DIRECT EXECUTION OF LISP ON A LIST_DIRECTED ARCHITECTURE

J.P. SANSONNET, M. CASTAN, C. PERCEBOIS, D. BOTELLA, J. PEREZ

Laboratoire Langages et Systèmes Informatiques
Université Paul Sabatier
118 route de Narbonne 31062 TOULOUSE CEDEX
FRANCE

We have defined a direct-execution model dedicated to non-numerical processing which is based upon an internal representation of source programs derived from LISP. This model provides good support for both sophisticated editing (syntactical parsing, tree manipulation, pretty-printing, ...) of conventional languages and artificial intelligence languages. A high level microprogramming language (LEM) was designed to write the interpreters and the editors. A hardware processor was built and a LISP interpreter, microprogrammed in LEM, has been operational since September 1980.

First, the influence of LISP on the LEM language and the architecture is discussed. At the LEM level, we will see that LISP has prompted the control constructs and the access functions to the tree-structured internal form. As for the architecture, we present the hardware implementation of a special garbage collector based upon reference counters.

In turn, the machine has influenced the implementation of LISP. We present here the structure of our LISP interpreter and we give evaluation measures dealing with size, development effort, speed ; they prove that programming in LEM is easy, short to debug and very concise. Moreover, the speed of our LISP interpreter confirms that the architecture is very efficient for symbolic processing.

1. INTRODUCTION

In conventional Direct-Execution schemes [1,2], internal forms are linear Directly Executable Languages (DELs) ; i.e. they are composed of contiguous fields with a variable length. Generally field lengths are defined by frequency-based encoding [3]. Such forms have proved to be bad environments for both translation and interpreting steps : the different processors must achieve complex work to access the internal form both in the encoding and decoding mode. Moreover, linear DELs do not satisfy the constraints dictated by interactive functions. For example, from the DEL it is not possible to come back to the source-text. In fact, even if DELs are more sophisticated than order-codes, they present the same drawbacks in the perspective of emulation.

Therefore we have retained a non-linear form where each entity will be built from a unique object, the cell :

(LPT,RPT,DES)

Roughly, LPT (Left PoinTer) and RPT (Right PoinTer) are references to other cells and DES is a tag used to describe the entity supported by the cell. The addition of a tag to the binary cell allows the DEL designer to define various environments : binary trees, triangular trees as in MINERVE [4], lists as in LISP [5]. These environments have proven to be good for editing and interpreting and they have resulted in eminent interactive programming systems with LISP : INTERLISP [6] or PASCAL : MENTOR [7]. As the internal form was derived from LISP it was called the 3L form (for LISP-Like-Language) and the associated Direct-Execution scheme was called the 3L-model [8].

In the 3L-model, a first processor, the editor, achieves a bidirectional translation between the source-text and the associated 3L form. High level editing functions can be attached to the editing process : syntactical parsing, pretty-printing, meta-evaluation. A second processor, the interpreter, directly evaluates the 3L form through hardware operators. Editing and interpreting do not dictate the same speed constraints but, as they manipulate the same form, they will be expressed in the same language : the Language for EMulation (LEM). LEM is the high level microprogramming language of a hardware processor that is specially designed to process 3L forms very efficiently : the 3L-machine (M3L) [10].

Rather than a LISP-machine, M3L must be viewed as a CELL-machine ; i.e., it can emulate every high level application provided that it is expressed in terms of (LPT,RPT,DES) cells, but obviously LISP remains a model for such a scheme and many of LISP's features are embedded in M3L. As the syntax of LISP is trivial, editing and interpreting functions are

considerably reduced. Therefore, our first software application on M3L was the microprogramming of a LISP interpreter, called GLISP. In the following sections, we shall discuss the supports of list processing and we shall give some evaluating results upon the implementation of GLISP.

2. LEM

2.1. An abstract machine

Several abstract machines have been proposed to implement LISP on conventional computers. For example, the VLISP systems are written with VCMC2 [11], and A. LUX has defined a two-leveled abstract machine [12] so as to support sophisticated memory management. They are prompted by the large semantic gap lying between LISP and its host systems. As a result, a statement in the abstract machine produces a lot of instructions when it is compiled or macro-generated.

The Language for EMulation is an abstract machine specialized in the manipulation of (LPT,RPT,DES) objects. From that, it would seem to be easy to define the level of LEM functions between LISP itself and the machine-language. However, LEM is also the microprogramming language of M3L. This dictates a one-to-one correspondence between LEM statements and M3L fixed-size microinstructions, so that, LEM concepts will be the LISP functions that we will be able to implement directly in hardware.

Beyond functional constraints, emulation processing prompts a special shape for the language. LEM has to express cell-manipulations in a clear and structured way, thus, typical objects and functions will be embodied in a high level ALGOL-like syntax that must be compiled to produce the fixed-size microinstructions of M3L. On the other hand, it may happen that the microprogrammer needs to directly control intimate resources within the machine.This work, referred to as exotic functions, is expressed in LEM through the following :

$$f(p_1,p_2 \cdots p_n)$$

f corresponds to the opcode of a M3L microinstruction and p_i's are its parameters. When such a statement is encountered in a LEM text it is just assembled, under the complete responsibility of the microprogrammer. The high level part of LEM provides a good programming practice and the exotic part of LEM allows the efficiency to be maintained.

2.2. Resources and functions

Basic objects of LEM are decimal or hexadecimal constants and predefined registers. The allowed operations on registers and constants are :

- MOVE R0 ← R1
- COMPARISON IF R1 < 0 THEN R2 ← # 00FF
- ADD and SUB R1 ← R1 + 1

The main resource of LEM is the pair-cells memory which can be easily defined by :

```
CONST N=16  % value on the M3L prototype %
TYPE  cell = STRUCTURE
             CAR : ARRAY [1..N] OF bit ;
             CDR : ARRAY [1..N] OF bit ;
             DES : ARRAY [1..8] OF bit
             END ;
VAR pair-cells-memory : ARRAY [1..2^N] OF cell ;
```

Each cell can partially be accessed to one of its three fields owing to the CAR , CDR and DES functions in the read or write mode.

Examples :

CAR(R1) ← R2 : The cell whose address is into R1 receives the content of R2 in its CAR field

R3 ← DES(R0) : R3 receives the descriptor part of the cell addressed by R0

To determine accurate LEM functions, we have made static and dynamic measures upon a first LISP interpreter [13]. It appeared that accessing to the whole cell in the read mode (UNCONS) was critical whereas the symmetrical operation (CONS) was obviously less used. Therefore, in LEM, we can "decode" a cell in a single microinstruction cycle with :

```
FETCH R0 INTO R1 AND R2
                 equivalent to   R1 ← CAR(R0)
                                 R2 ← CDR(R0)
```

2.3. Control structures

A LEM module (interpreter,scanner,parser...) is a non-ordered set of little microprocedures. In a module, the control appears at two different levels:

. Internal control

The form of LEM dictates the use of structured control constructs. We retained :

- The sequence : BEGIN-END
- The conditional : IF-THEN-ELSE
- The CASE-OF enables a multiple branch to be performed according to the DES field. It is then possible to decode the cells efficiently.
- The LOOP with EXIT, because it is more general and more convenient than the DO-WHILE construct.

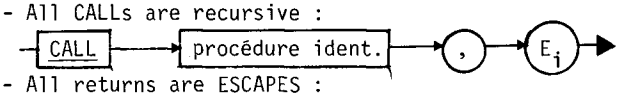
. External control

With respect to the external control to procedures, the recursive CALL is the main control structure. To this purpose, the LEM registers are divided into two groups :

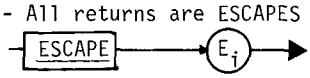
- A_i registers serve for current work and for parameter passing between LEM microprocedures
- R_i registers are local to microprocedures. When a CALL occurs they are saved and restored when returning.

To the recursivity, a general escape mechanism has been added. The principle of escapes was given by REYNOLDS [14] and was implemented in LISP by GREUSSAY [15]. This control construct is hardwired on M3L and appears in LEM as the complementary concept to recursive CALL. In LEM :

- All CALLs are recursive :



- All returns are ESCAPES :



when an ESCAPE statement is performed the recursivity stack is popped until encountering the same (Ei) tag. If omitted, the escape tag is implicitly E_0.

. Tail-recursions are eliminated when using the CONTINUATION microinstruction which passes the control from a microprocedure to another without context switching, i.e. without push operation.

2.4. Examples of microprogramming in LEM

LEM is directly derived from LISP thus the text is very concise and as one fixed-size microinstruction corresponds to one LEM statement the microcode is very compact.

It is difficult here to give a meaningful example involving escapes because it needs the description of a full emulation module. The first example shows how passing from LISP expression to LEM is easy. It is the ACKERMANN function :

```
PROCEDURE ACK ;
% INPUT A0=n ; A1=m %
% OUTPUT A0=ACK(n,m) %
BEGIN
IF A0=0 THEN BEGIN
             A0 ← A1+1 ;
             ESCAPE
             END ;
IF A1=0 THEN BEGIN
             A0 ← A0-1 ;
             A1 ← 1 ;
             CONTINUATION ACK
             END ;
R0 ← A0 ;  % save N %
A1 ← A1-1 ;
CALL ACK ;  % A0 ← ACK(N,M-1) %
A1 ← A0 ;
A0 ← R0-1 ;
CONTINUATION ACK
END ;  % 13 µI %
```

```
(DE ACK(N M)

  (COND
        ((ZEROP N)(ADD1 M))

        ((ZEROP M)(ACK (SUB1 N) 1))

        (T (ACK (SUB1 N)(ACK N (SUB1 M))))))
```

As a second example, here is the interpretation of LISP functions CAR and NTH when their arguments have been evaluated by EVAL and when invoked by APPLY :

```
PROCEDURE NTH ;
% INPUT : A0=n, A1 points to the list head %
% OUTPUT : A0 points to the nth cell       %
BEGIN
IF  A0 > 1  THEN  BEGIN
            A0 ← A0-1 ;
            A1 ← CDR(A1) ;
            CONTINUATION NTH
            END ;
A0 ← A1 ;
ESCAPE
END  % 6 µI %

PROCEDURE CAR ;
% INPUT : A0 → List      %
% OUTPUT : A0 ← CAR(list) %
BEGIN
   A0 ←  CAR(A0) ;
   ESCAPE
END  % 2 µI %
```

When it is compiled, the procedure ACK produces 13 fixed-size 32-bit microinstructions only (NTH produces 6 µI and CAR 2 µI). These three examples illustrate the great conciseness of LEM texts due to its LISP-like semantic.

3. M3L

The M3L project started in september 1977 with the definition of the 3L-model. The LEM language was then designed and an architecture was studied in 1978. The M3L prototype was put on the drawing board in 1979 and became operational in June 1980 [16]. Now it supports a complete microsystem and the GLISP interpreter. The prototype is composed of 700 chips, most of them being related to memory resources.They are wrapped on five boards supporting up to 160 16-pin standard chips. The boards are placed into a 40 x 50 x 50 cm box which is connected to a display screen and to a microsystem (H11) responsible for file management.

Because the M3L prototype is an implementation of the LEM abstract machine, we wanted to make it as simple as possible. For this reason,the resources of M3L are connected through a single 16-bit general bus. Beyond input/output and interrupts management, the main resources are : the pair-cells memory, the micro-context stack and a fast arithmetical operator (AMD 2903) which contains the A_i registers. The M3L machine was developed at the university at the cost of $ 14000.

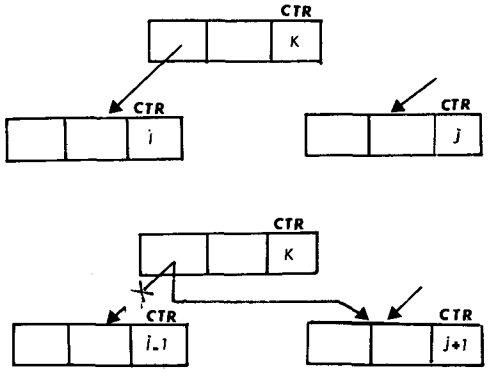
3.1. The pair-cells memory

The implementation of the pair-cells memory corresponds exactly to the LEM description. 64 K (CAR, CDR,DES) cells are available with a 150 ns access time. They are made of 16 K-bit dynamic MOS memory chips. There is no cdr-coding as in the GREENBLATT LISP-machine [28] because we thought that the cost of saved memory (lower than 8 percent of the whole prototype cost) did not justify the complexity thus introduced in memory management.

. A garbage collector with reference-counters

To achieve the recycling of discarded cells, several standards have been proposed. The method of NEWELL for IPL-V [17] gives the responsibility of memory management to the programmer. The method of Mc CARTHY is the most popular [5], a "garbage collecting" procedure visits the lists in two steps : during the first one, used cells are marked and during the second one, non-marked cells are bound to the free-list. The principal disadvantage is the important suspension time in case of a great deal of cells. To overcome this drawback , some authors have proposed real-time algorithms [29]but they need complex multiprocessing environments or a two-processor architecture. Another method [18,19] associates

a reference counter with each cell which becomes free when its counter is equal to zero. This method was studied by Mc CARTHY [20] but is was not implemented because of hardware obstacles, however it is used by WEIZENBAUM for SLISP [21]. The principle of the garbage-counter is illustrated below :



Its main advantage is that cells can immediately be reclaimed when they become free. Unfortunately,there are some drawbacks : counters are memory consuming, ±1 operations are time consuming and circular lists cannot be reclaimed. DEUTSCH and BOBROW give a solution to the two first points [22] by using a file where the operations are stored. Thus, it is possible to delay the work, and for example, to do it during input/output.

. Implementation on M3L

Since M3L is a list-directed architecture, it seemed interesting to integrate the elements of the garbage-counter at the hardware level. The first question is the determination of the counter size. Our measures [13] on LISP have shown that a 3-bit counter enables more than 98 percent of the cells to be reclaimed. These results are confirmed by CLARK and GREEN measures [23] on large LISP programs.

Unfortunately, cells are also referenced by registers and, when a recursive call occurs, they are pushed onto the stack memory. The reference number grows in proportion to CALLs, so it can overflow rapidly. Like DEUTSCH and BOBROW, we separate the cell-reference counter from the stack-reference counter, the later being reduced to a one-bit tag which indicates if the cell is referenced by the stack. Likewise, when a pop operation is performed, updating the cell tag requires a complete checking operation upon the stack. Therefore, the first occurrence of a reference into the stack will be marked as well. Then, the algorithms are :

PUSH(α) : If the cell α is not marked then mark the cell & mark the reference into the stack else nothing to be done

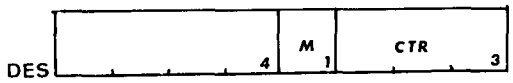
POP(α) : If the popped α is marked then delete the mark of the cell α else nothing to be done

In LEM, only two exotic microinstructions are needed to implement the garbage-counter :

- The counting one achieves ±1 operations
- The marking one achieves push/pop operations.

Marking and counting are performed in a single microinstruction cycle, thus providing a very efficient garbage-collector.

To implement counters (CTR) and cell marks (M) a part of the DES field is used :



To implement stack marks, a one-bit tag is added to each local R_i register, and stored in the MARK-stack.

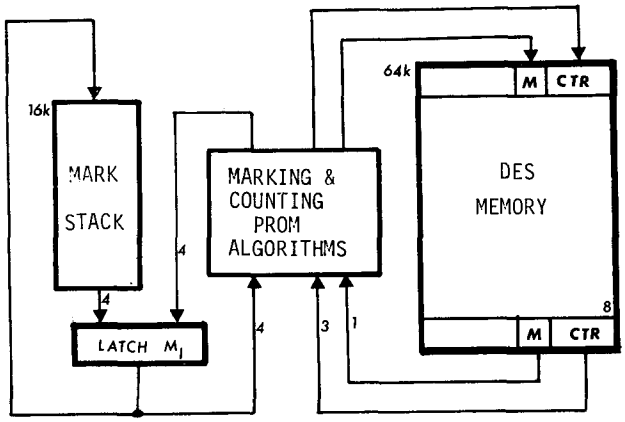


Fig.1. Garbage-counter hardware support

This implementation needed only four 4116 chips to support the MARK-stack and five 32 x 8 PROMs to handle the counting & marking algorithms.

3.2. The stack memory

If a CALL microinstruction is performed, the current microcontext is pushed onto the stack memory. If it is an ESCAPE the corresponding microcontext is popped. To ensure an efficient support to the recursivity, a whole microcontext is puhsed (or popped), in parallel, in a single microinstruction cycle.

Typically a microcontext contains the values of the R_i registers with their garbage marks, the microprogram counter and the escape tag. An important point is knowing how many R_i registers are required in emulation processing. Our measures on a first LISP interpreter [13] led to the following diagram :

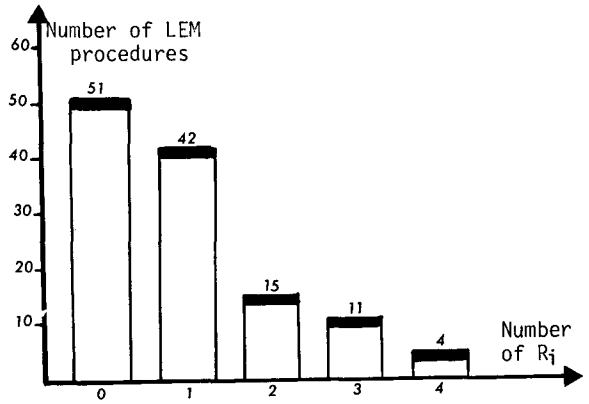


Fig.2. Static occurrences of R_i registers in LEM microprocedures

Four R_i registers have proven to be enough to support a LISP interpreter. This number was retained for the M3L prototype, giving a 90-bit microcontext word. The stack memory and the pair-cells memory are made with the same technology. Therefore, 16-K microcontexts are available and they can be extended on a second storage, up to 64 K, thanks to a hardwired swapping mechanism.

The implementation of escapes is very simple.The ESC microinstruction is performed, as many times as necessary, until finding an escape tag into the stack that is equal to the microinstruction escape tag (fig.3).

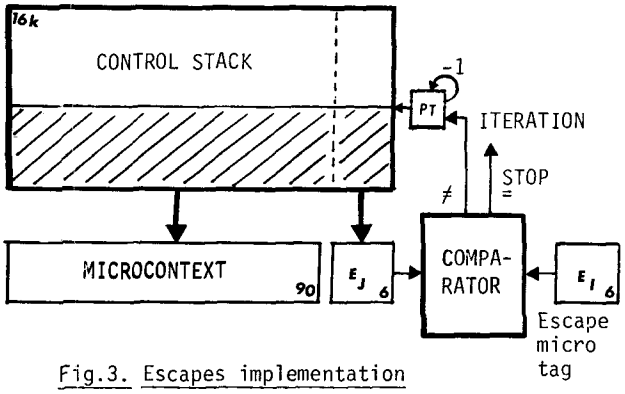


Fig.3. Escapes implementation

An important result of the stack implementation is that a CALL instruction is done in one microinstruction cycle just as a simple register-to-register transfer. This has proved to involve considerable changes in programming practice, in particular for the GLISP interpreter.

4. GLISP

4.1. Structure of the interpreter

Presently, GLISP features about 50 standard functions. It is constantly augmented, but the main concepts of LISP are already implemented : DEFUNs, MACROs, LAMBDAs, ESCAPEs, CLOSUREs ... In addition, a MATCH function was implemented and included by the artificial intelligence group so as to evaluate the performance of M3L.

For historical reasons, GLISP is not build from the APPLY/EVAL model. It inherited its structure from TLISP [24] where each standard function is responsible for the evaluation of its arguments. This model is interesting because it avoids EVLIS calls and needs no typed representation (n-SUBR, FSUBR ...). Thus, it is faster than the APPLY/EVAL model, but it involves more complexity in SUBRs microprogramming.

In parallel with GLISP a microsystem was written to manage hardware resources. It handles interrupts, Input/Outputs, bootstraping, 32-bit floating arithmetic (log, sin, powers ...) and a debugging module which enables the user to display and to change all the registers and the memories within M3L. Not only does GLISP call the microsystem for performing Input/Outputs, but also it interacts with the debugger, owing to escapes. Hence, a LISP evaluation can be suspended, the user can access to the heart of the machine, he can display and change values, and then the current LISP evaluation can be resumed.

4.2. Frequency measures

. The table I gives the static occurrences of registers in the GLISP module:

I	0	1	2	3	4	5	6	7	8	9	11	12	13	14	15
A_i registers	426	145	20	5	8	9						1*	5*		
R_i registers	208	135	55	22											

(*) A12 and A13 are used to communicate with the microsystem

TABLE I - Static occurrences of registers in GLISP

These results show that 8 A_i registers would be enough and that 4 is a good number for R_i's.

Table II gives the occurrences of LEM microinstructions in a GLISP module.

- Statically, the dramatic percentage of external control (45 percent of the total amount) shows how important CALL, ESCAPE and CONTINUATION microinstructions are. The elimination of tail-recursions proves to be essential. With respect to memory access, writing and reading are balanced, further, note the relative importance of the FETCH microinstruction.

- The dynamic measures confirm the importance of external control and emphase the role of the FETCH microinstruction. Obviously, reading becomes much more important than writing. Generally, they show that formal processing, dealing with memories and register resources is prevalent, whereas numerical processing and exotic functions are quite rare.

MODE / MICROINSTRUCTIONS	STATIC		DYNAMIC	
	PERCENTAGE	PARTIAL AMOUNTS	PERCENTAGE	PARTIAL AMOUNTS
1. MEMORY ACCESS				
1.1. READING				
- Separate reading				
. LEFT	4.19		5.35	
. RIGHT	2.34		2.29	
. DES	0.01		2.64	
- FETCH microinstruction	3.35	9.89	7.53	17.81
1.2. WRITING				
. LEFT	3.10		2.88	
. RIGHT	2.93		2.63	
. DES	2.93	8.96	2.36	7.87
2. REGISTER MOVES	22.61	22.61	23.60	23.60
3. EXTERNAL CONTROL				
- Recursive CALL	18.01		11.50	
- CONTINUATION	14.15		7.32	
- ESCAPE	12.81	44.97	11.50	30.32
4. COMPARISONS				
- for conditional branch	11.81		14.50	
- for decoding (CASE-OF)	0.17	11.98	5.56	20.06
5. EXOTIC	1.34	1.34	0.05	0.05

TABLE II - Occurrences of LEM microinstructions in GLISP

4.3. Performances

. Size

The table III gives a size evaluation for the different modules that are now operational on M3L. Other size evaluations have been made by handcoding these lines into the assembly language of a very typical 16-bit machine : the ratio of conventional instructions to LEM microcode is 7. This exhibits the semantic gap which is lying between list-directed applications and this computer.

LEM MODULE	Number of LEM lines	Number of fixed-size microinstructions
GLISP interpreter	2047	1295
MATCH module	319	217
Arithmetical module	393	155
Microsystem	2083	1250
TOTAL	4842 lines	2917 µI

TABLE III - Size evaluation of LEM

. Effort

When developing a processor in LEM, effort is considerably reduced by the conciseness of LEM texts and the accuracy of the LEM abstract machine. For example, the complete system that is in the table III has been set up by two people in six months. On a conventional system, this will require more than 20000 instructions to write and debug.

. Speed

In [11] J. CHAILLOUX gives some measures about eminent LISP implementations on DEC systems. They concern the familiar Fibonacci function :

```
(DEFINE FIB (N)
   (COND ((ZEROP N) 1)
         ((EQ  N 1) 1)
         (T (ADD (FIB (SUB1 N))(FIB (DIFFER N 2))))))
```

The table IV gives execution times in seconds when FIB is applied to the number 20 : (FIB 20)→ 10946.

execution times in seconds	INTERLISP	MACLISP	VLISP-10	VLISP-11	GLISP
COMPUTER	PDP-10 KA-10	PDP-10 KA-10	PDP-10 KI-10	PDP-11 LSI 11/02	M3L
CLOCK PERIOD	1000 ns	1800 ns	1000 ns	-	500 ns
EXECUTION TIME	105 s	48 s	13.5 s	35 s	2.6 s

TABLE IV - FIB (20)

The Table IV proves that symbolic processing requires a specific hardware implementation to get best results. The following measures are dealing with the Chip-LISP built at the MIT [27] and called SCHEME 79. It is very difficult to compare the architecture of M3L with that of SCHEME 79 because M3L is a complete computer whereas SCHEME 79 features neither memory resources nor input/output device ; thus, it must be connected with a host machine. Moreover, it is supplied with a very poor arithmetical unit (± 1) so it is no longer possible to operate with above definition of FIB. In order to achieve a more accurate comparison, the addition

operator has been redefined in a recursive way. The measures of the Table V were made with the following definitions (see [30]) :

```
(DE ADD (X Y)
   (COND ((ZEROP X) 1)
         (T (ADD (SUB1 X)(ADD1 Y)))))

(DE FIB (N)
   (COND ((ZEROP N) 0)
         ((ZEROP (SUB1 N))1)
         (T (ADD (FIB (SUB1 N)) (FIB (SUB1 (SUB1 N)))))))

(FIB 20) → 6765
```

execution times in seconds	SCHEME CHIP-LISP	GLISP M3L
clock period	1595 ns	500 ns
32 k-cell free-memory	∿ 60 s	13 s
32 k-cell half free memory	∿ 180 s	16 s

TABLE V - M3L VERSUS SCHEME 79

. LISP/LEM translation

We have seen that the LEM semantic was very near from LISP. The result is that user's functions, written in LISP, are easy to translate into LEM. Then, best performances are obtained. In the table VI VLISP-11, GLISP and LEM-coded procedures are compared. FIB is the Fibonacci function given above, with the first definition, ACK is the Ackermann function (see § 2.4) and MATCH is a classical pattern maching function applied to X and Y with : (SETQ X (LIST 1 2 ... 2000) Y (LIST 1 2 ... 2000)).

execution times in seconds	VLISP11	GLISP	LEM
FIB (20)	35 s	2.6 s	<0.1 s
ACK (3,4)	20 s	1.8 s	<0.1 s
(MATCH X Y)	30 s	2.8 s	0.1 s

TABLE VI - LEM-CODING IN FAST

5. DISCUSSION

5.1. M3L is a high level concept architecture

M3L has a hardware knowledge of high level objects, like binary cells, and of their management, e.g. the hardwired garbage counter. It is also aware of high level control concepts as recursivity, escape and tail-recursion eliminating. One could think that these new things would lead to a complex machine ; on the contrary, they have yielded a more uniform architecture. Indeed, there is no need to embed in hardware high level concepts if they are never put to use. As for the concepts mentioned above, they correspond to 65 percent of the static occurrences in GLISP (50 % dynamic) !

Puting in hardware high level concepts has a real impact on programming practice. For example, performing the LEM microinstructions :

CALL proc , E1

needs no more execution time than performing a familiar register move :

A0 ← (A1)

This point explains that microprocedures in the GLISP interpreter are very short : 8 microinstructions on the average, and that they call each other intensively : every other time, GLISP statements are CALLs, ESCAPEs or CONTINUATIONs. Thus, size is reduced, programming is more structured and LEM texts are modular, readeable, versatile and maintainable.

5.2. M3L is a distributed architecture

1. Two kinds of processing

The architecture of M3L is based upon the division of functions into two classes [25] :

- Formal processing : It is related to user understanding. It achieves environment transfers between the external world and the computer. It is essentially non-numerical.
- Effective processing : It is related to hardware resources. It achieves the work really demanded by the user. It concerns arithmetic, Input/Output and special purpose operations .

Conventional computer systems are built from effective processing only. On the other hand, M3L is built from formal processing but it also includes effective processing capabilities. At the LEM level, formal processing is expressed through the high level syntax and effective processing through exotic functions. In the same way, at the hardware level, formal processing is supported by memory resources whereas effective processing is supported by arithmetical, logical and I/O functions.

2. An efficient support of effective processing

One could think that a formal processing directed architecture could not be efficient in traditional work, but the separation of functions has enabled us to supply our prototype with advanced technology. The family AMD 2900 was preferred because it is complete, versatile and it provides fast functions : ALU functions are performed by an AM 2903, interrupts are managed by an AM 2914 and sophisticated arithmetic is done by an AM 9511 monolithic microprocessor. Therefore, M3L can compare with todays conventional computers.

3. Distribution of effective processing

At the hardware level, effective processing is held by the Emitter/Receptor concept : An effective processor is viewed as a black-box connected to the general bus as an Emitter (from the box to the bus) or as a Receptor (from the bus to the box) and it is interfaced by two exotic microinstructions. Presently, the 9511 chip, the I/O system and a debug board are connected in such a way, and 12 other E/R ports are available to plug extra hardware tools. In particular, it is possible to add and to control either fast hardware operators (fast multiplication/division, array processors ...) or very specialized operators (graphic processors ...). Then, formal processing can be regarded as the head and effective processors as the arms.

5.3. M3L is an application-directed architecture

Generally, applications are achieved in three steps : first, specific functions and objects are exhibited, second, an Application-Directed-Language (ADL) is defined, and finally, one tries to implement the ADL on a host machine. At this point, the main obstacle is the absence of good hosts to support the third step. One can :

1. Use the 3L-model

The editing step of the 3L-model is convenient for ADL implementation because the syntactical function can be completely parameterized and it is possible to adjust the concrete syntax. Presently an editor is being made which will be able to support PASCAL, LEM and other languages. The interpreting step offers a good implementing environment for ADLs because it is more versatile, more debuggable and easier to write than a conventional compiling step. Moreover, LEM is an efficient support, in particular, it provides fast interpreting processes.

2. Build a dialect of LISP

With LISP, it is easy to create new functions and to add them to LISP itself with a great uniformity, then yielding a dialect of LISP. Therefore, LISP is a good host for new applications. For example, the artificial intelligence group has defined such a dialect, ARGOS II [26], dedicated to robot control. When they are directly coded in LEM, some of its functions (e.g. MATCH) are about 500 times faster than the original version that runs on a CII-HB IRIS 80 computer.

However, it is not always possible to define dialects of LISP when a specific syntax is needed. It will be the case for the implementation of PLASMA on M3L. For graphical applications, together with specific syntax of the control language, hardware constraints are appearing, e.g. the connection of a specialized processor. In that situation, M3L can handle both software and hardware constraints.

ACKNOWLEDGEMENTS

This work was done at the Paul SABATIER University in the Laboratory of Professor R. BEAUFILS and was sponsored by French INRIA under grant # 79.027.

Special thanks to J.L. DURIEUX for TLISP, to J. CHAILLOUX and P. GREUSSAY for fib.measures, to M. SIRVEN for its MATCH implementation and to H. PRADE who desperatly tried to obtain measures on other LISP implementations.

REFERENCES

[1] Y. CHU - Direct Execution Computer Architecture IFIP Congress - Montreal 1977

[2] L.W. HOEVEL - "IDEAL" directly executable languages : an analytical argument for emulation IEEE Trans.Computers Vol.C-23 - 8 - Aug.1974

[3] W.T. WILNER - B1700 memory utilization FJCC AFIPS,Montvale New Jersey - 1972

[4] P. MAURICE, A.M. COUHAUT - Spécifications d'un éditeur syntaxique LEGOS - IRIA Report - Dec.1980

[5] J. Mc CARTHY - LISP 1.5 Programmer's manual MIT Press - Cambridge - 1962

[6] W. TEITELMAN - INTERLISP Reference Manual XEROX Palo Alto - 1976

[7] G. HUET, G. KAHN, P. MAURICE - Environnement de programmation PASCAL - Ref.Manual - IRIA - 1977

[8] J.P. SANSONNET - The 3L-MODEL, an alternative to the Von Neumann architecture - LSI report #77 Toulouse - Janvier 1980

[9] J.P. SANSONNET, M. CASTAN, C. PERCEBOIS M3L : A List-Directed Architecture - ISCA-7 La Baule - May 1980

[10] J.P. SANSONNET, M. CASTAN, C. PERCEBOIS Architecture of a multi-language processor based on list structured DELs - Intern. Workshop on HLL Computer Architecture - Fort Lauderdale - May 1980

[11] J. CHAILLOUX - Le modèle VLISP : description, implémentation et évaluation - Thèse - Paris VI April 1980

[12] A. LUX - Etude d'un modèle abstrait pour une machine LISP - Thèse - Grenoble - March 1975

[13] M. CASTAN - Etude et définition d'une émulateur : M3L - LSI Report - Toulouse - June 1978

[14] J.C. REYNOLDS - GEDANKEN : A simple typeless language based on the principle of completeness and the reference concept - CACM Vol.13 - 5 - May 1970

[15] P. GREUSSAY - Contribution à la définition interprétative et à l'implémentation des λ-langages - Thèse -Paris VI - 1977

[16] M. CASTAN - Conception et réalisation d'une machine spécialisée dans le traitement des formes arborescentes - Thèse - Toulouse - October 1980

[17] A. NEWELL - IPL-V Manual - 2nd ed.Prentice Hall Englewood N.J. 1964

[18] G.E. COLLINS - A method for overlapping and erasure of lists - CACM - 3 - Dec.1960

[19] H. GERLERNTER et al. - A FORTRAN compiled list processing language - JACM - 7 - April 1960

[20] J. Mc CARTHY - History of LISP - ACM Sigplan Notices - Vol.13 - 8 - August 1978

[21] J. WEIZENBAUM - Symetric list processor CACM Vol.6 - Sept.1963

[22] L.P. DEUTSCH, D.G. BOBROW - An efficient incremental automatic garbage collector - CACM Vol.19 - 9 - Sept.1976

[23] D.W. CLARK, C.C. GREEN - A note on shared list structures in LISP - Information Processing Letter - Vol.7 - 6 Oct.1978

[24] J.L. DURIEUX - Manuel TLISP - LSI Report Toulouse - 1977

[25] J.P.SANSONNET, D.BOTELLA, J.PEREZ - An experience of functions distribution in a list-directed architecture EUROMICRO Journal (1982)

[26] M.CAYROL, B.FADE, H.FARRENY - Formal objects and feature associations in ARGOS II - Proc. 6th IJCAI - Tokyo 1979

[27] G.L.STEEL, G.J.SUSSMAN - Design of a LISP-based microprocessor - CACM Vol.23 - 11 - Nov.1980

[28] A.BAWDEN, R.GREENBLATT, J.HOLOWAY - LISP machine progress report - MIT memo # 444 - 1977

[29] P.L.WADLER - Analysis of an algorithm for real-time garbage collection - CACM Vol.19 # 9 Sept76

[30] G.J.SUSSMAN, J.HOLLOWAY, G.L.STEELE Jr, A.BELL SCHEME 79 - LISP on a chip - Computer July 1981