記号処理 12－12
（1980. 6. 20）

Fortran-based LISP system for REDUCE

Nobuyuki Inada
Information Science Laboratory
The Institute of Physical and Chemical Research
2-1, Hirosawa, Wako-shi, Saitama, 351. Japan

ABSTRACT

In this paper, we describe the implementation of a LISP system, written entirely in Fortran, whose external specification is an extension to the standard LISP report by the university of Utah. The system is designed to be best fitted to efficiently run the REDUCE algebra system and its software family such as a portble LISP compiler, a one pass pretty printer, etc.

1. Introduction

Five years ago, we started development of LISP system specially for efficiently running LISP-based algebra system REDUCE[1]. Since then, several versions of LISP have been in existence among our group, which we call collectively HLISP (H for hash)[2]. Last year, based on the experience with running HLISP and HLISP based REDUCE, we redesigned the whole system for the following purposes:

1) To incorporate several hash based algorithms[3].
2) To systematize the tag handling.
3) To make the system truely portable.
4) To make the system completely compatible with standard LISP[4].
5) To speed up the REDUCE.
6) To provide basis for FLATS[5].

We have had difficulty in implementing the REDUCE on the HLISP system specially at first. When Prof. Hearn, who is a chief implementor of the REDUCE, stayed at our institute, we had attempted to implement a new version of the REDUCE system with Cambridge analytic integrator. At that time, we prepared a front end program in HLISP for establishing standard LISP, but the discrepancy between HLISP and standard LISP was found to be larger. Although we slightly updated the HLISP system, several standard LISP functions could not been supported after all.

The sophisticated LISP systems, e.g. INTERLISP[6] on the PDP 10 family and LISP/360[7] on the IBM 360/370 family, are available only on such specific machines due to their machine dependency, but some portable LISP systems such as HLISP in Fortran are available on several machines at several sites. In fact, we have used HLISP interpreter in Fortran as a host of the REDUCE at our institute, but LISP system in a high level language without decent LISP compilers sometimes takes many hours for a large computation. It is highly desirable that even Fortran-based LISP system provides a LISP compiler to execute the REDUCE with high efficiency.

The REDUCE system and its software family are most coded in RLISP, which is a sweetened version of standard LISP and whose syntax is a subset of the REDUCE. It is pretty preferable that the new interpreter system comforms to the standard LISP report by the university of Utah for the software transportability and for the easy installation of the REDUCE on such a system without the source changed. When we wish to define another data type in addition to standard LISP, such an extension should be done as part of a constant to keep upward compatibility with it.

We may face criticism on use of Fortran as a system writing language, but we dared to adopt Fortran for a LISP interpreter, because the LISP interpreter could anyway be coded entirely in Fortran. Fortran compilers are unversally available on many machines, we can easily use a high level optimizing compiler against other languages. Fortran provides a debugging tool in itself and run-time support subroutines including file systems, and the term to develop the system will be shorter if we can take advantage of the above ones.

2. Description of Interpreter

2.1. Lisp Memory and Data Types

In order to express several data types, a pointer is represented as a tag-information pair. Therefore each pointer has two fields in a word where the tag is a field to denote the data type and the information is either a value with no extra memory requirement or an address pointing to its data structure. The following picture shows a pointer in this implementation.

tag (9 bits)	information (27 bits)

Furthermore both to find out the data type and to make access to it in a short time, we should decide the order of the tag value for data types both in existence and to be added later even if the system is written in a high level language. Reflecting the fact that there are several data representations for a single data type in our implementation, the tag value is defined as follows:

value	tag	information
0	pair	an address pointing to a pair object
1	identifier	an address pointing to an identifier object
2	short string	a value of one to three characters
3	string	an address pointing to a long string object
4	- big float	an address pointing to the object of an absolute big floating point number
5	- short float	an address pointing to the object of an absolute double precision floating number
6	- big integer	an address pointing to the object of an absolute arbitrary precision integer
7	- short integer	a value (-99999999 to -1)
8	+ short integer	a value (0 to 99999999)
9	+ big integer	the same as minus big integer
10	+ short float	the same as minus short float
11	+ big float	the same as minus big float
12	func. pointer	a value with both the number of arguments and a real address of the entry point
13	vector	an address of base of a vector object

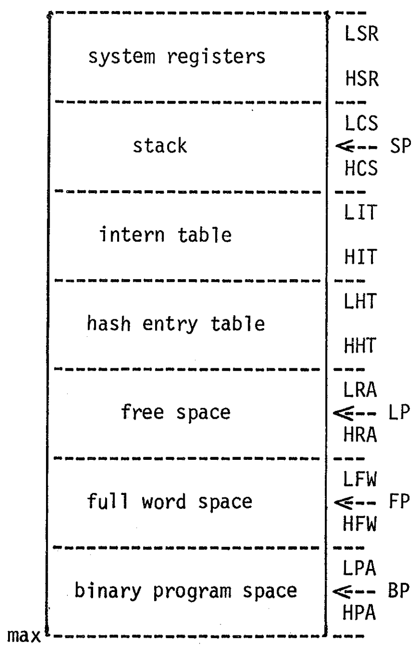
We next introduce a memory organization for LISP in this implementation. The memory layout should be designed for LISP data so as to be handled uniformly and for the size of each separate block so as to be changeable if necessary. We can distinguish data stored in LISP memory into the following three kinds:

1) A pointer mentioned above.
2) A full word of either a string of characters or a floating point number.
3) An instruction code.

In this implementation, since a Fortran one-dimensonal array is used to implement LISP memory the word "address" means an index value of the array element and the words "real address" for a physical address.

Since these blocks are assigned in a single fortran array, the size of each block can be changed when necessary Data in the full word space are non-tagged, and data in the program space are non-tagged when instructions, or tagged otherwise. Data in the other spaces are all tagged even in the stack or in the system registers.

It is designed so that the first three blocks may not be pointed to by any pointers in the system. Therefore if such a pointer is in existence in the system, it is not considered as a root to be marked by the garbage collector. Moreover the address space of the system is larger than that of the current implementation, so the garbage collector does not consider such an address to be a root, either.



The aims to establsh the system registers are as follows:

1) To make it efficient to intern identifiers of a single character print name only with address calculation using its ASCII code.
2) To place variables to communicate with the interpreter, e.g. LAMBDA, NIL, T, QUOTE, COND, PROG etc.
3) To put variables for the compiled program to communicate with the system.

The current version uses 180 entries of system registers, 96 for part of the intern table, 16 for argument regsters, and the rest for variables or place holders to associate with system identifiers.

The system provides a single stack for the following purposes:

1) To store errorset information, controling the system.
2) To store items instead of a list, evaluating arguments.
3) To use as working stores, in LAMBDA or PROG expression.
4) To store a previous value, bindig variables.
5) To store return addresses, linking function.

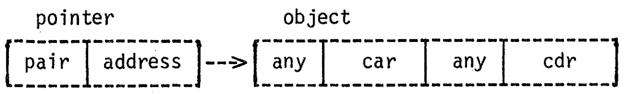
The stack uses more two another tags in addition to those mentioned before. One is a tag to automatically rebind variables to their old values, and the other to return to a caller. In our implementation, even a return address in the stack has a tag.

In the standard LISP report the existence of an intern table, the same terminology as OBLIST, is explicitly defined but its structure never defined in it. HLISP and INTERLISP do not have such an intern table, and therefore all the identifiers are uniquely represented other than standard LISP. However in this implementation we provides the intern table to support interned and not interned identifiers as one of the design principles. We employ an open addressed hash method to intern an identifier, and the key to generate a hash sequence is a print name associated with the identifier, not a pointer to the identifier at all. A single character identifier is not interned in this table, but interned in system registers by its ASCII code value. The current version accepts more than 2000 different identifiers with the size of this table remaining unchanged. Since there are no pointers pointing to the intern table in the system, we can reuse an entry deleted by REMOB if the hash index of the key reaches the empty cell after keeping an address of a deleted cell in first probing, and we can expect a shorter probe.

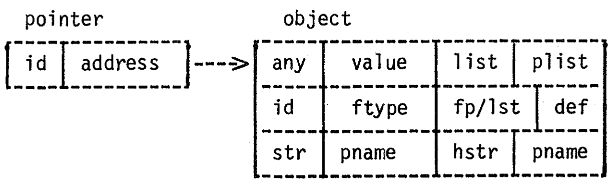
The hash entry table is used to establish a uniquely represented pointer for any S-expression in the system. So the pointer to this table is protected from the relocation of it. We proposed to add the associative features by hashing to the standard LISP two years ago. The standard LISP does not subpport any unique representated data in itself, and this table is prepared positively for such a future extension at present.

In our implementation, the free space keeps several types of LISP objects including a traditional car-cdr pair. We introduce arbitrary length of contiguous block to realize a vector type data structure whose elements are randomly accessed by the address calculation. As a result, we need a genetic order compactifying garbage collector[8]. In order to make best use of the effectiveness of the hardware buffer (cache) memory, each object starts at double word boundary.

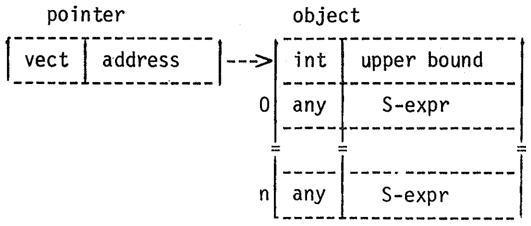
The object dotted-pair, pair for short, requires 2 words to construct its car and cdr part. The pointer and the object of the pair is illustrated in the following diagram:



The object identifier, id for short, requires 6 words to store a bound value, its property list, function type and definition if any, and its print name. The pointer and the object of the identifier are illustrated in the following diagram:



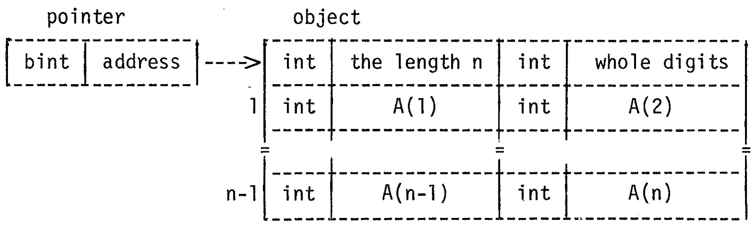
The object vector is taken as a contiguous block in the free space. The access to any item is only done by the vector handling functions defined in standard LISP. If the size of the object is an odd number, the successive one word is not used for the sake of the hardware time efficiency. The upper bound value is stored in the first item of the vector object. The pointer and the object of the vector are illustrated in the following diagram:



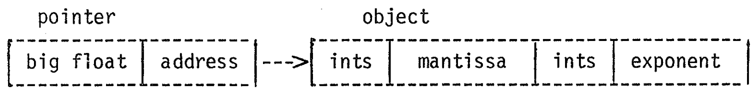
The object of a big integer is implemented like a vector structure. The big integer enables us to calculate with an arbitrary number overflow free or not chopped. In this implementation we made the base 10 to the 8th rather than 2 to the 27th. The magnitude of the big integer can be expressed by:

A(1) + A(2)*10**8 + ... + A(n-1)*10**(8*(n-2)) + A(n)*10**(8*(n-1)).

The pointer and the object of the big integer are illustrated in the following diagram:

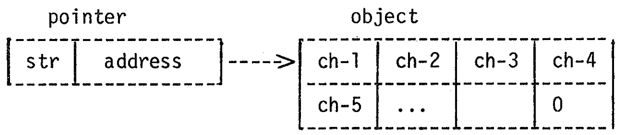


The object of a big float is represented as a mantissa-exponent pair like a dotted-pair, and therefore requires 2 words of memory to store the mantissa in the left part and the exponent in the right. The system handles the mantissa and the exponent in an arbitrary precision by use of a big integer. The pointer and the object of the big float are illustrated in the following diagram:



The tag "ints" represents either a short integer or a big integer. In this system the sign of the number is stored in the tag. We next introduce a short integer with no memory requirement. The above numbers are based on an absolute value representation, but the short integer based on the two's complement value for the sake of the time efficiency on arithmetic operations.

The full word space is used to store non-tassed data such as string and machine depended floating point number. The string object is used either as a string data type or as a print name of identifiers. The pointer and the object of the string are illustrated in the following diagram:



As known from the diagram, the length of the string is arbitrary and the string terminates with a null code of ASCII. This simple structure is considered reasonable as described in the book by Allen[9]. If accesses to strings are so frequent, we should supply another structure, e.g. a bitwised array in the full word space.

A double precision floating point is a machine dependent representation in the following diagram:

```
     pointer                  object
[ sfloat | address ]-->[s| mantissa (62 bits)          | exponent(9) ]
```

A function pointer can be illustrated in the following diagram:

```
[ funcp | # of args | real address ]
```

The second field is 9 bits in length and used to represent the number of arguments passed to this function, and the third 18 bits used as a real address pointing to the entry of this function.

The last contiguous block is a binary program space to store instructions compiled by the portable LISP compiler and loaded by a LAP75 loader. The instruction format of the FACOM 230-75 is as follows:

```
[  op   | b | x |    opr      ]
'----15----'-3-'----18------'
```

In this implementation, we use the first 15 bits as an operation code, the second 3 bits as an index register, and the last 18 bits as an operand.

2.2. Fast Predicates Evaluation

Reflecting the advantage of a tag-information pair, most predicate functions corresponding to data types can be implemented as follows:

```
U = PP ( GETTAG (U) ),
```

where GETTAG is an open coded statement function to set a tag as an integer, and PP is an array whose name is corresponding to each predicate.

In Fortran the time to execute the above statement is about 3 microseconds on the FACOM 230-75, but in assembly language the time can be reduced to only 0.3 microseconds by using a shift operation and a register pre-assigned. Each element of the array according to predicates is set to either T or NIL at initial phase of the system.

However, in the current system we use the following codes for some predicates from the reason that the existing predicates are a few in number:

```
IF(U .GE. ID)  GO TO 1008
               GO TO 1006,
```

or

```
IF(U .LT. ID)  GO TO 1008
               GO TO 1006,
```

where this is an implementation for PAIRP, and that is one for ATOM.

2.3. Two Level Intern Table

As stated before, we made the intern operation effective by using two tables. Especially, it is very frequent to make access to the single character intern table by READCH and EXPLODE functions and to invoke functions such as DIGIT and LITER in large applications such as REDUCE and language processor. In our implementation, the READCH function is equivalent to:

```
CAR(TTIN()),
```

where TTIN() returns the next character in ASCII code.

DIGIT and LITER are implemented only by the comparison of both the value of the argument and the appropriate position of the intern table, because the address of the object of the single indentifier is ordered by its ASCII code.

2.4. Fast Evaluation

EVAL, APPLY and EVLIS are implemented as a single Fortran subroutine to realize a recursive call and to rapidly evaluate LISP functions and others. We here observe some methods or techniques important to speed up LISP.

We can prepare some pointers without evaluating recursively, e.g. constants, the value of a variable or a quoted S-expr (QUOTE <any>) is directly handled at the step before invoking EVAL, and the other functions are evaluated within EVAL through a secondary entry not as to reevaluate arguments shown the above. In this implementation, we use the stack to spread arguments to a function, not using EVLIS and the arguments list. EVAL, APPLY and EVLIS call each other directly eliminating tail recursions but the effectiveness is guaranteed.

We use only three function types such as EXPR, FEXPR and MACRO, and even for compiled functions we use EXPR and FEXPR. The function pointer to represent the entry point of the function is the only one kind, in other words the function pointer to built-in functions is an address within the Fortran program and the pointer to compiled function the location in binary program space within Fortran array.

A PROG interpreter within EVAL interpreter also employs some fast techniques; one is an association of a go-list and another is that some very frequently used functions such as SETQ, COND, GOTO and RETURN are implemented to directly jump to their entry without invoking EVAL at a top level in PROG. When a label reference takes place by executing GO and the label is not the first element of go-list, the system changes go-list and puts the label at the first place.

3. Compiler system

The portable LISP compiler[10] is available from the university of Utah and the virtue of the choice of macros as object codes enabled us to implement the compiler system only for a few weeks. We only prepared the programs of MAC75 macro expansion and LAP75 loader for the FACOM 230-75. In this compiler implementation, we did not optimize macro expansion except LINK macro, for the interpreter could use only a accumulator and a index register.

The detail description is omitted due to space consideration.

4. Performance evaluation

The measurement of the system performance is often apt to bias an implementor's choice, but a just and fair evaluation is very difficult. Someone might say the measurement was just nonsense. However we dared to measure the performance only in terms of execution time, because the current system has been intended to be a host of the REDUCE.

We prepare a function for the measurement in RLISP suntax.

```
SYMBOLIC PROCEDURE FF(X,Y,Z);
  IF X>Y THEN FF( FF(X-1,Y,Z),FF(Y-1,Z,X),FF(Z-1,X,Y) )  ELSE Y;
```

We obtained the ratio of 17 when compared with both Intepreter and Compiler in case of FF where the functions GREATERP and SUB1 were expanded as open subroutines.

count of FF calls		Interpreter	Compiler	ratio(i/c)
FF(6,3,0)	673	119 ms	7 ms	17
FF(8,4,0)	12605	2234 ms	132 ms	17

We next observe the case of more use of built-in functions. We use built-in functions of APPEND, EQUAL, LIST, CONS, CAR, CDR and GREATERP, and user defined functions of SMALLER, MINM, DIFFLIST and SEQUENCE.

```
SYMBOLIC PROCEDURE SEQUENCE L;
    BEGIN SCALAR U,V;
      U := L;  V := MINM L;  W := NIL;
  A;  IF NULL U THEN RETURN W;
      V := MINM U;  U := DIFFLIST(V,U);
      W := APPEND(W,LIST V);     GO A
    END;

SYMBOLIC PROCEDURE SMALLER(X,Y);
    IF Y>X THEN X ELSE Y;
```

```
SYMBOLIC PROCEDURE DIFFLIST(A,X);
    IF NULL X THEN NIL
    ELSE IF A=CAR X THEN DIFFLIST(A,CDR X)
    ELSE CAR X . DIFFLIST(A,CDR X);

SYMBOLIC PROCEDURE MINM L;
    IF NULL L THEN NIL
    ELSE IF NULL CDR L THEN CAR L
    ELSE SMALLER(CAR L,MINM CDR L);
```

number of items	Interpreter	Compiler	ratio(i/c)
20	115 ms	14 ms	8.2
40	420	50	8.4
60	683	78	8.8
80	979	115	8.5
100	1362	148	9.2

5. Concluding Remarks

We described the detail of the Fortran-coded LISP system for the REDUCE. Since the system was designed to comform to the standard LISP report by the University of Utah, the REDUCE and its software family could be run on this system with higher efficiency than we had expected. To use Fortran as a system writing language enabled us to develop the LISP system only for three man-months. We could provide a compiler about ten times faster than interpretive execution on our Fortran LISP system.

The current system has more than 200 system functions including all the standart LISP functions and if only the interpreter is coded in a low level laguage, it is considered that the speed of the system will become about two to three times faster and the memory occupation will reduce two third to three fourth. The need for a large computation using the algebra system will grow larger, this system will be best suited for such needs.

Acknowledgement

The author would like to express his gratitude to Prof. E. Goto and the other members of the FLATS project for valuable comments and lots of encouragements.

References

[1] Hearn, A.C., "REDUCE 2 User's Manual", second ed., UCP-19, Univ. of Utah, (1973).
[2] Goto, E., "Monocopy and associative algorithms in an extented LISP", Information Science Lab., Tech. Rept. 74-03, Univ. of Tokyo, (Apr. 1974).
[3] Goto, E. et al., "FLATS, a machine for numerical, symbolic and associative computing", Proc. of the 6th annual symp. on Computer architecture, IEEE, (Apr. 1979).
[4] Marti, J. et al., "Standard LISP report", UCP-60, Univ. of Utah, (Jan. 1978).
[5] "Report of the FLATS Project", Information Science Lab. RIKAGAKU KENKYUSHO, (Oct. 1978).
[6] Teitelman, W., "INTERLISP Reference Manual", Xerox Palo Alto Center, (Oct. 1974).
[7] "LISP/360 Reference Manual", Stanford Computation Center, Stanford Univ., SCC024, (Mar. 1972).
[8] Terashima, T. et al., "Genetic order and compactifying garbage collectors", Information processing Letter Vol. 7, No. 1, pp. 27-32, (Jan. 1978).
[9] Allen, J.R., "Anatomy of LISP", McGraw-Hill, New York, (1978).
[10] Griss, M.L. et al., "A portable LISP compiler", UCP-76, Univ. of Utah, (Jun. 1979).
[11] Takeuchi, I., "report of the second LISP contest", IPSJ SIGSAM, (1978).