```
**************************************************
**************************************************
***                                            ***
***  Name:                                     ***
***                                            ***
***  Project:    1       Programmer:  MWK      ***
***                                            ***
***  File Name: REWRIT.DOC[L70,TES]            ***
***                                            ***
***  File Last Written:   9:19 20 Feb 1973     ***
***                                            ***
***  Time: 18:45          Date: 15 Jul 1973     ***
***                                            ***
***            Stanford University             ***
***      Artificial Intelligence Project       ***
***        Computer Science Department         ***
***           Stanford,  California            ***
***                                            ***
**************************************************
**************************************************
```

THE LISP70 PATTERN MATCHING SYSTEM

Lawrence G. Tesler
Horace J. Enea
David C. Smith

Stanford University

February, 1973

ABSTRACT

LISP70 is a descendant of LISP which emphasizes pattern-directed computation and extensibility. A function can be defined by a set of pattern rewrite rules as well as by the normal LAMBDA method.. New rewrite rules can be added to a previously defined function; thus LISP70 functions are said to be "extensible". It is possible to have the new rules merged into the function automatically so that special cases are checked before general cases. Some of the facilities of the rewrite system are described and a variety of applications are demonstrated.

BACKGROUND

During the past decade, LISP [16] has been a principal programming language for artificial intelligence and other frontier applications of computers. Like other widely used languages, it has spawned many variants, each attempting to make one or more improvements. Among the aspects that have received particular attention are notation [1,10,14,21], control structure [4,11,17,19], data base management [12,17,22], interactive editing and debugging [24], and execution efficiency.

A need for a successor to LISP has been recognized [3], and several efforts in this direction are under way. The approach being taken with TENEX-LISP is to begin with an excellent debugging system [23] and to add on flexible control structure [2] and Algol-like notation. The approach taken by LISP70 and by the LISP-related ECL [26] is to begin with an extensible kernel language which users can tailor and tune to their own needs.

"Tailoring" a language means defining facilities which assist in the solution of particular kinds of problems which may have been unanticipated by the designers of the kernel. "Tuning" a language means specifying more efficient implementations for statements which are executed frequently in particular programs.

A language that can be used on only one computer is not of universal utility; the ability to transfer programs between computers increases its value. However, a language that is extensible both upward and downward is difficult to transport if downward extensions mention machine-dependent features [7,8]. LISP70 generates code for an "ideal LISP machine" called "ML" and only the translation from ML to object machine language is machine-dependent. Thus, downward extensions can be factored into a machine-independent and a machine-dependent part, and during program transfer, the machine-dependent recoding (if any) is clearly isolated.

Among the improvements LISP70 makes to LISP are backtrack control structure [19], streaming [15], pattern-directed computation, and extensible functions.

The subjects to be covered in the present paper are pattern-directed computation and extensible functions.

PATTERN-DIRECTED COMPUTATION

Many of the data tranformations performed in LISP applications are more easily described by pattern matching rules than by algorithms [12,17,22,25]. In addition, pattern matching rules are appropriate for the description of input-output conversion, parsing, and compiling [20]. LISP70 places great emphasis on "pattern rewrite rules" [5,6,13,27] as an alternative and adjunct to algorithms as a means of defining functions.

A brief explanation of rewrite rule syntax and semantics will be presented with some examples to demonstrate the clarity of the notation.

Each rule is of the form DEC→REC. The DEC (decomposer) is matched against the "input stream". If it matches, then the REC (recomposer) generates the "output stream".

A literal in a pattern is represented by itself if it is an identifier or number, or preceded by a quote (') if it is a special character.

```
RULES OF SQUARE =
        2 → 4,
        5 → 25,
        12 → 144 ;
```

A private variable of the rule is represented by an identifier prefixed by a colon (:); it may be bound to only one value during operation of the rule.

```
RULES OF EQUAL =
        :X :X → T,
        :X :Y → NIL ;
```

A list is represented by a pair of parentheses surrounding the representations of its elements. A segment of zero or more elements is represented by an ellipsis symbol (...).

```
RULES OF CAR =
        (:X ...) → :X ;

RULES OF CDR =
        (:X ...) → (...) ;

RULES OF CONS =
        :X (...) → (:X ...) ;

RULES OF ATOM =
        (:X ...) → NIL,
        :X → T ;

RULES OF APPEND =
        (...) (...) → (... ...) ;
```

If a segment needs a name, it is represented by an identifier prefixed by a double-colon (::).

```
RULES OF ASSOC =
        :X (... (:X ::Y) ...) → (:X ::Y),
        :X (...) → NIL ;
```

A function F can be called in a pattern, passing it a single argument ARG, by the construct: ARG@F (there are also ways of passing several arguments to a function).

```
RULES OF LENGTH =
        ( )         →  0,
        (:X ...)    →  (...) @LENGTH @ADD1 ;
```

LIST STRUCTURE TRANSFORMATIONS

The following set of rules defines a function MOVE_BLOCK of three arguments: a block to be moved, a location to which it should be moved, and a representation of the current world. The function moves block :B from its current location in the world to location :TO, and the transformed representation of the world is returned.

```
RULES OF MOVE_BLOCK =

        :B :TO (... (:TO ... :B ...) ...)
             → (... (:TO ... :B ...) ...),

        :B :TO (... (... :B ...) ... (:TO ...   ) ...)
             → (... (...    ...) ... (:TO ... :B) ...),

        :B :TO (... (:TO ...   ) ... (... :B ...) ...)
             → (... (:TO ... :B) ... (...    ...) ...),

        :B :TO (... (... :B ...) ...)
             → (... (...    ...) ... (:TO :B)),

        :B :TO (...)
             → (BLOCK :B NOT IN (...)) @ERROR ;
```

In the first case, the block is already where it belongs, so the world does not change; in the second, the block is moved to the right; in the third, to the left; in the fourth, the location :TO does not exist yet and is created; in the last case, :B is not in the world and the ERROR routine is called.

Functions such as MOVE_BLOCK have been used in a simple planning program written by one of the authors. Imagine writing MOVE_BLOCK as

an algorithm; it would require the use of auxiliary functions or of a PROG with state variables and loops. Bugs would be more likely in the algorithm because its operation would not be so transparent.

REPLACEMENT

If F is any function, then the construct <F> occurring in a DEC pattern signifies "replacement". This means that F is invoked to translate a substream of the input stream, and that substream is replaced by its translation. The altered input stream can then continue to be matched by the pattern to the right of <F>.

The following example is from the MLISP compiler, which calls itself recursively to translate the condition and arms of an IF-statement to LISP:

```
RULES OF MLISP =

        IF <MLISP>:X THEN <MLISP>:Y ELSE <MLISP>:Z
                → (COND (:X :Y) (T :Z)),

        IF <MLISP>:X THEN <MLISP>:Y
                → (COND (:X :Y) (T NIL)),

        IF <MLISP>:X
                → (MISSING THEN) @ERROR,

        IF      → (ILLEGAL EXPRESSION AFTER IF) @ERROR ;
```

Here is another example. The predicate PALINDROME is true iff the input stream is a mirror image of itself, i.e., if the left and right ends are equal and the middle is itself a palindrome.

```
RULES OF PALINDROME =   :X                  →      T,
                      :X :X                 →      T,
                :X <PALINDROME>T :X         →      T,
                        ...                 →      NIL ;
```

The replacement facility provides both the non-terminal symbols of syntactic parsing and the "actors" of PLANNER [12].

EXTENSIBLE FUNCTIONS

New rules may be added to an existing set of rewrite rules under program control; thus, any compiler table or any other system of rewrite rules can be extended by the user. For this reason, a set of rewrite rules is said to be an "extensible function". The "ALSO" clause is used to add cases to an extensible function:

```
RULES OF MLISP ALSO =

        IF <MLISP>:X THEN <MLISP>:Y ELSE
                → (MISSING EXPRESSION AFTER ELSE) @ERROR,

        IF <MLISP>:X THEN
                → (MISSING EXPRESSION AFTER THEN) @ERROR ;
```

Extensions can be made effective throughout the program or only in the current block, as the user wishes.

A regular LAMBDA function can also be extended. Its bound variables are considered analogous to a DEC and its body analogous to a REC. Accordingly, the compiler converts it to an equivalent rewrite function of one rule before extending it.

THE EXTENSIBLE COMPILER

To make an extensible compiler practical, the casual user must be able to understand how it works in order to change it. To demonstrate that it is not inordinately difficult to understand the LISP70 compiler, those rules which get involved in translating a particular statement from MLISP to LAP/PDP-10 are shown below. A simplified LISP70 (typeless and unhierarchical) is used in the examples, but the real thing is not much more complicated. The statement to be translated is:

```
IF A < B THEN C ELSE D
```

The rules invoked in the MLISP-to-LISP translator are:

```
RULES OF MLISP =

        IF <MLISP>:X THEN <MLISP>:Y ELSE <MLISP>:Z
                        → (COND (:X :Y) (T :Z)),

        :X  '<  :Y      →       (LESSP :X :Y),

        :VAR            →       :VAR ;
```

The LISP-to-ML compiler below utilizes the following feature: if a colon variable occurs in the REC but it did not occur in the DEC, an "existential value" (which is something like a generated symbol) is bound to it. Here, the existential value is used as a compiler-generated label.

```
RULES OF COMPILER =

        (COND (T :E))
              → :E @COMPILER,

        (COND (:B :E) ...)
              → :B @COMPILER
                (BRANCH_FALSE :ELSE)
                :E @COMPILER
                (BRANCH :OUT)
                (LABEL :ELSE)
                (COND ...) @COMPILER
                (LABEL :OUT),

        (LESSP :A :B)
              → :A @COMPILER
                (PUSH_DOWN)
                :B @COMPILER
                (COMPARE LESS),

        :V    → (LOAD :V) ;
```

The ML-to-LAP translator below assumes that the value register of the ideal machine is represented on the PDP-10 by a register named "VAL", that there is a single stack based on register "P", and that variables can be accessed from fixed locations in memory.

```
RULES OF ML =

        (BRANCH_FALSE :LBL)
              → (JUMPE VAL :LBL),

        (BRANCH :LBL)
              → (JRST :LBL),

        (LABEL :LBL)
              → :LBL,

        (PUSH_DOWN)
              → (PUSH P VAL),

        (COMPARE LESS)
              → (CAMGE VAL 0 P)
                (TDZA VAL VAL)
                (MOVEI VAL 1)
                (POP P),

        (LOAD :V)
              → (MOVE VAL :V) ;
```

The code generated is thus:

```
        (MOVE VAL A)
        (PUSH P VAL)
        (MOVE VAL B)
        (CAMGE VAL 0 P)
        (TDZA VAL VAL)
        (MOVEI VAL 1)
        (POP P)
        (JUMPE VAL E0001)
        (MOVE VAL C)
        (JRST E0002)
E0001   (MOVE VAL D)
E0002
```

Peephole optimization guided by another rewrite function can reduce this to six instructions.

AUTOMATIC ORDERING OF REWRITE RULES

In most pattern matchers, candidate patterns to match an input stream are tried either in order of appearance on a list or in an essentially random order not obvious to the programmer. LISP70 tries matches in an order specified by an "ordering function" associated with each set of rewrite rules.

One common ordering is "BY APPEARANCE", which is appropriate when the programmer wants conscious control of the ordering. Another is "BY SPECIFICITY", which is useful in left-to-right parsers and other applications where the compiler can be trusted to order the rules so that more specific cases are tried before more general ones. When neither of these standard functions is appropriate, the programmer can define and use specialized ordering functions, or can extend SPECIFICITY to meet the special requirements.

Automatic ordering is a great convenience for a user who is extending a compiler, a natural language parser, or an inference system. It can eliminate the need to study the existing rules simply to determine where to position a new rule. Ordering functions can also be designed to detect inconsistencies and ambiguities and to discover opportunities for generalization of similar rules.

As an example, take the LISP-TO-ML translator "COMPILER", which includes the following rule for the intrinsic function PLUS (slightly simplified for presentation):

```
RULES OF COMPILER =

        (PLUS :X :Y)
              → :X@COMPILER
                (PUSH_DOWN)
                :Y@COMPILER
                (ARITHMETIC ADD) ;
```

To add special cases to the compiler for sums including the constant zero, the user could include the following declaration in a program:

```
RULES OF COMPILER ALSO =

        (PLUS :X 0) → :X@COMPILER,

        (PLUS 0 :X) → :X@COMPILER ;
```

The compiler is ordered by SPECIFICITY, which knows that the literal 0 is more specific than the variable :X or :Y. Therefore, both of the new rules would be ordered before the original PLUS rule. Suppose the added rules were placed after the general rule; then the original rule would get first crack at every input stream, and sums with zero would not be processed as special cases.

AN ORDERING FUNCTION

The complete definition of the ordering function SPECIFICITY is beyond the scope of this paper. It works roughly as follows. Comparing DEC patterns by a left-to-right scan, it considers literals more specific than variables, a colon variable at its second occurrence more specific than one at its first occurrence, and a function call with an "@" more specific than a variable but less specific than a literal. The specificity of a replacement <F> is that of the most general rule in the function F.

A DEC with an ellipsis is considered to expand to multiple rules in which the ellipsis is replaced by 0, 1, 2, 3, ... ∞ consecutive variables. The specificity of each expanded rule is considered separately. Observe that between two expansions of an elliptic rule some other rewrite rule of intermediate specificity may lie.

Example:

```
RULES OF SILLY =
        A ... B ... C   →       1,
        A B :X :Y       →       2 ;
```

Two of the expansions of the first rule are:

```
        A B :X C        →       1,
        A :Z B C        →       1,
```

and the second rule of SILLY comes between these in specificity.

SPECIFICITY is itself defined by a system of rewrite rules. To give a flavor of how this is done, a very simplified SPECIFICITY will be defined. It takes two arguments (DEC patterns translated to LISP notation) and returns them in the proper order.

```
RULES OF SPECIFICITY =

        (COLON :V) (LITERAL :L)
                → (ORDER (LITERAL :L) (COLON :V)),

        (LITERAL :L) (COLON :V)
                → (ORDER (LITERAL :L) (COLON :V)) ;
```

OTHER FACILITIES AND APPLICATIONS

Other facilities of the rewrite system include side-conditions, conjunctive match, disjunctive match, non-match, repetition, evaluation of LISP and MLISP expressions, look-ahead, look-behind, and reversible rules.

Out of rewrite functions, it is easy to define systems of inference rules, assertions, and beliefs. LISP70 has facilities for retrieving either all or the first of the assertions in a set of rules that match a given pattern.

Rewrite rules are a great help in natural language understanding, whether the methods used are based on phrase structure grammar, features, keywords, or word patterns. A use of LISP70 with the latter method is described in a companion paper [9].

CONCLUSIONS

Many of the design decisions of LISP70 are contrary to trends seen in other "successors to LISP". The goals of these languages are similar, but their means are often quite diverse.

Concern with good notation does not have to compromise the development of powerful facilities; indeed, good notation can make those facilities more convenient to use.

Emphasis on pattern-directed computation does not overly constrain the programmer accustomed to writing algorithms. Rewrites and algorithms can be mixed, and the most appropriate means of defining a given function can be selected.

LISP70 does not limit the use of pattern rewrite rules to a few facifities like goal-achievement and assertion-retrieval. A set of rules can be applied to arguments like any other function, and can stream data from any type of structure or process to any other.

Automatic ordering does not prevent the programmer from seizing control, but allows him to relinquish control to a procedure of his choosing to save him tedious study of an existing program when making extensions.

The LISP70 kernel is being debugged on a PDP-10 at the time of this writing (February, 1973). The language has already been used successfully in programs for question-answering and planning. After the kernel can reliably compile itself, extensions are planned to improve its control structure, editing, and debugging capabilities, and versions may be bootstrapped to other computers.

REFERENCES

[1] Abrahams, P. W. et al, "The LISP 2 Programming Language and System", Proc. AFIPS FJCC 29 (1966), 661-676

[2] Bobrow, D. G. and Wegbreit, B., "A Model and Stack Implementation of Multiple Environments", Report No. 2334 (March 1972), Bolt, Beranek, and Newman

[3] Bobrow, D. G., "Requirements for Advanced Programming Systems for List Processing", Comm. ACM 15, 7 (July 1972), 618-627

[4] Burstall, R.M., Collins, J.S. and Popplestone, R.J., Programming in Pop-2, University Press, Edinburg, Scotland (1971), 279-282

[5] Colby, K. M. and Enea, H., "Heuristic Methods for Computer Understanding of Natural Language in Context Restricted On-Line Dialogues", Math. Biosciences 1 (1967), 1-25

[6] Colby, K. M., Watt, J., and Gilbert, J. P., "A Computer Method of Psychotherapy", J. of Nervous and Mental Disease 142 (1966), 148-152

[7] Dickman, B. N., "ETC: An Extensible Macro Based Compiler", Proc. AFIPS SJCC 38 (1971), 529-538

[8] Ruby, J. J., "Extensible Languages: A Potential User's Point of View", in [18], pp.137-140

[9] Enea, H., Colby, K. M., and Moravec, H., "Idiolectic Language-Analysis for Understanding Doctor-Patient Dialogues", (submitted to IJCAI)

[10] Enea, H., "MLISP (IBM 360/67)", Computer Science Technical Report CS 92 (1968), Stanford University

[11] Hewitt, C., PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot, Ph.D. Thesis (Feb 1971), MIT

[12] Hewitt, C., "Procedural Embedding of Knowledge in PLANNER", Proc. IJCAI 2 (1971), 167-182

[13] Kay, A., "FLEX, A Flexible Extendible Language", CS Tech. Report (1968), U. of Utah

[14] Landin, P. J., "The Next 700 Programming Languages", CACM 9, 3 (March 1966), 157-166

[15] Leavenworth, B. M., "Definition of Quasi-Parallel Control Functions in a High-Level Language", Proc. Int'l. Comp. Symp. (Bonn, 1970)

[16] McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I", Comm. ACM 3, 4 (April 1960), 184-195

[17] Rulifson, J. F., Waldinger, R. J., and Derksen, J. A., QA4, A Language for Writing Problem-Solving Programs, Proc. IFIP (1968), TA-2, 111-115

[18] Schuman, S., ed., "Proceedings of the International Symposium on Extensible Languages", ACM SIGPLAN Notices 6, 12 (Dec. 1971)

[19] Smith, D. and Enea, H., "Backtracking in MLISP-2", (submitted to IJCAI)

[20] Smith, D. and Enea, H., "MLISP2 -- A Programming Language for Writing and Debugging Translators", (forthcoming)

[21] Smith, D., "MLISP (PDP-10)", Artificial Intelligence Memo No. 135, Stanford University, Oct. 1970

[22] Sussman, G. J. and McDermott, D. V., "Why Conniving is Better than Planning", Proc. AFIPS FJCC 41 (1972), 1171-1180

[23] Teitelman, W. et al, BBN-LISP Reference Manual, (July 1971), Bolt, Beranek, and Newman

[24] Teitelman, W., "Toward a Programming Laboratory", Proc. IJCAI 1 (1969), 1-8

[25] Teitelman, W., Design and Implementation of FLIP, a LISP Format Directed List Processor, Scientific Report No. 10 (July 1967), Bolt, Beranek, and Newman

[26] Wegbreit, B., "The ECL Programming System", Proc. AFIPS FJCC 39 (1971), 253-262

[27] Weizenbaum, J., "ELIZA -- A computer Program for the Study of Natural Communication Between Man and Machine", Comm. ACM 9, 1 (Jan. 1966), 36-45",