AFCRL-66-302

FORMAT-DIRECTED LIST PROCESSING IN LISP

Daniel G. Bobrow
Warren Teitelman

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

Contract No. AF19(628)-5065
Project No. 8668
Scientific Report No. 3

April, 1966

(The work reported was supported by the Advanced Research Projects Agency under ARPA Order No. 627, Program Code No. 6 D 30.)

Prepared for:

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD, MASSACHUSETTS

AFCRL-66-302

FORMAT-DIRECTED LIST PROCESSING IN LISP

Daniel G. Bobrow
Warren Teitelman

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

Contract No. AF19(628)-5065
Project No. 8668
Scientific Report No. 3

April, 1966

(The work reported was supported by the Advanced Research Projects Agency under ARPA Order No. 627, Program Code No. 6 D 30.)

Prepared for:

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD, MASSACHUSETTS

ABSTRACT

This report describes a notation and a programming language for expressing, from within a LISP system, string transformations such as those performed in COMIT or SNOBOL. A simple transformation (or transformation rule) is specified by providing a pattern which must match the structure to be transformed and a format which specifies how to construct a new structure according to the segmentation specified by the pattern. The patterns and formats are greatly generalized versions of the left-half and right-half rules of COMIT and SNOBOL. For example, elementary patterns and formats can be variable names, results of computations, disjunctive sets, or repeating subpatterns; predicates can be associated with elementary patterns which check relationships among separated elements of the match; it is no longer necessary to restrict the operations to linear strings since elementary patterns can themselves match structures. The FLIP language has been implemented in LISP 1.5 and has been successfully used in such disparate tasks as editing LISP functions and parsing Kleene regular expressions.

SECTION I

INTRODUCTION

The processes involved in symbol manipulation can be expressed in a number of different notations. For example, IPL employs a machine language type of notation in which elementary processes are performed on data structures. It utilizes the equivalent of an accumulator (the push-down accumulator H∅) and a push-down program counter H1. A second type of notation for expressing symbol manipulation is found in LISP, a function-oriented language. In this language, transformations of symbolic structures are achieved by applying functions to lists and using the values of these functions. Functions may be defined using composition, conditional expressions, and recursion. These processes make LISP a very powerful symbol-manipulating programming language; however, the explicit function-oriented nature of LISP makes it difficult to express some operations and transformations which are necessary for the solution of certain types of problems. Basically, these problems require locating certain substructures in a larger structure, either to ascertain their presence, to find their value, or as is more usual, to use them is assembling other structures.

Such transformations may be characterized (and caricaturized) by the following instructions for transformation: find in this string the substring consisting of the three elements immediately preceding the first occurrence of an a, and find the element just before the occurrence of a b which follows these three elements; if such elements exist, exchange the position of the three elements and the one element, delete the a, and replace the b by a c.

The LISP formalism cannot easily express such processes, although each can be individually programmed (if only because LISP is a universal symbol-manipulating language). A notation for expressing such transformations is the basis for a number of programming languages that exist today, such as COMIT, SNOBOL, and AXLE. Each provides a formal method for selecting substrings from a string, and then indicating the structure of the transformed string. These formalisms make it easy to write rules which perform string transformations such as rearrangement, deletion, insertion, and selection of elements from contents. However, it is cumbersome in these languages to express some of the operations which are expressed quite easily in LISP. Some of the latter operations depend very strongly on the fact that LISP can have sublists within lists to unlimited depth, whereas COMIT has lists only to depth 3 and SNOBOL and AXLE deal only with linear strings.

An obvious solution to this notational difficulty is to provide both types of language capability, a function-directed list processing and a format-direct list processing notation within the same programming system. This paper is a report of the marriage of these two capabilities within the LISP 1.5 programming system, and a proposal for the type of capability needed within the LISP 2 system. The implementation in LISP 1.5 of FLIP (Format List Processor, done by Teitelman) is based upon, but is a considerable generalization over, programs and writings of Bobrow and MacIntosh, and has certainly been influenced by features of the string-processing languages referenced above.

The notation used in this paper will be that of FLIP, although this notation probably should not be carried over in its entirety to LISP 2. Part of the awkwardness in the current notation is due to the awkward way in which reading and printing occur in the LISP 1.5 system. The important thing to realize here is that utilization of this language involves a translation from the external language to a more efficient form for internal use. Therefore, it will be possible when more sophisticated translators are available (as in LISP 2) to provide whatever notation the user wishes. The critical thing will be the semantic features available to the user.

The features of the string-processing languages mentioned above can be divided into two almost-independent features. The first is the method of specifying a parsing, or segmentation, of a list structure or string according to a pattern; and the reconstruction of a string according to a format, utilizing parts found in a parsing. Each transformation of this type is called a rule. The second feature of the system deals with the flow of control between rules, and its dependence on success or failure of the matching process used to find the parsing. Because FLIP is embedded within LISP, it need not have its own control mechanisms.*

Our principal concern in this paper will be the match operation which yields the desired parsing of an input list, and the construct operation used to form new lists. Because these processes are independent, they should be treated separately; there needn't be just one construction corresponding to a single match or parsing. In FLIP in LISP 1.5, these processes are implemented by two functions, match and construct. The two arguments of match are (1) a list to be parsed, and

---

* In fact, several different useful executive programs have been written in LISP to facilitate using sets of rules, but since these are so easy to change or write anew for each application, we shall discuss them only briefly in the conclusion.

(2) a pattern which specifies the parsing desired. The arguments used by construct are (1) a representation of a parsing found by a match and (2) a format which specifies the desired structure of a list.

SECTION II

THE MATCHING PROCESS

The purpose of the matching process is to determine whether or not an input list is an instance of a particular (input) pattern. If it is, the match process is designed to tell us this and also to yield a parsing of the list with respect to this pattern. This parsing can then be used by the construct process to build new list structures.

The pattern mentioned here is a list of elementary patterns, and each of these must match a portion of the input list, or else the entire pattern will not match the list. Furthermore, these portions, or segments as they will be called, must together, and taken in order, make up the entire input list. This set of segments will then constitute the parsing of the list. As an example, let us consider a pattern composed of the following three elementary patterns:*

$	which matches anything
$n	which matches a segment of length n
x	which matches x, i.e., a segment of length 1 consisting of a single item equal to x.

* This notation is taken from COMIT and is representative of the external notation used in FLIP. A translating function is used to convert the external notation to the internal notation for a pattern used by match.

Suppose the pattern was

($ $3 A $ $1 B $)

and the list to be parsed was

(A W X Y Z A B C D E B C D)

then the parsing would be (using [...] to denote a segment of a list):

[A W] [X Y Z] [A] [B C D] [E] [B] [C D]

where each segment corresponds to one elementary pattern. Note that the A pattern did not match the first A, because the $3 pattern must first find a segment of length 3. The first $ matches the segment up to the beginning of that matched by the $3. Similarly, B does not match with the first B after the second A because there must be at least 1 item between them to satisfy the $1 pattern. Finally, note that if the $ at the end of the pattern were not present, then there would be no match because there is no way for the segments of the match to make up the entire list.*

---

* This is similar to using SNOBOL rules in anchor mode.

In the rule above we used A and B as quoted items, that is they matched items identical to themselves. In order to match with an item named A, we use the notation (=A) which will treat A as a variable. Thus if A is a variable which has as its value the list (X Y Z), then the pattern

($ (=A) $)

matches the list

(A X Y Z (X Y Z) Q)

with parsing

[A X Y Z] [(X Y Z)] [Q]

In the example above we note that the elementary pattern (=A) matched a segment consisting of exactly one item, the list (X Y Z). This was because the value of A was the list (X Y Z).*

It is also possible to treat a variable as a segment. In FLIP this is denoted by using the special symbol "**". Thus

---

* The operator = is essentially an evaluation operator, and as such it allows the user to call any LISP function from within a pattern. The value of the LISP function for the specified arguments is used in the match process.

if the pattern were ($ (** (=A)) $), a match would occur with the list given above and have the parsing:

[A] [X Y Z] [(X Y Z) Q]

Almost all of the elementary patterns we shall discuss can be used to match either a single item in the workspace, or a segment of the workspace. We saw above that we used the special symbol "**" to indicate that a segment was to be matched. Similarly, in FLIP, the special symbol "*" is used to indicate that a single item is to be matched. We really should have used (* (=A)) in the first example above. However, since common usage of variables is to match single items, a default declaration is inserted by the translator, making (=A) equivalent to (* (=A)). In general, the translator allows the common usage of each elementary pattern to be specified by default.

Since one of the advantages of working in LISP is the ability to handle complicated structures, we want to utilize FLIP rules to match nonlinear lists. An elementary pattern which achieves this is the subpattern. A subpattern matches a single item which is a list in the same way that the top-level pattern matches the top-level list. For example, given the list

(A (B C) D (B E F) G)

and a pattern

($ ($ F $) $),

a match will occur and yield the top-level parsing

[A (B C) D] [(B E F)] [G]

Furthermore, the internal representation of [(B E F)] which matches the subpattern ($ F $), reflects the fact that it has been matched by a subpattern, and also includes the parsing with respect to that subpattern

[B E] [F] [ ].

In line with a general philosophy for design of programming languages which states that any place where a constant may appear it is desirable to allow a variable to appear, in FLIP we have the ability to have a variable whose value is a pattern. If A is such a variable, then ($* (=A)) is an elementary pattern which will match a sublist which matches with the pattern which is the value of A. Thus if the value of the variable A were ($ F $), then the pattern ($ ($* (=A)) $) is equivalent to the pattern shown above, ($ ($ F $) $), and will yield corresponding identical parsings. However, the difference is that one can change the value of A, and obtain different parsings with the same pattern.*

* A can of course be replaced by an arbitrary LISP computation.

Suppose a user wishes to use a variable pattern to match a segment of the workspace, rather than a single item. By utilizing the special symbol "$**", the user may specify that a variable should be treated as a pattern, and be used to match a segment of the workspace. Thus if the value of A were ($1 E) the pattern ($ ($** (=A)) $) would match the workspace (A (B E) D E F) with parsing [A (B E)] [D E] [F], and the internal representation of the segment [D E] would indicate that it was matched by a subpattern and would include the parsing [D] [E].

Another string processing language, AXLE, uses an assertion table as well as rules, and thus attains the ability of defining a variable having one of several alternative values. The same effect can be achieved directly in FLIP by using an elementary pattern (EITHER (E1) (E2) ... (En)) which will match a segment containing a single item, which is either E1 or E2 or ... or En. We can use the EITHER elementary pattern directly in a pattern, or we can define a variable to be equal to such a subpattern. One useful example of this would be to define the variables

digit = ((EITHER (1) (2) (3) (4) (5) (6) (7) (8) (9))) and

integer = ((EITHER (=digit) (($** (=digit)) ($** (=integer)))

With these two variables defined, it is then possible to use

($** (=integer))

in the workspace to find a segment of the workspace which matches the Backus normal form definition of integer given in the above two rules.

Another very useful ep, or elementary pattern, is one indicated by REPEAT. This ep allows one to match a pattern which appears repetitively in the workspace. The elementary pattern (REPEAT E1) will match zero or more occurrences of the elementary pattern E1 in a workspace.* Two simple examples for utilizing this REPEAT are ((REPEAT $2)) which will match the workspace if and only if there are an even number of items in the workspace and ((REPEAT 1(=digit))) which will match a string of one or more digits in the workspace, giving an alternative way of recognizing integers as defined above.

---

* (REPEAT E1 E2 ... En) matches zero or more occurrences of a sequence of segments matching E1 E2 ... En, e.g., (REPEAT A $1 B) matches (A X B A Y B A Z B). In addition, if a number n is inserted immediately after the REPEAT, the subpattern must match n or more times.

The EITHER and REPEAT subpatterns are similar to the operations of disjunction, "v", and * in the formation of regular expressions. In fact, given the definition of a regular expression, it is very easy to write the FLIP matching pattern which will parse a string if, and only if, it is an example of that regular expression. For example, the pattern

(REPEAT (EITHER (A B) ((REPEAT (EITHER (B C) (D E F)))))

will be equivalent to the regular expression

(A B v (B C v D E F) *) *

and will match with

A B D E F B C B C D E F A B.

The reader may have noted that the definition of the $ elementary pattern is not complete. For example, suppose the workspace is (A B C D C D E) and the pattern is ($ C $). According to the definition, both [A B] [C] [D C D E] and [A B C D] [C] [D E] are admissible parsings. This ambiguity will not arise in practice because the operation of the match is a sequential, left to right process. Thus the first match will be the one that is found.

Normally, one does not consider the match process in these terms, i.e., as a series of distinct operations. One of the

niceties of a format processing language is precisely that the user can specify a search procedure in terms of the structures being searched for, and not be concerned about the details of the search. This makes the language more or less goal-oriented. However, for some purposes, it is desirable to think of the matching process (and constructing process) as proceeding from left to right with each elementary pattern performing a certain operation on the partial match, work-space, etc.

This latter conceptualization is not without its rewards. Since the match does proceed from left to right, at the time any particular elementary pattern is operating, all the elementary patterns to the left of it must have already matched with acceptable segments (or this elementary pattern would never have been reached), and these can therefore be referenced. This referencing can be done in two ways, one similar to the way it is done in COMIT and the other similar to the SNOBOL notation. Each elementary pattern is assigned a mark (number) corresponding to its position in the total pattern. Thus if the pattern were ($ $2 A $), the $2 would be referenced by the mark 2 and the A would be referenced by 3. These marks or numbers, refer back to previously matched segments in the workspace. An item can be named by embedding

it in a list of the form (NAME <name> <ep>). For example, the pattern ($ (NAME FOO $2) would give the same matches as the pattern above, but will assign a value to the variable FOO corresponding to the segment matched by the $2.

Given a pattern ($ $2 $ 2 $), this will match a workspace which has a segment of two items repeated in the workspace. For example, it will match the workspace (A B C D E B C D) with the parsing [A] [B C] [D E] [B C] [D] where the $2 in the pattern matched [B C] and the mark 2 matched the second occurrence of this segment in the workspace.* One can use marks to reference single items by utilizing the special symbol "*". For example, the pattern ($ $2 $ (* 2) $) will match a workspace (A B C (A B) D) with a parsing [ ] [A B] [C] [(A B)] [D].

These temporary names, or marks, which refer to pieces of a parsing, will be saved in the parsing of a particular workspace with respect to a particular pattern. However, one may want to use part of the workspace in setting up the values of variables. This can be done as shown earlier or by using

---

* One can also use marks as inputs to LISP computations, e.g., ($ $2 $ (=CAR 2) $) will match (A B C D E B G) and yield [A] [B C] [D E] [B] [G].

an elementary pattern within the match which will actually give a variable a value corresponding to the value of one of the marks (or any LISP computation). The notation used in FLIP is ($SET VAR 2), for example, which will give as a permanent value for VAR the value of the mark 2 at the time this VAR is set up (even if the rest of the match fails).

In addition to being able to utilize marks to describe matched segments of the workspace, there are marks, more complex in notation, which describe segments of subpatterns which have been matched within a pattern. We will not discuss that notation here, but only mention that it is important to be able to pick out any segment parsed in any subpattern of a pattern, including subpatterns in an EITHER or REPEAT elementary pattern. One of the philosophies of design in FLIP was that all information obtained is saved and made available to the user in some way. The details of the current implementation are found in the reference cited.

Considering the pattern as a search from left to right, one may at some point in the matching process, wish to ask questions about segments that have been already matched. That is to say, one may wish to ask questions about the relationships between the current segment and previously matched segments which are more complex than straightforward matching criteria.

We allow this to be done in FLIP by allowing LISP predicates to be attached to patterns, and allowing these predicates to have as arguments the previously matched segments. Thus, for example, the pattern

```
( $3 $3 / (EQUAL (=REVERSE 1)) )
```

will match with any workspace containing six elements where the first three elements are the same as the last three elements in automatically reverse order. The first argument of this predicate is the segment matched by the second $3. Then with a workspace (A B C C B A) this pattern would give the parsing [A B C] [C B A].

As an example of how one might use a mark within a pattern, one can use a pattern which looks like

```
(QUOTIENT $1 (TIMES $ 2 $) )
```

for determining whether or not a quotient (in LISP formalism) has a common factor in it. The 2 refers back to the segment matched by the $1, the second elementary pattern in the pattern.

The elementary pattern which essentially runs the left to right search is the $ pattern. It is this pattern which changes over and over again every time there is a failure in patterns to the right of it in attempting to match. The $

pattern matches the smallest segment it can, namely the null segment first, and each time is increased by 1 item until th patterns to the right of it match. However, consider the case where we have the pattern ($1 $ 1 $5 $). This pattern will match any workspace in which the first item is repeated at least five items before the end of the workspace. This left to right search is run essentially by the second elemer pattern, the $, which keeps increasing the size of the segme it matches and then allows the 1 and $5 to attempt to match. However, what happens once the $5 runs out of things to matc and there is no longer any possibility of obtaining a match? Even so, this second elementary pattern, the $, continues to increase in length trying to let the further rightmost elementary patterns obtain a match; this is now impossible, and there is a way of transmitting this information back. B attaching a special failure predicate to the $, it can test to see whether it should continue searching or not. That is because all the information on what happened further on in the match is retained, the $ pattern can test to see whether it should continue expanding and attempting a match. We wil not discuss the notation used for this in FLIP, but this ver powerful feature makes for much more efficient FLIP programs

Because pattern matching is essentially a left to right sear there are certain things which are very awkward to express.

For example, suppose we wish to locate the last A that appears any place before the first B. The pattern ($ A $ B $) will not work because the A pattern will match the first A it comes to in a workspace, and will not skip over all A's to make the second $ as small as possible.* In order to facilitate this search, which is essentially a right to left search, we have incorporated into FLIP an elementary pattern which will do a reverse search. A pattern which would match the workspace and find the last A before a B would be ($ B ($R A) $ B $), and would match the workspace (Q A R A L M B D) with the correct parsing.

Each elementary pattern used in a pattern has been implemented by utilizing a separate LISP function which is called by the FLIP executive program, and given the workspace and partial match as arguments. Because each pattern is separate, the system is modular and new elementary patterns can be added very easily. The only things that must be done are:

1. tell the translator how to translate a particular form found in a pattern, that is, into what function the form should be translated, and

---

* The pattern ($ A $ / (NOT CONTAIN A) B $) will work.

2. program the new function so that it interfaces properly with the other functions in FLIP.

If, on the other hand, a new function for searching is needed only infrequently, a new elementary pattern can be inserted into a pattern directly by giving its function definition right there. Thus any LISP function can be called from within the match, and can be used in aiding the match.

Occasionally, one would like to locate a particular structure in a list disregarding its depth. For example, working with a list

(A B (C D E) ((F G (M (N O)) P)) I J)

to refer to the (N O) one must use a pattern

($ (($ ($ (N O)) $)) $)

to refer to the S-expression at the correct depth. The internal representation of LISP necessitates this sort of specification, since parentheses are not characters but structural symbols. However, if one considers this list as a linear string of characters, locating (N O) would be trivial. Therefore, we have defined a function, FLATTEN, which transforms a list into a linear string of characters (or atoms) substituting special atoms L* and R* for left and

right parentheses, respectively. This function is isomorphic in a sense to the function PRINT, which takes a list structure and converts it into a linear string, substituting the print names for atoms, and using the characters "(" and ")" to indicate depth. Thus if the list above was called X, then (FLATTEN X) would be

(L* A B L* C D E R* L* L* F G L* M L* N O R* R* P R* R* I J R*).

Now to locate (N O) we need merely write ($ L* N O R* $). We can then perform the necessary transformations and reconstruct the final structure using the function UNFLATTEN.

In utilizing FLATTEN to work with list structures in flattened form, it turns out that a very useful elementary pattern is one that locates a structure containing a specified substructure. Thus where X was given as defined earlier, it would be nice to locate the structure (M (N O)) in the flattened structure by specifying that it contains M, or (N O), or even just N. This latter is similar to asking FLIP to locate N and then back up until it finds two pairs of balanced parentheses. Such an elementary pattern, denoted by UPN, was added, in just the way specified earlier, after the rest of the structure of FLIP had been defined. The pattern used would be ($ N UP2 $), or equivalently ($ M UP1 $). This is

another example of a case in which one wishes to do a right to left search at some point in the left to right search which governs the match.

SECTION III

THE CONSTRUCT PROCESS

The purpose of the construct process is to construct a new list structure using a format and the parsing from a match. Since the flavor of the construct is very similar to that of the match, and, in fact, it uses many of the same functions as the match does, we will not go into it in nearly as great detail as we have the match.

The inputs to the construct process are a representation of the parsing found by match, and a format. The format is a list of elementary formats, which are evaluated sequentially from left to right, their values being attached to the list structure under construction. For example, suppose we wish to locate the item just before an A, and wish to reverse the order of these two items while replacing the A by a B. Then as a pattern for the matching process we would use ($ $1 A $) and as a format for construct process we would use (1 B 2 4). Here the numbers refer to the segments matched by the first, second, and fourth marks in the pattern, and B is a quoted item which stands for itself. There are elementary formats analagous to each of the elementary patterns for matching. For example, one may use (= VAR) to compute the value of a variable and to insert it as an element in the construct. Instead of simply a variable, one can indeed evaluate a LISP function and insert the results of this LISP

computation into the new list being constructed. In addition, there is the same flexibility (and notation) for specifying whether the result of a computation, and the value of any elementary format, should be placed in the new list being constructed as an item or as a segment. There are elementary formats which allow one to reference elements of sub-patterns in a work space, and elementary formats which are themselves formats, so that one can construct new list structures in a very general way. There are also elementary formats which allow one to perform construction in a way parallel to the parsing done by EITHER, and choose which construction element, or elementary format, you wish to use, depending on which of the elementary patterns was the one matched in EITHER. There is also an elementary format which allows repeated constructions in the same way that the REPEAT elementary pattern allows one to look for a repeated pattern in a work space.

SECTION IV

CONTROL PROGRAMS

A number of LISP functions have been written which control flow in a set of FLIP rules. Some of these do the following:

1. Repeat use of each rule until it fails, and then go on to the next.

2. Every time a rule is successful go back to the top of the set of rules. On failure go to the next rule.

3 On a successful match, control goes to a specified labelled rule (similar to COMIT).

LISP 2 is an algol-based language with (labelled) statements. Format directed list processing will be done by a special FLIP statement which will provide a convenient way to call match and construct implicitly. The general form in source language for this FLIP statement will be

FLIP W, U←<pattern>, A←<format>...., D←<format>, S(s_1) F(s_2)

W is a locative expression for a list which will be the work space for the pattern match. U is the name of a symbolic array in which the representation of the parsing will be stored if the

match is successful. The "←" is used in LISP 2 to indicate assignment of value (as the = is in FORTRAN). An implicit call is made to match with the <pattern> and W as arguments. A, B, ... D are locative expressions which are assigned values (if the match is successful) of a list constructed according to their respective <format>'s.

If the match is successful, after assignments to A, ..., D control goes to the statement labelled s_1 (indicated by $S(s_1)$). If the match fails, a transfer is made to s_2 (indicated by $F(s_2)$).

Many elements of this statement are optional. If "U←" is omitted, a local array is created, but not named. If "A←" (the first locative expression with a format) is omitted, it is assumed to be W. No other formats need appear. $S(s_1)$ and/or $F(s_2)$ may be omitted; transfer will be made to the next statement for an omitted condition label. If (s_3) appears, unconditional transfer is made to the statement labelled s_3.

SECTION V

CONCLUSION

FLIP seems to be a very powerful addition to LISP 1.5 as a tool for symbolic manipulation. The combination of the function oriented list processing of LISP and the format directed list processing of FLIP allows easy expression of a wider range of processes then either individually. The language of FLIP seems to be as powerful for expressing string manipulations as COMIT, SNOBOL, or AXLE, if we assume an interpretation of the match as a search through a string variable of LISP 2. Finally, the fact that FLIP is itself written in LISP makes it very flexible and easy to change, and does not seem to sacrifice much in the way of efficiency.

REFERENCES

1. Berkeley, E. C. and Bobrow, D. G. (eds.) The Programming Language LISP: Its Operation and Applications, Information International Inc., Cambridge, Mass., 1964.
2. Bobrow, D. G., "METEOR: A LISP Interpreter for String Transformations," in (1).
3. Bobrow, D. G., "The COMIT Feature in LISP II" Project MAC Memo M-2919, MIT, Cambridge, Mass., February, 1964.
4. Cohen, K. and Wegstein, J. H., "AXLE: An Axiomatic Language for String Transformations," CACM 8, 11, Nov. 1965 (657-661).
5. Farber, D. J., Griswold, R. E., and Polansky, I. P., "SNOBOL, A String Manipulation Language," J. ACM II, 1 (1964) (21-30).
6. McCarthy, J. et al, LISP 1.5 Programmers Manual, MIT Press, Cambridge, Mass., 1963.
7. McIntosh, H. V., "LISP Conversion," Program Note 3, Centro Nacional de Calculo Del I. P. N., Mexico City, Mexico, February 1965.
8. Teitelman, W., "FLIP - A Format List Processor" MAC Memo M-263, MIT, Cambridge, Mass., September 1965.
9. Yngve, V., COMIT Programmers Reference Manual, MIT Press, Cambridge, Mass., 1961.

Unclassified

Security Classification

DOCUMENT CONTROL DATA · R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY *(Corporate author)*
Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b GROUP

3 REPORT TITLE
FORMAT-DIRECTED LIST PROCESSING IN LISP

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Scientific Report #3 - Interim

5. AUTHOR(S) *(Last name, first name, initial)*
Bobrow, Daniel G.
Teitelman, Warren

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
April, 1966	28	9

8a. CONTRACT OR GRANT NO.
Contract No. AF19(628)-5065

b. PROJECT NO. 8668

c. DOD element 6154501R

d. DOD subelement none

9a ORIGINATOR'S REPORT NUMBER(S)
BBN Report #1366

9b OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
AFCRL-66-302

10. AVAILABILITY/LIMITATION NOTICES
Distribution of this document is unlimited

11. SUPPLEMENTARY NOTES The work reported was supported by the ARPA under ARPA Order 627 Program. Code No. 6 D 30

12. SPONSORING MILITARY ACTIVITY
Air Force Cambridge Research Labs.
Office of Aerospace Research (CRB)
USAF, Bedford, Massachusetts

13. ABSTRACT

This report describes a notation and a programming language for expressing, from within a LISP system, string transformations such as those performed in COMIT or SNOBOL. A simple transformation (or transformation rule) is specified by providing a pattern which must match the structure to be transformed and a format which specifies how to construct a new structure according to the segmentation specified by the pattern. The patterns and formats are greatly generalized versions of the left-half and right-half rules of COMIT and SNOBOL. For example, elementary patterns and formats can be variable names, results of computations, disjunctive sets, or repeating subpatterns; predicates can be associated with elementary patterns which check relationships among separated elements of the match; it is no longer necessary to restrict the operations to linear strings since elementary patterns can themselves match structures. The FLIP language has been implemented in LISP 1.5 and has been successfully used in such disparate tasks as editing LISP functions and parsing Kleene regular expressions.

DD FORM 1 JAN 64 1473

Unclassified

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
List Processing String Manipulation LISP Pattern Matching Markow Algorithms						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through _____."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through _____."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through _____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS)*, *(S)*, *(C)*, or *(U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, roles, and weights is optional.