THE LISP70 PATTERN MATCHING SYSTEM*

Lawrence G. Tester**, Horace J. Enea, David C. Smith

Department of Computer Science
Stanford University
Stanford, California

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 a LISP70 function is said to be "extensible". It is possible to have new rules merged in automatically such 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[18] 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,[1,12,17,23] control structure,[5,14,20,22] data base management,[l5,20,25] interactive editing and debugging,[27] 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-USP is to begin with an excellent debugging system[26] and to add on flexible control structure.[2] The approach taken by LISP70 and by ECL[29] 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.[8,9] This consideration suggests the use of a machine-independent low-ievel language[4] in terms of which to describe downward extensions.

Capabilities of LISP70

The aim of LISP70 is to provide a flexible and parsimonious programming medium for symbolic processing and an efficient implementation for that medium on several machines,

The semantics of the LISP70 kernel subsumes LISP 1 .5 and Algol-60 semantics. The syntax provides three high-level notations: S-expressions, Algol-like MLISP expressions, and pattern-directed rewrite rules. The syntax and semantics can both be extended as described later in this paper. By extension, it is feasible to incorporate the capabilities of virtually any other programming language. Of course, one would take advantage of the techniques developed by its previous implementors; LISP70 simply provides a convenient rnedijm for doing this.

To maximize efficiency and to eliminate the possibility of an inconsistent compiler and interpreter, all programs in LISP70 are compiled. There is no interpreter in the usual sense; the function EVAL compiles its argument with respect to the current environment and then executes the machine-language code. To extend the language, extensions need only be made to the compiler, not also to an interpreter.

One disadvantage ot a compiler is that certain sophisticated debugging techniques such as the "BREAKIN" of TENEX-LISP[26] are more difficult to implement than in an interpreter. However, we feel that the extra effort needed for this is worth expending to retain the advantages of a compiler.

LISP70 generates code for an "ideal LISP machine" called "ML" end 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 receding (if any) is clearly isolated An execution

* This work was supported (in part) by Grant PHS MH 06645-11 from the National Institute of Mental Health, and (in part) by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183).
The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, NIMH, or the U.S. Government.

** Present affiliation: Xerox Corporation (Palo Alto Research Center).

image on one computer could be transliterated to ML and transported to a different machine. This capability could be used to transport programs around computer networks, and for bootstrapping of the compiler itself.

In order to execute the EVAL function, the compiler and parts of the symbol table must be present during execution. This requirement and the goal of extensibility are met by a pattern-directed translator whose rules are compiled into dense and efficient code. The same pattern matcher as used in the translator also is available for goal-directed procedure invocation in A.I. programs,

Among the specific improvements LISP70 makes to LISP are backtrack and coroutine control structure, streaming, long-term memory for targe data bases, data typing, pattern-directed computation, and extensible functions. The implementation provides dynamic storage allocation, relocation, and segmentation.

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

Pattern Directed Computation

Rewrite Rules

Many of the data tranformations performed in LISP applications are more easily described by pattern matching rules than by algorithms.[11,13,15,20,25,28] In addition, pattern matching rules are appropriate for the description of input-output conversion, parsing, and compiling.[2]" LISP70 pieces great emphasis on "pattern rewrite rules"[6,7,16,30] 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 eilipsis 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(X,Y,Z) can be called in a pattern by the construct: <F :X :Y :Z>.

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

The facilities described so far are standard in most pattern-directed languages.

List Structure Transformations

The following set of rules defines a function M0VE_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 :T0, 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 (...)
     → <ERROR (BLOCK :B NOT IN (...))> ;
```

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 M0VE_BL0CK have been used in a simple planning program written by one of the authors. Imagine writing M0VE_BL0CK 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 lucid.

Replacement

A function call in a DEC pattern is called a "replacement", A replacement has two interesting aspects. First, if the function requires more arguments than it is passed, it will take additional arguments off the front of the input stream. Furthermore, the value returned by a replacement is appended to the front of the input stream. Thus, the replacement <F> behaves like a non-terminal symbol of a top-down parser. In effect, the function 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
       → <ERROR (MISSING THEN)>,

   IF  → <ERROR (ILLEGAL EXPRESSION AFTER IF)>;
```

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;
```

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
       → <ERROR (MISSING EXPRESSION AFTER ELSE)>,

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

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. We have found this to be no problem with users of MLISP2, the predecessor to LISP70. Its extensible compiler has been used to write parsers quickly by A.I. researchers previously unfamiliar with parsing techniques.

To demonstrate that it is not inordinately difficult to understand the LISP70 compiler, those rules which get involved in translating a particular statement from MUSP to LAP/PDP-10 are shown below. A simplified LISP70 (typeles6 and unhierarchical) is u:ed 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 translation is thus:

```
(COND ((LESSP A B) C) (T D))
```

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 symbel) is bound to it. Here, the existentiai value is used as a compiler-generated label.

Trie language ML is based on the machine language of the Burroughs 5000 and its descendants. For example, the ML operator "DJUMPF" means "destructive jump if false". It jumps only if the top of the stack is false but always pops the stack.

```
RULES OF COMPILE =

        (COND (T :E))
              → <COMPILE :E>,

        (COND (:B :E) ...)
              → <COMPILE :B>
                (DJUMPF :ELSE)
                <COMPILE :E>
                (JUMP :OUT)
                (LABEL :ELSE)
                <COMPILE (COND ...)>
                (LABEL :OUT),

        (LESSP :A :B)
              → <COMPILE :A>
                <COMPILE :B>
                (FETCH (FUNCTION LESSP)),

        :V    → (FETCH (VARIABLE :V)) ;
```

The unoptimized ML-to-LAP translator below assumes that the stack of the ideal machine is represented on the PDP-10 by a single stack based on register "P", that there is a single working register "VAL", and that variables can be accessed from fixed locations in memory. (None of this is really true in the actual implementation.)

```
RULES OF ML =

        (DJUMPF :LBL)
              → (POP P VAL)
                (JUMPE VAL :LBL),

        (JUMP :LBL)
              → (JUMPA VAL :LBL),

        (LABEL :LBL)
              → :LBL,

        (FETCH (FUNCTION LESSP))
              → (POP P VAL)
                (CAMG VAL 0 P)
                (SKIPA VAL NIL)
                (MOVEI VAL T)
                (MOVEM VAL 0 P),

        (FETCH (VARIABLE :V))
              → (PUSH P :V) ;
```

The code generated is thus:

```
            ML                          LAP

  (FETCH (VARIABLE A))          (PUSH P A)
  (FETCH (VARIABLE B))          (PUSH P B)
  (FETCH (FUNCTION LESSP))      (POP P VAL)
                                (CAMG VAL 0 P)
                                (SKIPA VAL ZERO)
                                (MOVEI VAL 1)
                                (MOVEM VAL 0 P)
  (DJUMPF E0001)                (POP P VAL)
                                (JUMPE VAL E0001)
  (FETCH (VARIABLE C))          (PUSH P C)
  (JUMP E0002)                  (JUMPA VAL E0002)
  (LABEL E0001)               E0001
  (FETCH (VARIABLE D))          (PUSH P D)
  (LABEL E0002)               E0002          .
```

In the actual compiler, special rules for optimizing conditionals, a peephole optimizer, and additional working registers 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 convenient 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 USP-TO-ML translator "COMPILE", which includes the following rule for the intrinsic function PLUS (slightly simplified for presentation):

```
RULES OF COMPILE =

        (PLUS :X :Y)
              → <COMPILE :X>
                <COMPILE :Y>
                (FETCH (FUNCTION PLUS)) ;
```

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 COMPILE ALSO =

        (PLUS :X 0) → <COMPILE :X>.

        (PLUS 0 :X) → <COMPILE :X> ;
```

The compiler is ordered by SPECIFICITY, which knows that the literal 8 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 and a colon variable at its second occurrence more specific than one at its first occurrence. 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)) ;
```

Additional Facilities

The programmer can specify either deterministic or non-deterministic matching; the former case generates faster code while the latter provides for backtracking. 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.

DEC patterns can be used outside of rewrite rules for decomposition of data structures in MLISP statements.

Applications

It is easy to define a system of inference rules, of assertions, or of beliefs as a rewrite function. From a set of rules can be retrieved either all of the assertions or the first that match a given pattern, A robot planner could be organized into RULES OF PHYSICS, RULES OF INITIAL_CONDITIONS, RULES OF INFERENCE, RULES OF STRATEGY, etc. Note that goal-directed procedure invocation is performed within each of these functions separately. This allows for segmentation of large programs. Furthermore, it averts the need to rummage around a conglomerate data base of unrelated rules.

Rewrite rules are a useful tool for natural language analysis, whether the methods used are based on phrase structure grammar, features, keywords, or word patterns. A use of L1SP70 with the latter method is described in a companion paper.[10] The program described therein utilizes the replacement facility extensively.

Implementation of Rewrite Functions

In the initial implementation of LISP70, rewrite rules are processed in a top-down, left-to-right manner. During the ordering phase, the rules of each extensible function are factored from the left to avoid repetition of identical tests in identical circumstances. The resulting code is a discrimination tree that eliminates many choice-points for backtracking.

The backtracking implementation is an improvement of that developed for MLISP2.[22] It incurs little overhead of either time or space.

The machine code generated for rewrite rules consists primarily of calls on scanning and testing functions. These functions are generic and will process input and output of streams of any type, including lists, character strings, files, and coroutines. For example, streaming[18] intermediate results of compiler passes between coroutines circumvents expensive temporary storage allocation and speeds up the compiler.

Conclusions

Some 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. People who "think in Algol" should not have to cope with S-expressions to write algorithms. Neither should people who "think in patterns". Rewrites, MLISP, and LISP 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 facilities 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,

Preliminary versions of LISP70 have been run on a PDP-10 using a bootstrap compiler, As of June 15, a production version has not been completed. The language has been used successfully in programs for question-answering and planning. Extensions are planned to improve its control structure, editing, and debugging capabilities, and versions will be bootstrapped to other computers.

Acknowledgment

The authors wish to thank Alan Kay for valuable Insights.

REFERENCES

1 Abrahams, P. W. et al, "The LISP 2 Programming Language and System", Proe. 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 Brown, P. J., "Levels of Language for Portable Software", Comm. ACM 15, 12 {Dec. 1972), 1059-1062

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

6 Colby, K. M. and Enea, H, "Heuristic Methods for Computer Understanding of Natural Language in Context Restricted On-Lino Dialogues", Math. Biosciencec 1 (1967), 1-25

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

8 Dickman, B N, "ETC: An Extensible Macro Based Compiler", Proc. AFIPS SJCC 38 {19711,529-538

a Duby, J. J., "Extensible Languages: A Potential User's Point of View", in [21],pp.l37-140

10 Enea, K, Colby, K. M., and Moravec, H., "Idiolectic Language-Analysis for Understanding Doctor-Patient Dialogues", (in this proceedings)

1' Enea, H, and Tesler, L, "INTEGRATE", Unpublished Stanford University Class Project (1964)

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

13 Guzman, A., and Mcintosh, H. V., "CONVERT", Comm. ACM 9, 8 (Aug. 1966), 604-615

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

15 Hewitt, C, "Procedural Embedding of Knowledge in PLANNER", Proc. UCAI 2 (1971), 167-182

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

17 *Landin, P.* J., "The Next 700 Programming Languages", Comm. ACM 9, 3 (March 1966), 157-166

18 Leavenworth, B. M., "Definilion of Quasi-Parallel Control Functions in a High-Levei Language", Proc. Irtt'l. Comp. Symp. (Bonn, 1970)

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

20 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

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

22 Smith, D. and Enea, H, "Backtracking in MLISP2", (in this proceedings)

23 Smith, D, "ML1SP {PDP-10)", Artificial Intelligence Memo No. 135, Stanford University, Oct. 1970

24 Smith, D.C. and Enea, H.J., MLISP2 Manual, Artificial Intelligence Memo No. 195, Stanford University, June 1973

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

26 Teitelman, W. et al, B8N-LISP Reference Manual, (July 1971), Bolt, Beranek, and Newman

27 Teitelman, W., "Toward a Programming Laboratory", Prw. IJCAI 1 (1969), 1-8

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

29 Wegbreit, B.,p "The ECL Programming System", Proe. AFIPS FJCC 39 (1971), 253-262

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