

Jérôme Chailloux (*)
Matthieu Devin (**)
Jean-Marie Hullot (*)

(*) I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

(**) Centre de Mathématiques Appliquées
Ecole des Mines de Paris
Sophia-Antipolis
06370 Valbonne
France

Résumé: ce rapport décrit le système LE_LISP, développé dans le cadre du projet VLSI de l'INRIA, principalement conçu pour son efficacité, sa facilité de transport et la possibilité d'y construire des programmes importants. Ce rapport présente également l'environnement de programmation du système (dont une extension "langage objet", un meta paragrapheur, un générateur d'analyseurs, un éditeur universel ...) et quelques applications actuelles.

Abstract: This paper describes the LE_LISP system, developed at the VLSI project at INRIA, which has been designed for efficiency, easy transport and large systems construction. It also presents the programming environment (including an object oriented extension, a meta pretty printer, a parser generator, a universal editor) and some current applications.

LE_LISP, a Portable and Efficient LISP System

Jérôme Chailloux (*)
Matthieu Devin (**)
Jean-Marie Hullot (*)

(*) I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex
France

(**) Centre de Mathématiques Appliquées
Ecole des Mines de Paris
Sophia-Antipolis
06370 Valbonne
France

***Abstract:** This paper describes the LE_LISP system, developed at the VLSI project at INRIA, which has been designed for efficiency, easy transport and large systems construction. It also presents the programming environment (including an object oriented extension, a méta pretty printer, a parser generator, a universal editor) and some current applications.*

1. Why another LISP system ?

1.1. A bit of history

In 1981, the VLSI project of INRIA, directed by Jean Vuillemin, began the design of an ambitious VLSI workstation, using a unique structure for representing all aspects (graphic, simulation ...). We were convinced that LISP was a good implementation language for such a system, provided we could use one of the powerful *post MacLISP* LISPs. Happily (or sadly) hardware evolution was very fast and we inherited many different and incompatible machines. We were soon faced with the problem of transporting the system. The use of a real standard, portable LISP system became urgent. As most of our computers were running under UNIX, we could have chosen Franz-LISP [Foderaro&al81]; unfortunately, Franz-LISP had never been sucessfully implemented on any other machine but VAX. PSL [Griss82] seemed a real interesting system, but, in 1981, it was not yet available overseas. Common LISP [Steele&al82] was in limbo. Furthermore, we wanted the implementation of the LISP system on a new computer to be faster than the translation of our products from one LISP dialect to the other.

1.2. Main goals and issues

Portability: we decided to design a virtual machine, LLM3 [Chailloux83b], instead of using a high level language. Previous experiments had shown the inherent difficulties of combining easy transport and good performance; yet we were not willing to compromise on the latter point.

Efficiency: for the interpreter VLISP was a good base [Greussay77, Chailloux80]; for the compiler MacLISP was a reference [Moon74].

Environment comfort: The complete LE_LISP system includes the interpreter, the compiler, a lot of debugging tools, full screen text editors and provisions for calling external linked procedures. The CEYX programming environment provides a structured, multi-modes, multi-windows editor: BIGMACS [Hullot83].

Compatibility: we put a great care in the choice of functions names, thus avoiding any dangerous synonyms with the other dialects. Indeed, most LE_LISP programs run, through a set of macros, on *Mac LISP like* LISPs.

Extensibility: either at a machine level, with the portable virtual machine LLM3 or, obviously, at LISP level.

1.3. State of the project

The first LE_LISP system began to run during fall 1981 on an Exormacs (Motorola). Its performances and completeness allowed us to build a VLSI workstation, using colored bitmap and mouse [Levy82, Chailloux&al83]. The system was then implemented on a VAX during fall 82, under UNIX, on which we now continue its development. In 1983 we implemented the system on Perkin Elmer 32 (Feb 83), VAX/VMS (Jun 83), HB68/DPS8 Multics (Dec 83) and Intel 8088/8086 (May 84). The two last implementation required only one month of work from scratch: the system had become *truly* portable.

LE_LISP v15 [Chailoux84] currently runs on the following machines:

- VAX UNIX 4.1BSD, 4.2BSD, IS3, VMS
- MC68000 (Exormacs, Apollo, Metheus λ 750, Fortune 32, SM90, Universe 68-CRDS)
- Perkin Elmer 32 UNIX
- DPS8/HB68 Multics
- IBM PC
- IBM 3081 VSMAX

We forsee to have it implemented on:

- Ridge 32
- HP 9000 (UNIX)

- NS 16000 (VME hardware)
- DPS7
- Gould SEL 32 (MPX 32)
- Cray 1 (using FORTRAN 77)

A reduced but upward compatible version, LE_LISP80, runs on Intel 8080/Zilog Z80 under CP/M [Chailloux83a, Saint-James84]. This version is distributed in high-schools by the French Ministry of Education.

2. The evaluator, its originalities

We detail here some of the genuine facilities that our system offers.

2.1. Function arguments tree binding

As NIL [White79, Burke&al82], LE_LISP uses a complete destructuration mechanism, when binding arguments to functions of type EXPR, FEXPR, MACRO or DISPLACE-MACRO. The parameter list of these user functions is a generalized tree on which the actual argument list is matched at each call. For FEXPRs, MACROs and DISPLACE-MACROs, this tree matching is obviously performed between the non-evaluated list of arguments and the formal parameters list. For EXPRs we do not want to CONS a list of the values of arguments in order to compute the matching: it is useless CONSing for 'usual' argument lists. We rather compute the bindings of formal parameters in parallel with the values of arguments. In this way, the binding processus of classical argument lists is not much more expensive than the usual non-destructuring mechanism.

Tree matching performs as follow*

parameters	*arguments*	*bindings*
(DE f (a b c) ...	(f 1 2 3)	a=1 b=2 c=3
(DE f a ...	(f 1 2 3)	a=(1 2 3)
(DE f (a . b) ...	(f 1 2 3)	a=1 b=(2 3)
(DE f ((a b c)) ...	(f '(1 2 3))	a=1 b=2 c=3
(DE f ((a . b) c) ...	(f '(1 2) '(3 4))	a=1 b=(2) c=(3 4)

This mechanism allowed the elimination of the LEXPR functional type. LEXPR are now defined with:

* = means *binds to*

```
(DE foo l
        ....
    )
```

2.2. Powerful non local exit control functions

LE_LISP provides a MacLISP like mechanism to handle exceptions and non local exits. We call it TAG/EXIT. TAG allows to dynamically define a named exit block and EXIT allows to terminate a previously defined exit block of a same name. Tag names can be computed at runtime with the EVTAG and EVEXIT functions, thus providing a dynamic non local control of the evaluation.

This mechanism has proved to be very convenient for simple error handling in LISP sub-systems. One usualy defines its toplevel and error-handling functions the following way:

```
(DE toplevel ()
    (WHILE T
           (TAG user-error
                <top evaluation>)))

(DE error-handling (message)
    (PRINT message)
    (EXIT user-error 'error))
```

Any calls to the error-handling function, will unwind all functions calls up to the TAG control block in the toplevel function.

Although often sufficient this mechanism lacks of precision: there is no way to know if the exit of a TAG block is normal or abnormal. People often bypass this with the emission of an otherwise impossible result when the exit was abnormal. We propose a new feature to control non local exits: LOCK.

The LOCK function allows to build blocks in which any attempt to EXIT can be controlled. Its first argument is a function with two arguments: the exit-handler. The rest of the arguments, the body of the expression, is evaluated by PROGN. On any exit out of the body, the exit-handler is called with the tag name and exit value as arguments. If the evaluation of the body terminates normally with value *V*, the exit-handler is still called, this time with the arguments () and *V*. Thus any call to LOCK always terminates through the exit-handler.

With LOCK it is easy to define new control structures such as ON-EXIT-DO,

CATCH-ALL-BUT, IGNORE-EXITS, and so on.

Ignoring all exits form a given block may be written:

```
(LOCK '(LAMBDA (tag val) val)
      <block>)
```

Catching all exits but those in a given set:

```
(LOCK '(LAMBDA (tag val)
         (IF (MEMBER tag <set>) (EVEXIT tag val)
             val))
      <block>)
```

Evaluating $form_i$ on each exit tag_i, in an ADA like style:

```
(LOCK '(LAMBDA (tag val)
         (SELECTQ tag
             (()   val)
             (tag1 <form1>)
             (tag2 <form2>)
             ..
             (tagN <formN>)
             (t (EVEXIT tag val))))
       <block>)
```

Note the use of EVEXIT to exit again on EXITs that we did not want to catch.

2.3. Full name hierarchical packaging

All symbol names are packaged. Extensive use of name packaging allows to avoid all name collisions and local parameters capture. Packaging is performed by using a new symbol property: the packagecell. Packagecells holds atoms, whose packagecell in turn can be used to determine a hierachy of packages.

Packages can be declared at READing time with macro characters, or dynamically with a set of specialized functions. This allowed the implementation of the LOAD-IN-PACKAGE feature, to automatically hold symbols of separate modules, in private packages. UNLOAD-PACKAGE is also possible as all atoms of a given package can be marked as garbage, and then discarded at the next GC, together with their values, functional values and other properties.

2.4. Tail recursion elimination

In order to minimize the stack size during a recursive computation, some forms of tail recursion are detected by the interpreter and the compiler. This feature, that first appeared in VLISP [Greussay76], is implemented in LE_LISP and has been extended on LE_LISP80 by E. Saint-James who introduced the notion of "obsolete environment" which became rapidly crucial on small systems [Saint-James84].

This feature gives a very clear programming style, where recursion is preferred to classical loops.
For example, printing an infinite table of integer squares can be coded, with no risk of stack overflow:

```
(DE SQUARE (n)
    (PRINT "The square of "  n  " is "  (* n n))
    (SQUARE (1+ n)))
```

Tail recursion is also detected and unwinded within such control structures as IF, COND, OR etc.
An iterative computation of the factoriel function looks like:

```
(DE FACT (n) (FACT1 n 1))

(DE FACT1 (n result)
    (IF (=0 n) result
        (FACT1 (1- n) (* n result))))
```

2.5. User controlled I/O buffers

LE_LISP provides a buffered I/O mechanism on both user terminal and text files, and an independant raw output mechanism on terminal. Buffered I/O is controlled by software interrupts: conditions such as END-OF-FILE, BEGINNING-OF-LINE or END-OF-LINE, raise a call to a predefined LISP function. These functions can be redefined by the user to obtain the desired behaviour.

For example, in a big system that can READ things from files, the END-OF-FILE condition must be trapped at the toplevel, to perform such actions as switching input on an other channel, changing prompt or echo mode, re-initializing global variables and so on. To get this one needs only to redefine the EOF function in order to realize an exit up to the toplevel. User toplevel must catch this exit and perform the desired actions. This can be written:

```
(DE EOF (channel)
    (EXIT EOF channel))

(DE USER-TOP ()
    (WHILE T
        (ON-EXIT (EOF <actions>)
                 <top evaluation>)))
```

Where ON-EXIT is a macro that generates a call to LOCK.

Some other user-controled software interrupts are END-OF-LINE (EOL), called when the output buffer is full and flushing has to occur, and BEGINNING-OF-LINE (BOL) called on READing when input buffer is empty. EOL is extensively used by the pretty printer, and BOL can be used for specialized I/O, prompting, etc..

The raw output mechanism is mainly used for CRT terminals interface, for example in the full screen text editor PEPE.

2.6. External routines calling

The DEFEXTERN function allows to call external routines previously linked to the kernel. Those functions can receive arguments and return results. The data types of arguments are converted to fit the external representation, and the result is converted to LISP standards. Such routines may be used either to obtain side effects (bitmap display), to get information (mouse tracking), or for special library calls and numeric computations (fortran libraries, array operations).

For example the following expression defines SCALAR-PRODUCT as being an external LISP function with 2 arguments (that are to be converted as vectors) that returns a short integer.

```
(DEFEXTERN SCALAR-PRODUCT (VECTOR VECTOR)
           FIX)
```

The actual machine code of SCALAR-PRODUCT may have been written in C or FORTRAN (available languages depends on the system) and must have been linked to the LISP kernel.

This feature has been of great help on SM90 (MC 68000, UNIX) to get full control on the bitmap and mouse directly from the LISP system.

2.7. Efficiency

Despite the fact that these kinds of bench-marks do not measure the right things [Gabriel82], we give below timings for the computation of the expression *(fib 20)* where fib is defined as:

```
(DE FIB (n)
    (COND ((EQ n 1) 1)
          ((EQ n 2) 1)
          (T (+ (FIB (1- n)) (FIB (- n 2))))))
```

Using the fast fixnum operators.

All time are given in seconds of user cpu time. **Inter**, **comp** and **opt** are time for interpreted code, compiled code with still interpreter compatibility and optimized compiled code. **Ptr** is the pointer size, **cy** is the numbers of memory cycles required to move a pointer.

System	Computer	ptr(cy)	inter	comp	opt
LE_LISP	VAX 780 (UNIX/VMS)	32(1)	4.25	0.65	0.12
LE_LISP	68000 Exormacs	32(2)	12.0	--	0.56
LE_LISP	68000 MicroMega32	32(2)	15.8	--	0.54
LE_LISP	68000 SM90 (fast)	32(2)	8.5	--	0.31
LE_LISP	Perkin Elmer 32/50	32(1)	7.1	0.99	0.23
LE_LISP	Multics	18(1)	5.9	--	--
LE_LISP	IBM PC	16(2)	21.0	--	--
LE_LISP	IBM 3081 VSMAX	--	0.76	--	--
LE_LISP_80	Z80 3Mhz	16(2)	24.0	--	--
MacLISP	HB68/Multics	72(1)	13.5	0.38	--
Franz-LISP	Vax 780 UNIX	32(2)	16.1	2.8	--
LISP machine	Symbolic 3600	36(1)	29.0	--	0.15

We also give bench mark allowing a comparison with Franz-LISP on the VAX. We show computation times for the Fibonacci function, computed with by a semantic interpreter written in ML [Cousineau84], and for various unification algorithms.

System	FIB[15]	FIB[20]	unif1	unif2
LE-LISP	26.3	302.5	34.2	20.
Franz-LISP	37.8	308.7	52.1	28.3

We explain why our interpreter and our compiler are so efficient, developing three main points: the hardware optimizations, the function calls, and LISP data manipulation.

Hardware optimizations

The Kernel of the interpreter is fully written in the virtual machine language LLM3. The main LLM3 operations are pointer movements, pointers comparisons (for type checking), and stack manipulation (for both data saving and routine calling). They can be easily implemented as a set of assembler macros on any stock hardware.

When implementing LLM3 on a new computer we can make optimizations to get full benefit of the specific hardware of the machine. Those concern three main points: registers allocation, LISP datas manipulation and micro-coding.

Micro-coding of various crucial parts of the kernel will be realized on VAX and Perkin-Elmer. Those parts include the beginning of the EVAL function, where trace test and type dispatching take place, and the construction/destruction of the stack control blocks when calling user defined functions.

Function calling

Calling LISP functions must be a very fast processus, especially for SUBRs. LE-LISP introduces a more refined functional typology among predefined and compiled functions. With both fast access to functional value (FVAL), and functional type (FTYPE) of symbols, it yields very efficient function calls.

System functions (SUBR) divide into nine different types: SUBR0/1/2/3, SUBRV1/2/3, SUBRN and FSUBR. SUBRVs are functions that accept a variable but limited number of evaluated arguments, such as GET/SET functions and some compiled LISP functions. Their introduction was a great improvement, diminishing the number of FSUBRs. Except for SUBRNs that pick them in the stack, arguments will always be given in machine registers. Calling SUBR only requires dispatching on functional types towards specialized modules that perform arguments evaluation, and then give control to the function code.

Type dispatching is performed with very few machine instructions. Evaluating the form *(CAR a)* up to the entrypoint of CAR, only takes 12

LLM3 instructions (without the evaluation of α). They expand in 17 instructions on the VAX and 21 on the DPS8. These include checking of the trace state of the interpreter, of possible stack overflow and of the presence of any extra arguments.

User defined functions divide into four types: EXPR, FEXPR, MACRO and DISPLACE-MACRO. DISPLACE-MACROs are a sub-type of MACROs that physically modify the function call, after their first evaluation. Evaluation of user functions takes 5 steps:

- stack control block building
- parameters evaluation and arguments binding
- tail recursion test and casual unwinding
- body evaluation
- arguments unbinding

Tail recursion test is very cheap: it only costs 4 LLM3 instructions when negative. The building of the stack block and the binding of arguments are the most time consuming part of the processus. We optimized them with the introduction of direct stack adressing capabilities, in order to reduce the number of POPs and PUSHs.

LISP datas manipulation

The speed of the interpreter is greatly dependant on the efficiency of LISP datas manipulation, and typechecking.

In order to obtain a fast typechecking we divide the memory space into separate zones, each of them being dedicted to a single LISP type. Typechecking is thus a simple pointer comparison with the limits of the zones.

We provide zones for Symbols, CONSes, Strings, Vectors, Floating numbers, and for a HEAP in which we hold the contents of strings and vectors. LE_LISP also uses short fixnums (in the range -32768..32767) that are not boxed, and arbitrary precision rational numbers, implemented as trees of fixnums in the CONS zone [Vuillemin&al84]. For a faster access to standard properties of atoms (FVAL, FTYPE, CVAL, PNAME..) we do not implement them on a Plist but with record fields.

All boxed objects can be garbage collected, the HEAP zone being compacted when necessary. The standard garbage collector uses a *Sweep & Mark* algorithm; but we will develop a *Stop & Copy* mechanism for systems who provides virtual memory, such as VAX.

3. How LE_LISP was made portable

3.1. LE_LISP has proved easily portable

The LE_LISP system has been especially designed to lead an easy implementation on any machine. We show below the time required to implement the whole system on various machines. As the initial LE_LISP version was developed on MC68000/Exormacs, implementation time is not given for this machine.

Computer	time	where	date
Exormacs	--	INRIA	fall 1981
VAX UNIX	2 month	INRIA	fall 1982
Perkin Elmer 32	4 month	CMA	Feb '83
Fortune 32	2 weeks	INRIA	Mar '83
SM90	2 weeks	INRIA	Mar '83
Apollo	2 weeks	INRIA	Aug '83
Metheus	1 week	INRIA	Dec '83
HB68/Multics	1 month	CMA	Dec '83
IBM PC	3 weeks	Act	May '84
Universe 68	1 week	DD	May '84

The MC 68000 implementations (Micromega, SM90, Apollo, Metheus, Universe 68) times are short because most Exormacs work could be used. The time spent for the last implementations, on fully different machines, is significant of the present transportability state of LE_LISP.

3.2. Philosophy of implementation

Implementing LE_LISP on a new computer requires the implementation of the virtual machine language LLM3, and the coding of an initializer, and of a small runtime for special devices (bitmap, mouse). LLM3 is best implemented as a set of assembler macros, or cross expansed with an other LISP system; the initializer and runtime are often written in a high level language (C, PL/1).

Most implementations of LLM3 needs a run-time library providing such functions as file I/O and system interface. This library, quite small, is generaly written in a high level language (PL/1, C, Pascal).

For our implementations on UNIX systems, a C expanser and a C initializer were used. The MC68000 versions have been cross-expansed on our VAX. The Multics version used a Maclisp expanser, a PL/1 initializer and a PL/1 library. The implementation is fully described in [Devin84].

4. The compiler

We think a good LISP compiler must:
- guarantee *full compatibility* with the interpreter, especially for the binding of variables.
- provide, at least, a ten time performance improvement (this looks small, but the interpreter is fast !).
- be quick and small.

We insist on the first point, for it allows debugging under the control of the interpreter. With a very efficient interpreter, and LE_LISP has one, compilation is not anymore needed before evaluation, but must be regarded as a final improvement.

For easy implementation our compiler works in two passes. The first one produces LLM3 code, in a LAP form. The second one, performed by a machine-dependent loader, translate LLM3 to binary code, with peephole optimizations. The whole process of compilation takes place in the LISP environment. We call it *in-core* compilation. We allow separate compilation and loading of pre-compiled modules, although such features are less crucial, since the operations of loading and compilation are fast.

The compiler accepts optional declarations, but works well without any. It automatically computes the scope of local arguments and optimizes the allocation of LLM3 registers. Needless to say, the compiler and the loader, both written in LISP, can be compiled.

The use of a virtual machine language as an intermediary for compilation has proved very efficient: for large systems, such as the ML compiler [Cousineau84], LE_LISP is now about two times faster than Franz-LISP (VAX 11/780 UNIX).

5. The LE_LISP Programming Environment

A lot of work has been made to provide LE_LISP with a powerful programming environment. We overview some of its tools.

5.1. CEYX: an Object Oriented Extension

CEYX [Hullot83] is a LISP extension allowing to create and manipulate objects: objects are the combination of a record like structure with a set of semantical properties which are the basic actions that can be performed on such structures. As in Smalltalk [Goldberg82], Loops [Bobrow83] and LISP Machine Flavors [Weinreb&al81], objects are arranged by families in a hierarchical manner so that they inherit properties of their ancestors.

CEYX is itself written in CEYX: it has been bootstrapped. All primitive objects are thus made available to the user. It makes the system fully extensible by the user.

5.2. VPRINT: a Universal Pretty Printer

VPRINT [Hullot84] is a pretty printer, which can be compared to [Waters81]: it allows to specify fully a textual representation for any LISP or CEYX structure. It is implemented as a virtual printing machine (a CEYX object) with semantical properties intended:
- to compose text horizontally and vertically,
- to insert (optional) cutpoints,
- to print characters.

The output of this virtual machine is directed to an output device, which can be a terminal, a file, a screen window, ... VPRINT is used extensively in BIGMACS, our universal editor, to maintain interactively textual representations of CEYX objects in windows.

5.3. Cx_Yacc: a Parser Generator

We have seen how to produce textual representations of LISP or CEYX structures in using VPRINT. Conversely, we use Cx_Yacc [Berry83] to generate parsers producing internal CEYX structures from a textual representation. The well-known Yacc [Johnson75] universal-parsers meta-parser has been modified so as to generate CEYX parsers instead of C-parsers.

Cx_Yacc has been used since 1982 to generate parsers for new programming languages under development. Moreover it is used in BIGMACS to allow mixed edition: at any time the user can choose to edit programs either on their textual representation using the BIGMACS textual mode, or on their internal CEYX representation using the BIGMACS hierarchical mode. The communication between the two modes is then insured by VPRINT and the parser generated by Cx_Yacc.

5.4. PEPE: a Full Screen Text Editor

PEPE [Chailloux83a] is a reduced version of the well-known Maclisp Emacs Text Editor [Greenberg80]. It includes a LISP mode used to edit LISP functions or LISP files. For the purpose of running on a Z80 machine this editor is very small (2000 cons-cells) and allows only mono-buffer, mono-window edition. It is nevertheless easily extensible at LISP level. It has been designed for educational purpose and is now used in french high-schools under LE_LISP80.

5.5. BIGMACS: a Universal Editor

BIGMACS [Hullot84] gives promise of editing any kind of object defined under CEYX. The kernel of the system consists in a virtual machine called SystemManager: it keeps pointers on a tree of processes, a current process, a list of screens and a current screen. Screens can be standard alphanumerical screens, bitmaps or color bitmaps. Each process uses an InputStream and an OutputStream to communicate with the external world and points to a processor. InputStreams are linked to input devices (keyboard, mouse, file) and OutputStreams to output devices (terminal, screen window, file). Basic families of objects have conducted to the design of specialized processors:

- Text Editor, which works on the structures: characters, lines and list of lines. This processor is a kin to Multics Emacs [Greenberg80];

- LISP Editor for editing LISP programs. It closely resembles the Interlisp editor [Teitelman76] with a video interface;

- Hierarchy Editor, which works on any kind of CEYX structures. It consists in a basic management system for keeping pointers onto a structure and some selected substructures. VPRINT is used to maintain interactively on windows a textual representation of the structures being edited. To each kind of structure is associated a set of actions (its semantical properties and the ones it inherits) which can be applied to instances of this structure. These actions are the basic menu of the pointed structure and can be activated either by keyboard keys or by pointing on displayed menus.

- Tree Editor. It is a refinement of the Hierarchy Editor focusing on the tree structure. It allows edition of programs on their abstract syntax trees. It can be compared to programming language oriented editors like Mentor [Donzeau&al75] or The Cornell Program Synthetizer [Teitelbaum&al81].

- Windows Editor, Library Editor, ... are examples of other refinements of the Hierarchy Editor.

The user has the full power of LISP and CEYX to define new kind of processors. It has been extensively used to tailor BIGMACS to various special purpose applications, including VLSI design aids [Chailloux&al83].

6. Applications

LE_LISP is now widely used in European Research Centers, Universities and Industries. We present in this section some interesting applications.

6.1. VLSI Design

LE_LISP, CEYX and BIGMACS are used as the basic management system for the INRIA VLSI workstation [Chailloux&al83, Levy82]. Many specialized VLSI processors have been built, including:

- the LUCIFER language (VLSI mask specification),
- the HELL language (Hierarchical Electrical Language) [Serlet84], including multi-mode simulators,
- the node extractor [Heintz82] and the design rule verifier [Gallot83],
- the VLSI mask editor called LUCIOLE [Levy82, Chailloux&al83] running on COLORIX, a home made color bitmap [Audoire83].

6.2. Symbolic computation

The Formel system developed by G. Huet and his co-workers at INRIA is running on LE_LISP. This system, based on primitives for manipulating first order terms (matching, unification, associative and commutative unification), as described in [Hullot80], is used for various computations on algebraic specifications (synthesis of canonical rewrite systems from equational theory presentations [Knuth&Bendix70], proofs by induction in standard models [Huet&Hullot81], solving word problems and isomorphisms problems in finitely presented algebra [LeChenadec83], first-order theorem-proving [Fages83]). Furthermore, a new system for manipulating mathematical theories is under implementation in LE_LISP. Its core is an ML environment adapted from Edimburgh LCF [Gordon,Milner,Wadsworth79]. An ML compiler has been developed in collaboration with the Cambridge LCF research group [Paulson83]. Experiments with denotational definitions of programming languages written in ML, and run as abstract interpreters, show great promises for quick prototyping of programming languages design [Cousineau&al84].

6.3. Design of Programming Languages

The LE_LISP programming environment is especially well suited to conduct research on new programming languages: the abstract syntax of the language can be specified using the CEYX tree structure, its concrete syntax can be produced by VPRINT and a parser can be generated by Cx_Yacc. Moreover, as soon as the abstract syntax is being defined, BIGMACS gives for free a generic edition scheme which can be refined incrementally. It has been extensively used at Ecole des Mines for the design of CDS [Berry&al81], at INRIA by the Parallelism Project (Meije, Ecrin) and at IRCAM for the Forme Language [Cointe&Rodet84].

7. Future developments

Future work on the system will be:

- *Micro coding* of vital parts of the kernel. This operation will be realized on VAX/780, and Perkin Elmer 32/50. On SM90 we will micro code a separate processor to test various architectures for the future LISP chip.

- *Multi-processors architecture*, around the MC68000 based SM90. We will use three MC68000 processors, the first one devoted LE_LISP, with 2 mega bytes of private memory, the second one to the UNIX system, and the last one dealing with a colored bitmap.

- *LLM3 chip*: we will build, with the Sycomore project, a complete processor running LLM3.

References

[Audoire 83] Audoire L., *COLORIX 90, notice de présentation*, rapport interne projet VLSI, INRIA, 1983.

[Berry&al81] Berry G., Curien P.-L., *Sequential algorithms on concrete data structures: the kernel of the applicative language CDS*, Acts of the French-American symposium on semantics, Fontainebleau 1981.

[Berry83] Berry G., *Cx_Yacc, a parser generator*, (à paraître), Ecole des Mines, Sophia Antipolis, 1983.

[Bobrow83] Bobrow D.G., Stefik M.J., *The Loops Manual*, Memo KB-VLSI-81-13, Xerox PARC.

[Burke&al82] Burke, Carette *NIL Notes for Release 0*, Massachusetts Institute of Technology, December 1982.

[Chailloux78] Chailloux J, *A Vlisp interpreter on the virtual VCMC1 machine*, LISP Bulletin #2, July 1978.

[Chailloux80] Chailloux J., *Le modèle Vlisp: description, évaluation et interprétation*, Thèse de 3ème cycle, Université de Paris VI, Avril 1980.

[Chailloux83a] Chailloux J., *LE_LISP 80 version 12, manuel de référence*, Rapport Technique no 27, INRIA, Juillet 83.

[Chailloux83b] Chailloux J., *La machine virtuelle LLM3*, internal report of the VLSI project, November 1983.

[Chailloux&al83] Chailloux J., Hullot J.-M., Levy J.-J., Vuillemin J., *Le Système LUCIFER d'aide à la conception de circuits intégrés*, Rapport INRIA 196, Mars 1983.

[Chailloux84] Chailloux J., *LE_LISP V15, Manuel de référence*, Rapport INRIA, June 1984.

[Cointe&Rodet84] Cointe B., Rodet J., *Formes: an Object and Time Oriented System for Music Composition and Synthesis*, 5th *ACM Conference on LISP and Functional Programming, August 1984.*

[Cousineau&al84] Cousineau G., Huet G., *Compilation de ML en LISP*, in preparation.

[Devin84] Devin M., *Le portage du système LE_LISP: mode d'emploi*, (to appear), Ecole des Mines, Sophia Antipolis, 1983.

[Donzeau&al75] Donzeau-Gouge V., Huet G., Kahn G., Lang B., Levy J.-J., *A Structure Oriented Program Editor: a first step toward computer assisted programming*. International Computer Symposium, North Holland Publishing Co, (1975).

[Fages83] Fages F., *Formes canoniques dans les algèbres booléennes, et application à la démonstration automatique en logique du premier ordre*, thèse de 3ème cycle, Université Paris Sud, Octobre 1980.

[Foderaro&al81] Foderaro, Sklower, *Franz LISP Manual*, Univ. of California, Berkeley, Ca., September 1981.

[Gabriel82] Gabriel R., Masinter L., *Performance of LISP systems*, 1982 ACM Symposium on LISP and Functional programming.

[Gallot83] Gallot L., *Techniques informatiques dans la vérification des*

gardes technologiques des circuits intégrés, Thèse de troisième cycle, Octobre 1983, Université de Paris Sud, Orsay.

[Goldberg82] Golberg A., Robson D., Ingalls D., *Smalltalk-80: The Language and its Implementation*. Reading, Massachussets: Addison-Wesley.

[Gordon,Milner,Wadsworth79] Gordon, Milner, Newey, Morris, Wadsworth, *A Metalanguage for Interactive Proof in LCF*, Internal report CSR-16-77, department of Computer Science, University of Edimburgh, Sept. 1977.

[Greenberg80] Greenberg B., *Prose and Cons - Multics Emacs: a Commercial Text-Processing System in LISP*, LISP Conference 1980.

[Greussay77] Greussay P.,
Contribution à la définition interprétative et à l'implémentation des lambda-langages Thèse d'Etat, Université de Paris VI, Novembre 1977.

[Griss82] Griss, *PSL: a portable LISP system*, ACM, 4th symposium on LISP and functional programming, 1982.

[Heintz 82] Heintz, *Un extracteur de circuits intégrés*, thèse de 3ème cycle, Paris Sud, Octobre 1982.

[Huet&Hullot81] Huet G., Hullot J., *Proofs by induction in Equational theories with constructors*, JCSS, Vol 25, No 2, October 1982.

[Hullot80] Hullot J.-M., *Compilation de formes canoniques dans des théories équationnelles*, thèse de 3ème cycle, Université Paris Sud, Octobre 1980.

[Hullot83] Hullot J.-M., *CEYX, a Multiformalism Programming Environment*, IFIP83, R.E.A. Mason (ed), North Holland, Paris 1983.

[Hullot84] Hullot J.-M., *CEYX, BIGMACS & Co, The Manual*, INRIA report to appear (1984).

[Knuth & Bendix 70] Knuth D., Bendix P., *Simple word problems in Universal Algebra*, in Computational probelms in Abstract Algebra, Ed. Leech J., Pergamon Press, pp. 263-297 1970.

[Johnson75] Johnson S. C., *Yacc: Yet another compiler compiler*, Comp. Sci. Tech. Rep. No. 32, Bell laboratories, Murray Hill, New Jersey.

[LeChenadec83] LeChenadec Ph., *Formes canoniques dans les algèbres*

finiment présentées, thèse de 3ème cycle, Université Paris Sud, 1983.

[Levy82] Levy J.-J., *On the lucifer system*, Prentice Hall, University of Bristol, B. Rondel, Ph. Treleaven Ed.

[McCarthy62] McCarthy J, *LISP 1.5 Programmer's manual*, The M.I.T. Press, Cambridge, Mass., 1962.

[Moon74] Moon D., *Maclisp Reference Manual*, MIT Cambridge (1974).

[Paulson83] Paulson L., *Recent Developments in LCF: Examples of structural induction*, Technical report No. 34, University of Cambridge, England.

[Saint-James84] Saint-James E., *Recursion is more efficient than iteration*, 5th ACM Conference on LISP and Functional Programming, August 1984.

[Serlet84] Serlet B., *Description structurelle et simulation de circuits intégrés*, thèse de 3ème cycle, Université Paris Sud, Janvier 1984.

[Steele&al82] Steel G., *Common LISP Reference Manual*, Spice Project, Carnegie Mellon University, November 1982.

[Teitelbaum&al81] Teitelbaum T., Reps T., *The Cornell Program Synthesizer: a syntax directed programming environment*, CACM 24,9 (September 81).

[Teitelman76] Teitelman W, *Interlisp Reference Manual*, Xerox PARC, (1976).

[Vuillemin&al84] Vuillemin J., Chailloux J., *Aritrary precision rational arithmetics in LE_LISP*, raport INRIA, to appear, 1984.

[Waters81] Waters, R.C., *Gprint: a LISP Pretty Printer Providing Extensive User Format-Control Mechanisms*. MIT/AIM-611, October 81.

[Weinreb&al81], Weinreb D., Moon D., *LISP Machine Manual*, Fourth Edition, Artificial Intelligence Laboratory, M.I.T., Cambridge, Mass., July 1981.

[White79] White J, *NIL - a perspective*, Proc. of the Macsyma User's conf., Washington D.C., June 1979.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique