AFCRL-67-0514

DESIGN AND IMPLEMENTATION OF FLIP,
A LISP FORMAT DIRECTED LIST PROCESSOR

Warren Teitelman

Bolt Beranek and Newman Inc
50 Moulton Street
Cambridge, Massachusetts

Contract No. AF19(628)-5065
Project No. 8668
Scientific Report No. 10

This research was sponsored by the Advanced Research Projects
Agency under ARPA Order No. 627, Amendment No. 2

15 July 1967

Contract Monitor: Stanley R. Petrick
Data Sciences Laboratory

Prepared for:

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

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Concluded)

DESIGN AND IMPLEMENTATION OF FLIP, A LISP FORMAT DIRECTED LIST PROCESSOR

SECTION I

INTRODUCTION

BACKGROUND AND MOTIVATION

LISP [1] is a function oriented 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, conditionals, recursion, etc., all of which makes LISP a very powerful symbol-manipulating language. However, this explicit function oriented nature of LISP sometimes makes it difficult to express operations and transformations necessary for the solution of certain types of problems. Basically, these are operations which require locating certain substructures in a larger structure, either to ascertain their presence, or as is more usual, to use them in assembling other structures.

Consider the transformation given by the following instructions: find in a list the first three atoms immediately preceding the first occurrence of the atom A, and find the atom just after the first occurrence of the atom B which follows these three atoms; <u>if</u> such elements exist, exchange the position of the three atoms and the one atom, delete the A and replace the B by C.

The LISP formalism cannot easily express a transformation of this type, although such transformations can be individually programmed. However, for applications that require many such transformations, this can be tedious and time consuming for the programmer.

A notation for expressing such transformations is the basis for a number of programming languages that exist today, such as COMIT, SNOBOL, AXLE, and METEOR [2] which was an earlier embedding of such a feature in LISP.[5] Each of these notations provides a formal method for selecting substrings from a string, and then indicating the structure of the transformed string. For example, the above transformation written in COMIT is:

```
    $+$3+A+$+B+$1+$ = 1+6+4+C+2+7
```

This is much easier to write and understand than the corresponding LISP code for this transformation. However, in COMIT and similar languages, it is cumbersome to express some of the operations which are expressed quite easily is LISP, especially those which depend strongly on the fact that lists can contain sublists to unlimited depth. An obvious solution to this difficulty is to provide both types of language capability within the same programming system.[5] For example in LISP, a programmer might use

```
   (FLIP (REVERSE W)  '($ $3 'A $'B $1 $)  '(#1 #6 #4 'C #2 #7))
```

to transform the reversal of the list W according to the above rule.

The philosophy behind such an extension in the syntax of LISP is similar to the motivation for allowing ALGOL type statements to augment the older LISP 1.5 notation. We do not extend the semantics of the language, but rather provide a capability which vastly simplifies the construction of certain types of programs.[5]

Some Preliminary Considerations

Let us consider an example of a transformation that is suitable for FLIP and arises from actual usage: the expansion of a FOR-statement for LISP.

The FOR-statement of ALGOL allows the programmer to specify a considerable range of iterative operations in a compact form. It would be a powerful syntactic extension to LISP. However, it would not represent a semantic extension, because the corresponding operations could be programmed directly using PROG statements with appropriate control loops. For example, if a programmer wished to form the sum of all positive numbers in a list L, he might write using the FOR-statement:

```
     (FOR X (IN L) (UNLESS (LESSP X 0)) (SETQ SUM (PLUS SUM)))
```

or he could write

```
     (PROG (Y) (SETQ Y L)
        LOOP (COND ((NULL Y) (RETURN NIL))
                   ((LESSP (CAR Y) 0) NIL)
                   (T (SETQ SUM (PLUS SUM (CAR Y)))))
             (SETQ Y (CDR Y))
             (GO LOOP) )
```

which would perform the same operation.

FOR would be implemented as a function for a LISP interpreter. However, when compiling a function containing a FOR-statement, the function will run much more efficiently if the FOR-statement is compiled "open", i.e. is transformed into an equivalent PROG which is then compiled.

There are two observations to be made concerning the FOR statement. First, it illustrates a practical use for a facility sucn as FLIP. Expanding the FOR statement involves determining which of several alternative patterns the statement matches, and constructing the appropriate PROG. In other words, it involves a transformation similar to, although more complicated than, the ones we have been discussing. If it were possible to express such transformations in straightforward way, perhaps by one rule, and - here is the second observation - if this did not cause (significant) degradation in the performance of the compiler, from the standpoint of running time or space, then a considerable amount of programming effort would be saved in the construction of the LISP compiler itself. However, such a facility would be of little or no use at all if it involved a high overhead. The thing that makes the FOR statement useful - and <u>used</u> by programmers is the fact that it does <u>not</u> cost anything, <u>and</u> it simplifies programming, with the first consideration outweighing the second. If the FOR statement were implemented in a different, less efficient way, for example interpretively via a call to a function FOR at run time, then although this feature would <u>still</u> simplify programming, most experienced programmers would prefer to write their own iterative loops because of the greater efficiency.

It has been my goal in developing FLIP to produce a facility that not only would be useful, but one which would be used. This has entailed developing a compact, yet fairly powerful notation for describing transformations, and a very efficient implementation of these transformations. One of the central considerations has been that the user should not have to pay for options which he does not use. In other words, simple transformations must have a simple notational representation, and run fast. Esoteric options which slow down the operation of the pattern matching because of the <u>possibility</u> of their being used are not desirable for our applications. With respect to efficiency, FLIP includes features which allow the programmer to exercise some degree of control over the manner in which the matching portion of the operation is carried out. However, even where this control is not exercised, a considerable amount of built in optimization insures the programmer of an efficient operation.

SECTION II

THE FLIP FORMALISM

A transformation in FLIP consists of two independent processes. The first is a parsing, or segmentation of the input structure according to some pattern. This is called the matching process. The second is a construction of a structure utilizing this parsing and some format. This is called the constructing process. A transformation is usually specified by a single pattern and format. The value of the transformation is NIL if the input list does not match the pattern, otherwise it is the result of the construction. It is possible to use the matching process as a pure predicate to test the form of an input. In that case a format is not required. One can also perform several constructions using a single parsing.

In the discussion that follows, the matching and constructing operations are treated separately because of this independence.

A. Control Mechanisms

Since FLIP is embedded within LISP, it does not have its own control mechanism. In COMIT, SNOBOL, etc., this is the section of the language devoted to the flow of control between the transformations or rules, and its dependence on the success or failure of the matching process used to find the parsing. Several different useful executive programs have been written in LISP to facilitate using sets of rules, for example,

1. Repeat use of each rule until it (the match) 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. After a match, control goes to a specified labelled rule. (This is very similar to the COMIT control mechanism.)

One control program, TRANSFORM, is described in Section IV. Others are easy to write since the user can call the matching and constructing functions directly.

B. The Matching Process

The purpose of the matching process is to determine whether or not the input list is an instance of a particular input pattern. If it is, the matching 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 input pattern is a list of elementary patterns. Each of these must match a portion of the input list, or else the entire pattern will not match the list. Furthermore, there must be no gaps in the list, i.e. these portions or segments as they will be called, must together, and taken in order, make up the entire 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	where n is a number, 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 (the value of) x.

For the pattern

```
($ $3 'A $ $1 'B $)
```

and the list

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

the parsing would be:

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

where each segment corresponds to one elementary pattern. Note that 'A 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.

Elementary Patterns

$, $n, and a variable are prototypes of three of the elementary patterns available in FLIP. Each of these patterns, as well as the ones we will encounter below, can be embellished considerably with various options. For tutorial purposes, we have chosen first to present each elementary pattern in its simplest form, as was done above, and then to introduce gradually the extensions and generalizations that are permitted. However, a complete summary of both the FLIP syntax and semantics may be found in the appendices.

The Variable Pattern

The variable pattern, or var for short, is so named because in its simplest form it matches with the value of some variable. Actually, var matches with the value of any LISP computation which includes variables as well as other more complicated expressions. In the above example, this computation was 'A, which is short for (QUOTE A). The value of this computation is simply A, and so this elementary pattern matched with A. If the value of X were A, then the pattern ($ $3 X $ 'B $1 $) would match identically with the one given above.

If X is a variable which has as its value the list (3 4 5), then the pattern ($ X $) matches the list (1 2 3 4 5 6 (1) (2 3) (3 4 5) (6)) with parsing

    [1 2 3 4 5 6 (1) (2 3)] [(3 4 5)] [(6)]

Suppose we wanted X to match with the segment [3 4 5] rather than the item (3 4 5), which is a sublist of the original list. We indicate this by using the prefix operator "*" and write ($ *X $).

The parsing would then be

    [1 2] [3 4 5] [6 (1) (2) (3 4 5) (6)]

Just as ($ X $) is identical to, in this case, ($ '(3 4 5) $), ($ *X $) is identical with ($ *'(3 4 5) $). This latter pattern also will match the same list as the pattern ($ 3 4 5 $). However, ($ 3 4 5 $) produces the slightly different parsing:

    [1 2] [3] [4] [5] [6 (1) (2) (3 4 5) (6)].

because it contains 5 elementary patterns instead of 3.

Variables and quoted expressions are two types of LISP computations. To indicate that a match is to take place with some other LISP computation, the prefix operator "=" is used.* For example, one can write

```
    ($ =(CAR (GET (QUOTE NAME) (QUOTE PROPERTY))) $)
```

or

```
    ($ =(PROG (X Y) ... (RETURN X))  $),
```

etc. In general, the elementary pattern =X matches a single element equal to the value of X, which is computed during the course of the match. To indicate that a match occurred with a segment of a list, we use the prefix operator "*", as before, and write *=X, where X is a LISP computation.

To refer back to items or segments already matched in the parsing, a special type of var called a mark is provided. For example, the pattern

```
    ($ $1 $ #2 $)
```

will match a list with two identical elements. For the list (A B C D E F G B X), the parsing would be

```
    [A] [B] [C D E F G] [B] [X]
```

In this example, #2 is a mark; it refers to the second elementary pattern, namely $1, and it matches with the identical item or segment that the $1 elementary pattern matched. For the list

* Actually, the "=" operator may be used for all LISP computations including variables and quoted expressions. However, since these two types occur so frequently, special allowance is made for them and the "=" operator can be omitted.

(A B C (B C) D E (B C) B C F) and the pattern ($ X $ #2 $), where X has the value (B C), the parsing is:

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

A mark always matches identically with the elementary pattern to which it refers. If for the same input list and value of X as above, we use the pattern ($ *X $ #2 $), then the parsing would be:

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

In this case, the mark matched with a segment [B C] instead of an item, [(B C)].

A mark can also be used in a computation. In this case it has the value of the segment matched by the elementary pattern to which it refers, or in the case that this pattern matches a single item, its value is that item. For example, we can write ($ $3 $ =(CADR #2) $) which matches with (A B C D E C G) producing the parsing [A] [B C D] [E] [C] [G]. The pattern ($3 *=(REVERSE #1)) will match with the list (A B C C B A) producing the parsing [A B C] [C B A]. Note that ($3 =(REVERSE #1)), will <u>not</u> match with (A B C C B A); it will match with (A B C (C B A)).

Sometimes for long patterns such as ($ 'A $ 'B $ $1 $ #6 $), it is easier to read and write the patterns if we allow the mark to count backwards from its position, writing ($ 'A $ 'B $1 $ #-2 $). Both of these patterns will "find" the first common elements following the first B that follows the first A.

We can also write #X to denote the Xth element of the parsing, where X is any LISP form that evaluates to a number. Similarly we can write $X to denote a segment of length X. Here it is important in order for the $ to be recognized as a prefix operator that there be no space between it and the X. For example, ($ X $) is not the same as ($X $), nor is ($ (CAR X) $) the same as ($(CAR X) $).

Summary

The three elementary patterns discussed so far are:

$	which matches anything
$X	where X is a computation whose value is a nonnegative number N; matches a segment of length N
=X	matches a single element equal to the value of X
X	where X is atomic; same as =X
'X	same as =(QUOTE X)
*=X	matches a segment equal to the value of X
*X	where X is atomic; same as *=X
*'X	same as *=(QUOTE X)
#X	where X is a computation whose value is a number N; matches with the same thing matched by the Nth elementary pattern. If N is positive, numbering proceeds from the front of the pattern, left to right. If negative, numbering proceeds from position where #X appears, right to left. (Note: |N|<position #X)

Predicates

Suppose we wanted to parse a list finding the last A before the first B. The pattern ($ 'A $ 'B $) is not sufficient, because with the list (A Z A Y A B C), for example, there are three possible parsings,

```
[] [A] [Z A Y A] [B] [C];
[A Z] [A] [Y A] [B] [C]; and
[A Z A Y] [A] [] [B] [C]
```

corresponding to the three different occurrences of A. While the last parsing is the one desired, FLIP and most other pattern driven languages would produce the first parsing, simply because it is the first one found.

One way to produce the last parsing is to restrict the segment that the first $ matches by requiring that it not contain an A. This is done in FLIP by means of a LISP predicate.

We write for the above pattern

```
($ 'A $[NOT (MEMBER 'A *)] 'B $)
```

By definition, $[X] matches anything for which the value of (X) is T.

To reference in X the segment currently matched by the $, you can use the variable "*", as in the above example.

Predicates can also be used with the $N pattern and the VAR pattern. Consider

```
($ $1[MEMBER * '(A E I O U)] $ $1[NOT (MEMBER * #1)] $)
```

Given the list (X Y Z I X M N), the parsing produced is [X Y Z] [I] [X] [M] [N].

Let us consider another example using predicates. We can write $X to match a segment of length X - suppose we wish to match a segment with a length between two bounds? Here we must use a predicate.

```
    ($ 'A $[AND (LESSP(LENGTH *) 5) (GREATERP (LENGTH *) 1)] 'B $)
```

matches with (A B C D E A X B B)giving [A B C D E] [A] [X B] [B] [].

Subpatterns

Predicates provide a means for calling the matching procedure recursively. For example, we could require that a list match with a given pattern by using the elementary pattern: $1[MATCH * PATTERN]. However, a more direct way to achieve this is by means of a subpattern. A subpattern is an elementary pattern that matches 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 the pattern

```
    ($ ($ 'F $) $),
```

a match will occur with the subpattern ($ 'F $) matching the list (B E F). One of the advantages of using the subpattern over a predicate is that the parsing produced by the recursive call to match is saved in the top level parsing. Thus the complete parsing for the match above would be

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

This sub-parsing can then be referred into by other elementary patterns. For example,

```
    ($ ($ 'B $1) $ #[2,3] $)
```

matches with

```
    ((A B C) (A B D) (A B E)  A E I O U)
```

producing the top level parsing

```
     [(A B C) (A B D)]  [(A B E)] [A] [E] [I O U].
```

Here #[2,3] refers to the third element in the second element in the parsing. When this elementary pattern is encountered, the second element in the parsing is found and, treating this as a parsing, the third element in this lower parsing is found. A match will then be made with the same item or segment matched by this element.

Since the mark notation using brackets is treated similarly to that without brackets, the numbers 2 and 3 in the example above could have been replaced by arbitrary computations, and similarly negative numbers could have been used. The notation #2 is equivalent to #[2]. The former is merely a convenient abbreviation. Similarly, marks using bracket notation can be employed in arbitrary LISP computations.

It is not necessary to refer into parsings produced by subpatterns; items matched by subpatterns can be referred to by MARKS in the same way as any other matched item. Thus the pattern ($ ($ 'B 'D) $ #2 $) will match with the list

```
     ((A B C) (A B D) (A B E) (X B D) (A B D) (A B C))
```

producing the top level parsing

```
     [(A B C)] [(A B D)] [(A B E) (X B D)] [(A B D)] [(A B C)]
```

Note that #2 did not match with (X B D) even though ($ 'B 'D) could have matched with it originally. #2 matched with (A B D), as did ($ 'B 'D) earlier.

If a MARK is used inside of a subpattern, it is evaluated using that subpattern's parsing. For example, in the pattern ($ ($1 $ #1) $), the #1 refers to the $1, not the first $, and this pattern will match with

```
((A B C) (D E F) (G H G) (I J K))
```

to produce

```
[(A B C) (D E F)] [(G H G)] [(I J K)].
```

If it is necessary to refer to the parsing outside of the current parsing, one uses the full MARK notation, with brackets, and heads this with the special token "↑". For example

```
($ $1 $ ($ #[↑,2] $) $)
```

matches with

```
(A B C (D E F) (G H I) (X Y C)),
```

with the $1 matching C. The "↑" denotes that counting begins with the top level parsing.

As with the case of VARs, a subpattern can be used to match a segment as well as an item. This is also indicated by the prefix operator "*". Thus we can write

```
($ $3 *('A $ 'B) $1 $)
```

as a pattern which matches with the same lists as those matched by the pattern

```
($ $3 'A $ 'B $1 $).
```

However, the first parsing will contain only five elements, since there are only five elementary patterns. The second parsing will of course contain seven elements. Furthermore, if #3 appears in the first pattern (at the top level) it will refer to the entire segment running from A through B, since this is what is matched by the third elementary pattern in that pattern. #3 appearing in the second parsing would refer to the single item A.

Finally, a subpattern can be computed. This is indicated by the prefix operator ":". Thus we have for the subpattern:

:X	matches a single item, a list, that matches, in the sense described above, the value of X treated as a pattern.
X	where X is a list, same as :'X
*:X	matches a segment that matches in the sense described above the value of X treated as a pattern
*X	where X is a list, same as *:'X

EITHER Pattern

The EITHER elementary pattern provides a means for defining a match of one of several alternatives. The general form for the EITHER pattern is

```
EITHER[E1; E2; E3; ...; En]
```

Each Ei is a sequence of elementary patterns. EITHER attempts to find a match with a segment of the list using first E1, and if that fails, then it tries with E2, etc. For example, the pattern

```
     ($ EITHER[A' $1; 'B $2; 'C $3] 'D $)
```

matches with the list

```
     (X Y Z A B C D E F D G)
```

producing the parsing

```
     [X Y Z] [C D E F] [D] [G].
```

The element corresponding to the segment [C D E F] matched by the EITHER also contains the parsing [C] [D E F] corresponding to the two elementary patterns in the third alternative, the one that matched. This is similar to the treatment of subpatterns described earlier.

As an example of the use of an EITHER pattern, consider the following definition of an integer.

```
     digit = EITHER[1;2;3;4;5;6;7;8;9;0]

     integer = EITHER[*:digit;*:digit *:integer]
```

With these two variables defined, we can use the pattern *:integer to determine whether a list matches the Backus normal form definition of integer given in the above two rules. For example, the pattern

```
($ 'A *:integer 'B $)
```

matches with the list

```
(X Y Z A 1 W B A 3 2 6 B C)
```

producing the parsing

```
[X Y Z A 1 W B]  [A]  [3 2 6]  [B]  [C].
```

(Admittedly this is not the most efficient way to find integers.)

If any of the Ei's are empty, then, the EITHER pattern can match with a null segment of the list. Thus the pattern

```
($ 'A EITHER ['B $1; 'C $2;] $)
```

matches with the list (X A Y) producing

```
[X] [A] [] [Y].
```

Here, the third (empty) alternative was used.

If a mark is used <u>inside</u> of an EITHER pattern, it is evaluated using the EITHER pattern's parsing, much the same as with the subpattern. For example, the pattern

```
($ EITHER[$1 'B #1; $1 #1] $)
```

will match with the list

```
(A B C D B D E)
```

using the first alternative. To refer to elements outside of the EITHER pattern's domain from inside of it, use the mark notation with brackets and ↑. Similarly, the EITHER pattern's parsing may be referred into using the mark notation with brackets. For example,

```
    ($ EITHER['B $1; 'C $1 $1] #[2,-1] $)
```

will match with the list

```
    (A B C D E E G),
```

with the second alternative being used.

The REPEAT Pattern

The REPEAT pattern allows one to match with a repetitive pattern. The general form for this elementary pattern is REPEAT[E], where E is a sequence of elementary patterns. REPEAT will match zero or more occurrences of this sequence. Thus the pattern

```
    ('A REPEAT['B $1])
```

will match with the lists

    (A), (A B C), (A B C B D), etc., but not with

    (A B C B C E), although

```
    ('A REPEAT['B $1] $)
```

would match with the latter list. As with the case of the subpattern and EITHER elementary pattern, the parsings obtained by REPEAT as it matches are retained and available during the course of the match and construct operation. Similarly, marks used inside of the REPEAT pattern refer to the parsing of the current repetition. For example, the pattern

```
     (REPEAT[$1 $1 #1]) will match with the list

     (A B A C D C E F E).
```

To refer to elements outside of the REPEAT pattern, marks with brackets and ↑ must be used.

The REPEAT pattern may take two optional arguments, N1 and N2. If N1 is present, the REPEAT pattern must match at least N1 times. If N2 is present, the REPEAT will match at most N2 times. For example, to match a segment containing from 1 to 6 letters (say a representation of a FORTRAN variable) one can use:

```
     REPEAT[$1[LETTER *] / 1 6]
```

where LETTER is a predicate which is true for letters.

The general form for REPEAT is thus:

REPEAT[E / N1 N2]	where the value of N1 and the value of N2 are both numbers, matches a segment of a list which matches repetitively the list of elementary patterns E at least N1 times and not more than N2 times.

REPEAT[E / N1]	same as REPEAT[E / N1 N2] where N2 is effectively infinite.
REPEAT[E]	same as REPEAT[E / 0]

The SET Pattern

The SET elementary pattern might more properly be called a pseudo-pattern, because it does not affect the match. The SET pattern is used to assign a value to a variable during the match. There are two forms for the SET pattern. The first, (SET X Y), where Y is some LISP form, assigns the value of Y to X. The second form is $X \leftarrow Y$, where Y is some elementary pattern. In this case X is set to whatever the elementary pattern matches. Thus the effect is the same as writing in a pattern Y followed by (SET X #-1). Note: Since this elementary pattern does not match and does not affect the parsing, it should be ignored when computing MARKs. Thus ($ (SET X Y) $1 $ #2 $) will match two common elements, the #2 referring to the $1.

Example

We are now ready to try a more complicated example. In LISP applications, one frequently wishes to locate a balanced pair of brackets in a string of tokens. Let us consider a general FLIP pattern which finds the first balanced string following the unique token LABEL, where the bounding tokens are variables; e.g. they could be BEGIN and END, or "[" and "]", etc. Let us refer to these tokens by the variables OPEN and CLOSE. The general idea is to find the first OPEN after LABEL and increment a counter one for each OPEN, and decrement the counter for each CLOSE until the count is zero. The pattern used is:

```
($ LABEL $ OPEN (SET N 1)
   REPEAT[EITHER[OPEN (SET N (ADD1 N));
                 $1[NOT (EQ * CLOSE)];
                 (SET N (SUB1 N)) $1[NOT (ZEROP N)]]]
   $)
```

After the REPEAT pattern has matched, N will be zero, and the segment matched will consist of the balanced string excluding the initial OPEN and final CLOSE. Suppose we wish, however, to bind the variable FOO to this entire balanced string. Then we might write

```
($ LABEL $ FOO←*(OPEN (SET N 1)
    REPEAT[EITHER[OPEN (SET N (ADD1 N));
                  $1[NOT (EQ * CLOSE)];
                  (SET N (SUB1 N)) $1[NOT (ZEROP N)]]] CLOSE
 ) $)
```

After this pattern has been matched, the value of FOO will be the segment from the first OPEN after LABEL to its matching CLOSE. Note the use of the subpattern consisting of four elementary patterns: a VAR, SET, REPEAT and another VAR.

C. The Construct Process

The purpose of the Construct Operation is to construct a new list structure using a format and a parsing from a match. Since the flavor of Construct is similar to that of Match, and Construct uses many of the same LISP functions as Match does, we will discuss it in less detail.

The inputs to Construct are a representation of the parsing found by Match, and a format. This 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 as specified below. For example, to perform the transformation on page 3 we match with ($ $3 'A $ 'B $1 $) and construct with (#1 #6 #4 'C #2 #7).

<u>VARF</u>

VARF is the elementary format that corresponds to the elementary pattern VAR. Its value is computed and attached at the end of the list structure under construction as an item, or, if the prefix "*" is used, as a segment. A MARK is attached as an item if the elementary pattern to which it refers matched as an item, otherwise as a segment. Negative numbers are permissible in MARK's used in the construct process; they refer to elements by a count from the right end of the parsing moving to the left. Thus the format (#1 #-2 #4 'C #2 #-1) is equivalent to the format above.

=X	X is evaluated and attached as an item at the end of the list being constructed, i.e. effect is the same as APPENDing (LIST X) to this structure.
X	where X is an atom, same as =X
'X	same as ='X, or =(QUOTE X)
*=X	X is evaluated and attached as a segment, i.e. APPENDing X.
*X	where X is an atom, same as *=X

*'X	same as *='X or *=(QUOTE X)
#X,#[..]	the elementary pattern referred to is located in the same way as in the match and then the item or segment it matched is attached appropriately to the list being constructed.

Subformats

The subformat corresponds to the subpattern elementary pattern. It is a list of elementary formats which are used to construct a new list in exactly the same way as top level elementary formats are used to construct a list. This sublist is then attached to the list being constructed either as an item or as a segment, as specified. Subformats may be computed; this is indicated by the prefix operator ":". Subformats may be used within subformats.

:X	X is evaluated and treated as a format. It is used to construct a list which is then added to the next higher level list as an item.
X	where X is a list, same as :'X
*:X	X is evaluated and treated as a format. It is used to construct a list which is then added to the next higher level list as a segment.
*X	where X is a list, same as *:'X

EITHERF

EITHERF is the elementary format corresponding to the EITHER pattern. It specifies the selection of an alternative format to use for construction depending on which alternative in the match was used. Its general form is

```
EITHER[E1; E2; E3; ..; En / X]
```

where each Ei is a sequence of elementary formats and X is a computation, usually a MARK, whose value must be a parsing corresponding to some EITHER elementary pattern. The format Ei corresponding to the chosen Ei in the EITHER pattern is used in construction.

Comments

1. Ei may be empty.

2. During the course of the construction using Ei, the current level parsing is that of the EITHER elementary pattern. This means that any MARKs not employing ↑ will be evaluated with respect to the EITHER parsing. Thus, if one matches the list

```
(X Y A C D E F G)  with

($ 'A EITHER['B $1; 'C $1 $1] $)
```

and constructs with

```
(#1 #2 EITHER[#2; #3 / #3] #4)
```

the result is

```
(X Y A E F G).
```

3. If X is not present, then the last EITHER (furthest right)

at the current level parsing is used. Thus, in the above case, "/ #3" could have been omitted. If no EITHER parsing is found, an error occurs.

REPEATF

REPEATF is the elementary format corresponding to the elementary pattern REPEAT. It specifies the iteration of a number of construction operations, the exact number being the number of times the corresponding REPEAT matched. Its general form is

REPEAT[E / X]	where E is a sequence of elementary formats and X is a computation, usually a MARK, which must produce a parsing corresponding to a REPEAT elementary pattern.

Comments

1. X may be omitted. In this case, the last (furthest right) REPEAT of the current level parsing is used. If none is found, an error occurs.

2. During the construction with E, the current level parsing is that of the corresponding parsing in the REPEAT elementary pattern, i.e. for the nth iteration of E, the nth match of the REPEAT pattern. Thus to delete every third element in a list, match with

```
(REPEAT[$2 $1] $),
```

and construct with

```
(REPEAT[#1 / #1] #2)
```

or simply

```
     (REPEAT[#1] #2).
```

3. The value of X may be a number. In this case, the format E is repeated that number of times, and the current level parsing is the same as that when the REPEATF was entered. Thus to convert

```
     (A B C D E) into (A A A B B B C C C D D D E E E),
```

match with

```
     (REPEAT[$1]),
```

construct with

```
     (REPEAT[REPEAT[#1 / 3]]).
```

The first REPEAT corresponds to the REPEATed pattern. The second one is executed 3 times for each time the REPEAT pattern matched, and #1 is the corresponding $1, etc.

4. It is possible to match with

```
     (REPEAT[EITHER['A $1; 'B $1]])
```

and construct with

```
     (REPEAT[EITHER[#2 #2; #2 #2 #2]]).
```

Here the alternative format is selected according to which EITHER matched on the corresponding iteration of the REPEATed format, and the #2 is evaluated against the parsing of EITHER. This transformation will produce

```
(X X Y Y Y Z Z X X Z Z Z)
```

from

```
(A X B Y A Z A X B Z).
```

In addition to these elementary formats, there is an assignment statement available in <u>construct</u> similar to that in <u>match</u>. This may be written either as (SET X Y) or X←elementary format. In the latter case, X is assigned the value of the elementary format. In both cases, the assignment statement does not affect the list being constructed.

<u>Example</u>

We are now in a position to write the transformation for the FOR-statement expansion described earlier. The general form of the FOR-statement we are using is:

(FOR <u>loopvar</u> <u>loopcontrol</u> <u>whilephrase</u> <u>unlessphrase</u> <u>statement</u>)

where

```
     loopcontrol  =  (LOOP X)
                  or (RESET X Y)
                  or (IN X)
                  or (ON X)
                  or (STEP N I)
                  or (STEP N I FN M)

     whilephrase  =  (WHILE X)
                  or empty
```

```
    unlessphrase  =  (UNLESS X)
                     or empty
```

where X and Y are arbitrary forms, N, I, M are forms that evaluate to numbers, and FN is a function.

If loopcontrol is (LOOP X), loopvar is initialized to X. For RESET, it is initialized to X at the start of the loop and reset to Y after each iteration. (ON X) indicates that the loopvar is to be cycled through the list X. It makes the FOR-statement work in a manner similar to the LISP function MAPLIST, setting loopvar to successive tails of the list X. (IN X) sets loopvar to successive elements of the list X. Finally, STEP specifies a numerical loopcontrol. In the first case loopvar is initialized to N and incremented by I after each iteration. No provision is made for termination of the loop. In the second case, the loop terminates when (FN X M) is true. For example, (STEP 1 1 (GREATERP 10) will cause 10 iterations with loopvar beginning at 1 and going to 10.

Regardless of which loopcontrol is used, the WHILE phrase, allows the user to specify a termination condition for the loop, and the UNLESS phrase specifies exceptions for which the statement is not to be executed.

The definition of FOR is given below. Note that the EITHER pattern allows a compact treatment for the various loop control cases. In particular, the two STEP cases and the IN and ON cases can be treated together.

Following the definition are six examples of expansions produced by FOR. The first two examples using LOOP and RESET both return

T for lists of even length and NIL for lists of odd length. The third example using IN prints all atomic elements in a list L, and the fourth example using ON prints all but the last four elements in L. The last two examples, using STEP, compute the sum of all odd integers less than N. The output was produced using a special print program, PRETTYFLIP, described on page 43.

```
 PRETTYFLIP((FOR))

(FOR
  (LAMBDA (X) (FLIP
      X
      '('FOR LOOPVAR←$1 (EITHER['LOOP $1;
                                  'RESET $1 $1;
                                  EITHER['IN;
                                         'ON] $1;
                                  'STEP $1 $1
                                    EITHER[$1 $1;
                                           ]])
        EITHER[('WHILE $1);
               ]
        EITHER[('UNLESS $1);
               ]
        $1)
      '((SET LOOP (GENSYM)) (SET LOOP1 (GENSYM))
        (SET EXIT (GENSYM))
        'PROG
        EITHER[NIL LOOP ('SETQ LOOPVAR #2);
               NIL ('SETQ LOOPVAR #2) LOOP;
               (PROGVAR←=(GENSYM)) ('SETQ PROGVAR #2) LOOP
                 ('COND (('NULL PROGVAR) ('GO EXIT)))
                 ('SETQ LOOPVAR EITHER[('CAR PROGVAR);
                                       PROGVAR]);
               NIL ('SETQ LOOPVAR #2) LOOP
                 EITHER[('COND ((#1 LOOPVAR #2) ('GO EXIT)));
                        ] / #[3,1]]
        EITHER[('COND (('NULL #[1,2]) ('GO EXIT)));
               / #-3]
        EITHER[('COND (#[1,2] ('GO LOOP1)));
               ]
        #-1
        LOOP1
        EITHER[;
               ('SETQ LOOPVAR #3);
               ('SETQ PROGVAR ('CDR PROGVAR));
               ('SETQ LOOPVAR ('PLUS LOOPVAR #3)) / #[3,1]]
        ('GO LOOP)
        EXIT))))
(FOR)
←
```

```
(FOR X (LOOP L)
  (COND
    ((NULL X)
      (RETURN T))
    ((NULL (CDR X))
      (RETURN NIL))
    (T (SETQ X (CDDR X)))))

(PROG NIL
  A0175 (SETQ X L)
        (COND
          ((NULL X)
            (RETURN T))
          ((NULL (CDR X))
            (RETURN NIL))
          (T (SETQ X (CDDR X))))
  A0176 (GO A0175)
  A0177 NIL
  )


(FOR X (RESET L (CDDR X))
  (COND
    ((NULL X)
      (RETURN T))
    ((NULL (CDR X))
      (RETURN NIL))))

(PROG NIL
      (SETQ X L)
  A0200 (COND
          ((NULL X)
            (RETURN T))
          ((NULL (CDR X))
            (RETURN NIL)))
  A0201 (SETQ X (CDDR X))
      (GO A0200)
  A0202 NIL
  )
```

```
(FOR X (IN L)
  (UNLESS (NULL (ATOM X)))
  (PRIN1 X))

(PROG (A0206)
      (SETQ A0206 L)
  A0203 (COND
          ((NULL A0206)
            (GO A0205)))
      (SETQ X (CAR A0206))
      (COND
        ((NULL (ATOM X))
          (GO A0204)))
      (PRIN1 X)
  A0204 (SETQ A0206 (CDR A0206))
      (GO A0203)
  A0205 NIL
  )

 (FOR X (ON L)
   (WHILE (GREATERP (LENGTH X)
       4))
   (PRINT (CAR X)))

 (PROG (A0212)
       (SETQ A0212 L)
   A0207 (COND
           ((NULL A0212)
             (GO A0211)))
       (SETQ X A0212)
       (COND
         ((NULL (GREATERP (LENGTH X)
                4))
           (GO A0211)))
       (PRINT (CAR X))
   A0210 (SETQ A0212 (CDR A0212))
       (GO A0207)
   A0211 NIL
   )
```

```
(FOR X (STEP 1 2)
  (WHILE (LESSP X N))
  (SETQ SUM (PLUS SUM X)))

(PROG NIL
      (SETQ X 1)
  A0213 (COND
        ((NULL (LESSP X N))
          (GO A0215)))
      (SETQ SUM (PLUS SUM X))
  A0214 (SETQ X (PLUS X 2))
      (GO A0213)
  A0215 NIL
  )


(FOR X (STEP 1 2 GREATERP N)
  (SETQ SUM (PLUS SUM X)))

(PROG NIL
      (SETQ X 1)
  A0216 (COND
        ((GREATERP X N)
          (GO A0220)))
      (SETQ SUM (PLUS SUM X))
  A0217 (SETQ X (PLUS X 2))
      (GO A0216)
  A0220 NIL
  )NIL
←
```

SECTION III

IMPLEMENTATION

This section discusses FLIP as a large systems program. It may help users to write more efficient FLIP programs by explaining the way FLIP works, but it contains no new information on the language. It has been included for completeness, and because we feel that much of the experience gained in experimenting with and using FLIP has been in the area of implementation, and may be transferrable to the design and construction of other large LISP systems.

The section is divided into three parts. The first part discusses the technique of translation. The second part discusses techniques for reducing the number of CONSes required. The third part describes the operation of the $ function, which is responsible for most of the search strategy in the matching operation. Thus the first part talks about ways of speeding up MATCH and CONSTRUCT before they are run. The second part talks about ways of speeding up MATCH and CONSTRUCT indirectly by reducing garbage collection time, and the third part talks about ways of speeding up MATCH while it is running, by a more efficient search.

A. Translation

Each of the elementary patterns and formats in FLIP have been implemented by a single LISP function. There is a function called $, and another function called VAR, etc. Although there are a number of options with which each elementary pattern or format can be embellished, for example, VAR can be made to match a segment or an item, and may or may not include a predicate, there is still sufficient similarity in the tasks performed to permit only one function.

However, given any input pattern or format, there is still the problem of determining which functions are to be called, and with what arguments. Since each of the seven elementary patterns discussed earlier come in a variety of forms, a certain amount of computation must be done to decide exactly which elementary patterns, and elementary formats, are represented in a given pattern or format. This is the task of the translators.

The purpose of translation is to do as much of the work of interpreting FLIP entities as possible, before the programs are run, and to do this work only once. This is similar to the philosophy of compilation. However, unlike compilers, the FLIP translators do not produce machine instructions, but a sequence of LISP forms, i.e., LISP functions with arguments. These forms correspond to the individual functions which carry out the operations specified by each elementary pattern or format. The arguments to these functions indicate the options utilized. Since each of these functions are themselves compiled, there is a minimum amount of interpretation at run time.

As an example, consider the pattern

```
    ($ 'A $1[NOT (MEMBER * #1)] *=(REVERSE #1)).
```

The translation of this pattern is

```
     ( ($)
       (VAR (QUOTE A))
       ($N 1 (NOT (MEMBER * (MARK (1)))))
       (VAR (REVERSE (MARK (1))) SEGMENT)  )
```
*

The translation of the first elementary pattern indicates that $ must match with any segment, and no predicates are used. The translation of 'A tells VAR that it is to match an item and that this item must be equal to the value of (QUOTE A). This is evaluated and VAR compares the first (next) element in the list being matched with A. The translation of the next elementary pattern informs the $N function that it must match a segment of length 1, and that the predicate (NOT (MEMBER * (MARK (1)))) must be true for this segment. $N calls the LISP function EVAL on this predicate after first binding the variable "*" to the segment in question. MARK is a LISP function whose value, in this case, is segment matched by the first elementary pattern, i.e., the $. The translation of the fourth elementary pattern indicates to VAR (the same function that handled the second elementary pattern), that it must match with the segment equal to the value of (REVERSE (MARK (1))).

* VAR is really a function of <u>three</u> arguments: the form to be matched, a segment-item flag, and a predicate. However, BBN LISP [6] automatically supplies NIL for arguments not transmitted to a function so that (VAR (QUOTE A) NIL NIL) is equivalent to (VAR (QUOTE A)), as translated above.

There are four different translation functions in FLIP. These functions translate respectively, a pattern, a format, an EITHER elementary pattern, and a dictionary. (The last feature is discussed in a later section.) Each of these translation functions are called at the appropriate point in the computation and return the translated version of their input. Furthermore, they also physically alter the list structure of the input. This avoids the necessity for multiple translations.

For example, consider the pattern

```
    ($ :(GET X 'PATTERN) $).
```

The translation of this pattern is

```
    (($) (PATTERN (GET X (QUOTE PATTERN))) ($)).
```

When the function PATTERN is entered, it evaluates

```
    (GET X (QUOTE PATTERN)),
```

and since this value is then to be treated as a pattern, PATTERN calls PATTRAN, the translating function for patterns. If X were an atom and the pattern stored on its property list under the property PATTERN were

```
    ($ $3 'A $ 'B $1 $),
```

the value of PATTRAN would be

```
    (($) ($N 3) (VAR (QUOTE A)) ($) (VAR (QUOTE B)) ($N 1) ($)).
```

Furthermore, the property list of X would have been changed to

```
(... PATTERN ($PATTRAN (($) ($N 2) ... ($N 1) ($))) ...)
```

after PATTRAN was finished operating. Should the same top level pattern ever be used again, or should any other pattern also refer to (GET X (QUOTE PATTERN)), no translation would occur because the pattern has already been translated, as indicated by the special flag $PATTRAN. Similar techniques are used formats, EITHER patterns, and dictionaries. Appendix 3 contains a summary of translation conventions.

<u>The FLIP Read Program</u>

Even before patterns, formats, et al, are translated to their final internal representation, some degree of processing is done on them by the FLIP read program. This program performs the same task as the normal LISP read program: that of transforming a sequence of characters into LISP atoms and more complicated S-expressions. However, the LISP read program only distinguishes the characters "(", ")", "[", "]", """, comma, period, space, line feed and carriage return whereas the FLIP read program is sensitive to these as well as

' ,$,*,=,:,←,+,;,/,↑,?,#,@,and \.

The pattern ($ 'A $3 $ 'B $1 $) is <u>read into</u> LISP by the FLIP read program as

```
(($) (QUOTE A) ($N 3) ($) (QUOTE B) ($N 1) ($)).
```

The only reason why translation is not carried out completely at read time is that the read program cannot know whether a given list will be used as a pattern, a format, in an EITHER pattern, etc., and therefore cannot set up the correct function calls. Instead, the read program separates out the individual entities thereby simplifying the task of translation.

For example:

```
($ X←Y←*=(REVERSE #1)[EQ (CAR *) (CAR #1)])
```

is read in as

```
(($)
 (SET X)
 (SET Y)
 (* = (REVERSE (MARK (1))) (EQ (CAR *) (CAR (MARK (1))))))
```

and then translated to

```
(($)
 (VAR (REVERSE (MARK (1))) SEGMENT
     (EQ (CAR *) (CAR (MARK (1)))))
 (SET1 X X)
 (SET1 Y Y) ).
```

Note that the function SET1 must operate after the elementary pattern VAR has matched, even though it is read in first. The translator performs this reversal.

A complete list of the transformations performed by the read program and the various translation programs may be found in the appendix. The read program itself is discussed in greater detail in Section IV, page 58.

PRETTYFLIP

In order that the user not concern himself about internal representation, a special printing program for FLIP has been written. This program, PRETTYFLIP, is responsible for the output shown here. PRETTYFLIP will accept translated or untranslated data and produce a pleasing and readable output.

No editing facilities designed specifically for FLIP data have been implemented, but it is possible to suppress in translation physical alteration of the input data. This will cause data to be translated anew each time it is used, but will force the data to remain in a more workable form than the translated version - which is designed primarily for efficient FLIP operation **rather than readability.**

The REJECT Mechanism

In addition to eliminating unnecessary interpretation at run time, the translator helps to speed up the operation of the match by providing a quick way of rejecting certain lists. For example, consider the pattern ($ $1 $ #2 'B $). No list can match this pattern which does not contain a B.

During translation of a given pattern, that translator detects and saves any occurrence of the elementary pattern VAR, (except for MARKS) and then stores this information with the translated representation. This information is then used by MATCH, and by the elementary pattern functions for EITHER, REPEAT, and PATTERN to reject lists that obviously will not match. A considerable savings in time is achieved by this simple pattern recognition heuristic.

If the VAR occurs as the first elementary pattern in the pattern, the REJECT mechanism can be even more strict and use the LISP predicate EQUAL on the first element in the input list instead of MEMBER on the whole list. If the first elementary pattern is not a VAR, the translator detects the VAR that is furthest right in the pattern, since this will save the most time. Thus for the pattern ($ $1 'A $ $3 'B $), matches will be attempted only with lists that contain a B.

If the VAR selected by the translator is to match a segment, CAR of its value is used for rejection, unless the value is NIL.

The translator is also sufficiently clever to note that X, in the pattern ($ X←$1 $ X $), cannot be used for rejection. However, the pattern **($ ($ X←$1 $) $ X $)**
will cause difficulties since X is set in
the match, and rejection (using X) is done before the match is tried. For these cases, the special prefix operator "?" is used to indicate to the translator not to use a VAR for rejection purposes. In other words, write

```
($ ($ X←$1 $) $ ?X $)
```

If the user wishes to use **a different** VAR for rejection than the one normally selected, he can similarly write

```
($ 'A $ ?'B $)
```

This would be useful, for example, if B's were more frequent than A's in the lists to be matched by this pattern.

Rejection information is stored at the beginning of the translation of the pattern. Thus the complete translation for the pattern ($ 'A $ 'B $) is

```
( (NIL NIL QUOTE B)
  ($)
  (VAR (QUOTE A))
  ($)
  (VAR (QUOTE B))
  ($))
```

and for the pattern ($ 'A $ *X $)

```
((NIL T . X)
 ($)
 (VAR (QUOTE A))
 ($)
 (VAR X SEGMENT)
 ($))
```

Notice in the rejection information of the latter example, (NIL T . X), the second element is T whereas it is NIL in the first example. This indicates that X is a segment, whereas 'B in the first example is an item. If either 'B or *X had appeared as the <u>first</u> elementary pattern in their respective patterns, the first element in the rejection information information would have been T, e.g. (T T . X) or (T NIL QUOTE B). CDDR of the rejection information is always the form to be matched.

Other Advantages of Translation

The fact that there exists an S-expression representation of the pattern to be matched, as opposed to compiled code, allows the individual elementary pattern functions themselves to take advantage of this information and to perform a rudimentary look-ahead. The most significant use of this information is made by $ and is discussed separately below. However, PATTERN, for example, will not attempt to match with an item if it is not the last item in the list but the PATTERN is. For example, in the pattern ($ $1 ($ $1 'A $)) no match will even be attempted using the elementary pattern ($ $1 'A $) with any list that is not the last element in the input list, regardless of whether or not it contains an A. Similar checks are made by other elementary patterns whenever a savings of time would result.

The translation is also used by CONSTRUCT to reduce CONSes by enabling it to decide when it is necessary to COPY a list and when this can be avoided. For example, in the format (#1 #1 #1), the value of the MARK must be copied at least the first two times. However, since the third elementary format is the last one, it is not necessary to copy its value before attaching it to the list being constructed.

B. CONS Reduction Techniques

The CONS operation in LISP constructs a new word of list structure by taking a cell from the free storage list. The true cost in time for performing a CONS is therefore not only the execution time for the CONS itself, but must include a proportionate amount of the garbage collection time necessary to reconstruct the free storage list when available space is exhausted. Therefore, in situations in which garbage collection is expensive relative to the total processing involved, eliminating superfluous CONSes can drastically reduce computation time. Furthermore, in a LISP system utilizing secondary storage, such as the BBN LISP system, the actual execution time for the CONS may be large, even excluding garbage collection, i.e., because it may involve reading in drum pages. Thus saving CONSes is almost always worth the effort.

This section describes briefly some of the ways in which CONSes are saved in the FLIP system. While they may or may not be directly applicable to another type of program, an awareness of the issues presented here should help all LISP programmers to write more efficient programs.

One can distinguish two cases in attempting CONS reduction. In the first, a CONS can be immediately eliminated by alternate coding. Programs may have a great number of such superfluous CONSes, if only because the straightforward way of writing a program is not always the most efficient.

The second case involves postponing performing a CONS because conditions to be determined later _may_ make is unnecessary. This technique is less obvious and more program dependent than simply eliminating CONSes, but, at least with the FLIP system, often results in far greater savings. Both techniques are discussed below.

Eliminating CONSes

The theme central to techniques for eliminating CONSes is the rather trivial question "Do I really need a CONS?" This question is not trivial when one stops to consider the meaning of CONS.

CONS means construct, i.e., create, a new piece of list structure. If the program does not need a new piece of list structure, it does not require a CONS.

A typical case is the decision between using APPEND or NCONC, the standard LISP functions for joining two lists, e.g., the result of joining (A B C) and (D E F) is the list (A B C D E F). APPEND operates by first CONSing C onto (D E F), and then CONSing B onto (C D E F), and finally CONSing A onto (B C D E F). It thus copies the first list entirely, performing three CONSes. NCONC on the other hand does no CONSes, but physically alters the first list so that the end of it points to the second list. The result in both cases is a list which looks the same.

The choice between APPEND and NCONC is really a choice between creating new list structure, and using old list structure. If the first list will be needed subsequently, for example, to be joined to a third list, just an NCONC cannot be used since it will destroy it. However, this does not mean that APPEND must be used. A situation occurred frequently in FLIP where it was necessary to create and process the join of list X with a list Y, and then the join of X with Z and process this, and so forth. By saving a pointer to the end of X, we were able to perform an NCONC on X with Y, and later recover by locating the point in X-join-Y at which Z was to be attached. Thus, if X had the value (A B C) and Y had the value (D E F), X was physically changed to (A B C D E F), but a pointer was saved whose value was (C D E F). Subsequently, X could be joined to Z, which might be (G H I), by performing RPLACD on (C D E F), giving (A B C G H I). Of course, this destroyed the join of X to Y, but then we were finished with this list. (A by-product of this technique is that the joining can be performed without searching for the end of the first list, since the pointer immediately gives this location.) The convention used in FLIP is to represent a list to be joined in this fashion as a pair of pointers, the first pointing to the head of the list, and the second to the point where the join is to be made. Thus X-join-Y would be ((A B C D E F) C D E F), and to join X to Z would simply require (RPLACD (CDR X) Z).

Postponing CONSes

The technique of postponing CONSes is based on the supposition that the CONS may never be necessary. For example, suppose that initialization of a process requires construction of several lists, and furthermore that the process may in fact terminate without using these lists. In this case, the programmer will save CONSes by not constructing these lists until they are needed, rather than performing all of the initialization at one time at the beginning of the program. Obviously, this will require more effort on the part of the programmer since he must install checks at the points that he wishes to use the lists to determine whether or not they have been constructed.

The most significant savings achieved in FLIP by use of postponement techniques occurs in the treatment of the $ pattern. If the segment matched by the $ is not of interest, the list structure corresponding to this segment need never be constructed. For example, in the transformation specified by

```
    (FLIP X '($ 'A $ 'B $) '(#5 #1))
```

the segment matched by the middle $ is not needed. However, there is no way of determining, in advance, which segments will be needed, if only because many different formats can be used with a single parsing.

The solution adopted in FLIP is to retain enough information to allow the construction of this segment without actually performing all of the CONSes required to create it. By postponing these CONSes until they are actually needed, a considerable savings is realized without any sacrifice in the generality of the $ pattern. This is done by representing the segment matched by the $ as two pointers into the list being matched. The actual form is ((P2 P1). POINTERS), where P1 is the list before the $ matched, and P2 is the list after $ has matched. In other words, the segment matched by $ is the "difference" between the two lists P1 and P2. The tag "POINTERS" is used to distinguish this data type from conventional list structure.

Once this structure is created, the segment it represents can be "lengthened" by simply changing P2, using RPLACA. This will correspond to adding new elements to the end of the segment although no CONSes are performed. This is extremely useful since the $ pattern will often search, attempt a match, fail, and resume the search many times before a complete match ultimately is found.

If the segment matched is referenced by a MARK, the difference between P1 and P2 is computed and stored with a changed representation so that upon subsequent references, no additional CONSes will be performed. If in the interim the segment is lengthened, only those CONSes necessary to lengthen the list structure already created will be performed. At each stage, additional CONSes are performed only if they are required. This, of course, is the essence of the postponement technique.

C. Search Strategy and the $ Function

Consider the pattern ($ $1 'A $) and an input list, or workspace, (A B C D A E F). The translation of this pattern is

```
((\$) (\$N 1) (VAR (QUOTE A)) (\$))
```

indicating that four elementary pattern functions are to be called. Each of these individual functions decide whether or not the elementary pattern embodied in their arguments will match with the current beginning of the workspace. It is necessary that all of them match in order for the pattern to match the list.

The $ function is entered first and it initially matches the null segment. $N then matches with A, as this is the first item in the workspace at that point. VAR is entered, evaluates its argument (QUOTE A), and compares the value, A, to the first item on the workspace. Since the workspace at this point is (B C D A E F), VAR fails, which causes $N to fail.

The $ function now continues searching and extends the segment it matches from NIL to (A). $N matches with B and again VAR fails. This process continues until $ matches with the segment (A B C), $N matches with D, and VAR successfully matches with A. The last $ is entered and matches with the rest of the workspace, which is then (E F). Note that the entire process required four calls to $N, and four calls to VAR.

Now let us consider the pattern ($ 'A $), and the list (B C D E A F G). If the match were carried out in the manner described above, VAR would be entered five times before it matched with A. This would be an extremely slow execution of a very simple search problem. Actually, the match is not carried out in this way. Instead the $ function uses a number of strategies to make a more efficient search.

Instead of expanding incrementally only one item at a time, the $ function looks at the elementary pattern following the $ in the pattern, and expands what the $ will match to include all items in the workspace which could not possibly match the next element. In the above example, the $ looks ahead and sees that the next elementary pattern is a VAR. It evaluates (QUOTE A), and then calls a subfunction which performs successive CDRs on the workspace looking for A. Only when it finds an A does it allow VAR to be entered and the match to proceed. This mode of operation is more efficient than calling VAR a number of times since the interpretation of the next element to be matched is performed only once, inside the $ function, instead of each time the next elementary pattern function is called.

The same technique is used when the next elementary pattern is:

1. a VAR that matches a segment;
2. a VAR that matches a MARK;
3. a PATTERN that matches an item, provided the translation contains rejection information;
4. a PATTERN that matches a segment, provided the translation contains rejection information indicating that the <u>first</u> elementary pattern function in the PATTERN is a VAR;
5. a REPEAT provided the translation contains rejection information indicating that the repeated pattern begins with a VAR; or
6. an EITHER provided the translation contains rejection information indicating that <u>each</u> of the alternatives begins with a VAR.

In each case, the $ function determines once what to search for and then calls a special function which does nothing but perform that search. Predicates used in conjunction with $N or VAR patterns are taken into account in the search. In addition, $ is sensitive to the presence of predicates modifying its own segment, but does not apply them until after it finds a suitable point in the workspace to attempt a match for the following elementary pattern.

As an indication of the time saved by these techniques, results of some test cases are presented below. Computation time is given in seconds, * and each computation was run ten times and the average time reported. The column labeled WITH corresponds to a match executed with the fast $ search strategy in effect. The column labeled WITHOUT corresponds to a match in which the next elementary pattern function was called each time the $ extended its segment. Note that the elementary pattern following the $ matches the fourth element of the workspace in examples 1, 3, and 5, and the twentieth element in examples 2, 4, and 6. Going from 4 to 20 with the fast $ involved only a 25% increase in computation time, whereas going from 4 to 20 using no search strategy required a 400% increase. In other words, without fast search techniques the time required to match is directly proportional to the length of the segment matched by $.

* The actual computation time is not significant, since these particular examples were run with an uncompiled FLIP in a highly competitive time-shared environment on the SDS 94Ø.

	WORKSPACE	PATTERN	WITH	WITHOUT
1.	(A B C D)	($ 'D $)	1.4	1.9
2.	(A B C D E F G H I J K L M N O P Q R S T)	($ 'T $)	1.7	9.0
3.	((A B C D) (E F G H) (I J K L (M N O P))	($ ('M $) $)	1.6	3.4
4.	((A B C D) (B C D E) (C D E F) etc. (T U V W))	($ ('T $) $)	2.1	16.1
5.	(X Y Z D A B C)	($ EITHER['C $1; 'D $2; 'E] $)	3.9	5.7
6.	(F G H I J K L M N O P Q R S T U V W X D A B C)	same	4.9	25.4

Consider the pattern ($ EITHER['C $; 'D $1] 'E $). If a C is found in the workspace, the $ inside of the EITHER is entered. However, at this point, the next elementary pattern function call is not to VAR, but to a special housekeeping function inserted by EITHER into the list of function calls immediately following the $. This function has the task of taking the images of the elementary patterns inside of the EITHER and grouping them together into a single parsing. It is important that this function operate before any other pattern functions are called so that patterns such as ($ EITHER['C $1; 'D $2] #[2,2] $), or
        ($ EITHER['C $1; 'D $2] #2 $) will operate correctly.

Note in our original example, ($ EITHER['C $; 'D $1] 'E $) that in order for the match to succeed, the $ must match everything up to an E. Therefore, $ looks <u>through</u> the housekeeping function call to the VAR, searches for an E exactly as before, and upon finding it, allows the match to proceed normally, i.e., calls the housekeeping function. However, the workspace is now positioned correctly so that the VAR <u>will</u> match on the first call. A similar situation occurs with the pattern (X←$ 'A $), which translates to (($) (SET1 X X) (VAR (QUOTE A)) ($)). The function SET1, which performs the assignment of the variable X to the segment that $ matches, must be called after $ matches, and before VAR. However, $ looks through SET1, finds the VAR, searches for A, and upon finding it, calls SET1 in the normal way.

$ will look through an indefinite number of calls to SET1 or housekeeping functions until it finds an elementary pattern function call specifying something that it can search for in the workspace. It then searches the workspace for the desired item or segment until it finds it; if the $ runs out of workspace, the next pattern cannot match and therefore the $ fails. Once $

locates the correct point in the workspace, it calls the next elementary pattern function and proceeds as before. In the special case that the next elementary pattern function was a VAR or $N, $ performs the addition of this pattern image to the parsing as well as its own image and bypasses the call to the function entirely.

If a failure occurs subsequently, $ continues the search by again looking for suitable place in the workspace. No additional interpretation or searching through the list of elementary pattern function calls is necessary. For example, in the pattern ($ 'A $1 'C $) and the list (X Y Z A M N O P A B C D), $ finds that the next elementary pattern is a VAR, and searches for A. After matching [X Y Z] and passing control to the third elementary pattern function which is $N, a failure occurs in the match with 'C. At that point, control reverts to $. $ takes up the search with the workspace (W M N O A B C) and looks for an A as before. It finds the second A, and this time a match occurs with $ matching (X Y Z A W M N O).

If $ finds an elementary pattern function call which matches something that it cannot search for specifically, such as another $, or the subpattern ($ $3), before it finds something that it can search for, it does a slow search, i.e., it initially matches the null segment, and calls the next elementary pattern function, etc. The $ thus assumes that any position in the workspace is as likely to produce a match as any other.

If $ exhausts the list of elementary patterns before encountering a pattern that it can search for, or one that it definitely cannot search for, it automatically matches with the remainder of the workspace, and calls the next elementary pattern function (if there is none, $ returns immediately with the successful parsing). For example, in the pattern ($ 'A X←$), which translates to (($) (VAR (QUOTE A)) ($) (SET1 X X)), the second $ matches with the rest of the workspace and calls SET1 to perform the assignment.

Another strategy of the $ function uses information about the position of an elementary pattern in the pattern. If the elementary pattern following $ is the last elementary pattern, and is to match an item, $ will immediately go to the end of the workspace, less one, before attempting a match. Thus in the pattern ($ 'A $ ($ 'B $)), the second $ will expand its segment through all elements except the last one, ignoring any intervening lists. A similar technique is employed for $N where N≠1. Thus in the pattern ($ 'A $ $3) the second $ will immediately match a segment consisting of the rest of the workspace less 3.

$$

The fast $ can cause difficulty in patterns where a MARK follows the $ which refers to either the $ itself, such as in the pattern ($ #1), or a portion of the match not yet completed, such as in the pattern ($ *('A $1 $) #[2,2] $). A special elementary pattern, $$, is available for this contingency. $$ acts exactly like $ in slow search mode. It never looks ahead or utilizes any of the fast search techniques described. The elementary pattern function SLOW$ does the work for $$.

SECTION IV

USING FLIP

This section discusses FLIP as a working system. It describes those functions that may be useful to a prospective user, and discusses some additions and extensions to FLIP. It assumes that the reader has read section II, describing the FLIP formalism, and has at least skimmed Section III, describing the implementation.

A. Functions

REED

Section II did not discuss the representation of FLIP elements as LISP S-expressions, e.g., how the pattern ($ 'A X←$1[NUMBERP *] $) is represented internally as an S-expression consisting of atoms and dotted pairs. However, this representation need not concern the user, because a special read program has been written for FLIP which accepts FLIP formalism as well as standard S-expressions. The name of this program is REED, and its value is one S-expression read from the indicated input file.

REED is a function of one argument, which determines the CONTROL setting. If this argument is NIL, CONTROL is set to NIL, the normal setting with LISP READ, and the REED program must wait until a carriage return before receiving any characters from the line buffer. However, REED will respond to Control A and Control Q in the usual way.

If REED's argument is T, the characters are delivered to REED as they are typed. Used in this way, REED counts parentheses and performs a carriage return and line feed, and returns its value when the count reaches zero. Control W and Control Q correspond to Control A and Control Q for READ, although their interpretation is somewhat different.

The character control W is used to delete the last element read, whether it be an atom or a list. REED echoes the element on the same line and performs a carriage return. Control W cannot be used beyond the beginning of a list.

Example: (Control W is underlined)

```
REED(T)
($ 'BW(QUOTE B)
W($)
WW$ 'A X $WW1W($N 1)           (1)
$1[NUBERPWNUBERP
WWNUMBERP *]W($N 1 (NUMBERP *))  (2)
)
($ (QUOTE A) X)
```

(1) Initial Control W's ignored because every element in the list has already been deleted. The two control W's are ignored following the <u>second</u> $ because REED has been called recursively, therefore, there is nothing to erase. After "1" is typed completing the FLIP token, control W may erase it, as in the example.

(2) Initial control W's are ignored because REED has been called recursively for predicates. Again, it is necessary to complete the expression before it can be erased.

Control Q is used to delete all elements back to the last open left parentheses or bracket. There is no line deletion. REED does a carriage return-line feed and echoes what has been deleted.

Example:

```
REED(T)
(A (B (C D) EQ
(B (C D) E
WA
WWQ                               (1)
(
X
X
REED(T)
($ 'QQXQ                          (2)
($) (QUOTE X)
)
NIL
```

(1) Control W is ignored as everything in the list has been deleted. Control Q deletes through the left parenthesis. Thus the situation is the same as when REED was first entered, and typing X causes REED to return X.

(2) The first two control Q's are ignored because REED has been called recursively. After this token has been completed by typing X, it is possible to delete everything.

Note: "." is not a break or separator character for REED. Thus "A.B" is a legitimate atom. To write (A . B), be sure to space between A, ".", and B. To write atoms unusually spelled as far as REED is concerned, use double quotes the same as with the LISP READ, e.g., "[($](()" is a legitimate atom.

Note: If REED is given a second argument, it will treat this as a file name. Otherwise it reads from the standard input file.

PILOT

A convenient way to use REED is via PILOT. PILOT is a function of one argument, that argument being used to determine the control setting for REED. PILOT reads and executes doublets for evalquote, using REED instead of READ. PILOT is buffered using ERRORSET, and prints "PROCEED": following and error. To exit from PILOT, type STOP, at which point PILOT returns NIL.

MATCH

MATCH is a function of four arguments. The first argument is the list to be matched. The second argument is the input pattern, which MATCH gives to PATTRAN (see below) to be translated. The third argument is optional and is a dictionary for use during the match. The dictionary feature is described later in this section. MATCH assumes the dictionary, if any, is already translated. The fourth argument is also optional and is an A-list. MATCH initializes the variables in this A-list before beginning the matching operation. The effect is identical to performing the bindings via the elementary pattern SET. For example:

```
     (MATCH X Y NIL '((FOO . 1) (FIE . 2)))
```

is the same as (MATCH X Z), where the first two elementary patterns in Z are (SET FOO 1) and (SET FIE 2). In the current implementation, variables are bound by performing the LISP function SET. This means that the programmer should use distinctive names for these variables.

The value of MATCH is either a parsing, in the event of a successful match, or NIL in case of a failure.

PATTRAN

PATTRAN is a function of one argument, a pattern to be translated. The value of PATTRAN is the translated version of this pattern. If the variable $TRAN is set to T, its normal setting, PATTRAN also physically changes its input pattern so that CAR of the input is the atom $PATTRAN, and CDR of the input is the translated version of the pattern. Whenever PATTRAN is called with a pattern already translated, it immediately returns the translated version without further processing. The same technique is used by the other translating functions, ORTRAN, FORMTRAN, and DICTRAN, all of which are sensitive to the setting of $TRAN.

PDEFAULT

The function PDEFAULT allows the user to introduce new translating conventions and notational schemes to PATTRAN. PDEFAULT is called whenever an element is encountered in a pattern that does not correspond to a recognizable elementary pattern, i.e. it does not correspond to notation utilizing EITHER, REPEAT, *, =, QUOTE, #, etc. PDEFAULT is a function of one argument, the element in question. The value of PDEFAULT, if not NIL, is translated instead of the unkown element. For example, one might define PDEFAULT so that the element (N<$<M) translated the same

as $[AND (GREATERP (LENGTH *) N) (LESSP (LENGTH *) M)], and could therefore be used to match with segments of length between N and M.

When the value of PDEFAULT is NIL and the unknown element is a list, it is translated as a subpattern. This corresponds to the definitions given earlier on page 18. When the unknown element is an atom (with the exception of the atoms .. and ... , which are translated as $ and $$ respectively) the value of (LIST DEFAULT ATOM) is used; thus setting DEFAULT to QUOTE will cause all atoms that appear in a pattern to be quoted.

The normal setting for DEFAULT is = and PDEFAULT is initially defined as (LAMBDA (X) NIL). Thus, if the user does not change either the variable DEFAULT, or the function PDEFAULT, the translation conventions are identical to those specified earlier, on pages 13 and 18, for VAR and subpattern.

MATCH2

MATCH2 is the function that does the work in matching. It is a function of three arguments: the current workspace, the current pattern, (already translated) and the current match. Whenever an elementary pattern function determines that its elementary pattern matches, it calls MATCH2 with the new workspace pattern and match to perform the matching for the rest of the workspace and pattern. In the event of a successful match, the value of MATCH2 is a parsing. In the event of failure, CDR of the value of MATCH2 is NIL (CDR of a parsing is never NIL). Thus except for some initialization, MATCH is essentially defined as

```
     (PROG (X)
          (SETQ X (MATCH2 WS (PATTRAN PATT) NIL))
          (COND
             ((CDR X) (RETURN X))
             (T (RETURN NIL))))
```

<u>Variables used inside of MATCH</u>

The following variables may be of interest to the user:

WS	current workspace
PATT	current pattern list
MATCH	current level match
$MATCH	push down list of higher level parsings, used by subpatterns, EITHER, and REPEAT elementary patterns
$A	current A-list
$D	current dictionary

Thus the elementary pattern $1[EQ (CAR WS) FOO] has the same effect as the elementary pattern =FOO.

In addition to these variables, the variable TRAC controls a tracing option in MATCH2. When TRAC is set to T,the workspace, parsing, and A-list, if any, are printed each time MATCH2 is called, or equivalently, for every elementary pattern. If the value of TRAC is a <u>list</u>, printing occurs whenever an elementary pattern function is a member of this list, or if its position, from the right end of the pattern list, is a member of this list. Thus if TRAC is set to (1 $N), tracing will occur in the pattern ($ FOO←$1 'A $) whenever either the function $N or the final $ is entered. This option is useful for debugging.

NOCONS

It is possible to use MATCH as a pure predicate to produce a value T or NIL but no parsing. When used in this mode, no CONSes are performed to construct the parsing, which means that for most cases, no CONSes are required for the entire operation.

This has been achieved by using calls to a function KONS, instead of CONS throughout MATCH. Executing (NOCONS T) will cause the definition of KONS to be altered so that it merely returns an appropriate quoted S-expression. Executing (NOCONS) will cause the definition of KONS to be restored to CONS.

Since no parsing is being saved, MARKs cannot be used. However, it is still permissible to use the assignment elementary pattern. Thus while the pattern ($ $1 $ #2 $) will not work with NOCONS turned on, the pattern ($ FOO←$1 $ FOO $) will. In the latter case, no CONSes will be performed. Of course the pattern ($ FOO←$2 $ *FOO $) will require two CONSes each time the assignment is made in order to create the list structure corresponding to the segment matched by $2.

NOCONS During a Match

It is dangerous for the user to execute a NOCONS while inside of a match because of certain initialization problems. Therefore, the special elementary pattern \ is provided for turning off CONSes for certain portions of the match. When \ appears as the first element in the PATTERN, EITHER, or REPEAT elementary patterns, it means that the parsing is not saved for this elementary pattern only. NOCONS will be restored after this pattern is executed.

For example:

```
    MATCH((A B C D E F G) ($ EITHER[\ 'C $2; 'D $1] $))
```

will yield the parsing [A B] [C D E] [F G], but no subparsing for the EITHER elementary pattern.

Similarly, MATCH ((A B C D E F G) (\ $ 'D $)) will yield a parsing indicating that (A B C D E F G) was matched, but no information about the component parts. This technique is also useful for turning off CONSes for certain selected rules in a rule set.

MATCHP

MATCHP is a pure predicate. It performs (NOCONS T), calls MATCH, restores NOCONS, and returns with the value of MATCH, which is either T or NIL.

MAPMATCH

MAPMATCH is a function of five arguments. The first argument is a list and the second a pattern as with MATCH. Similarly, the fourth argument is the optional dictionary and the fifth argument the optional Alist. The third argument is a functional argument. The effect of MAPMATCH is to apply this function to all possible matches using the given pattern and workspace. For example, the value of MAPMATCH given

```
     (A B C D E),($ $2 $), and (LAMBDA (X) (CONSTRUCT X '(#2)))
```

is

```
     ((A B) (B C) (C D) (D E))
```

It is best to process the parsing immediately as done above, instead of saving it, because many of the elementary pattern functions, e.g., $, EITHER, REPEAT, etc., will physically alter the parsing.

CONSTRUCT

CONSTRUCT is a function of four arguments. The first argument is a parsing, and the second argument is a format, which CONSTRUCT gives to FORMTRAN to be translated. The third and fourth arguments are exactly the same as for MATCH, i.e., a dictionary and an A-list. CONSTRUCT can be used with no parsing, provided the format does not contain any MARKS, or EITHER or REPEAT formats which require a parsing. For example:

```
    (CONSTRUCT NIL '(X *=(GET Y Z) Y))
```

is the same as

```
    (CONS X (APPEND (GET Y Z) (LIST Y)))
```

During the operation of CONSTRUCT, the atom "*" is bound to the entire top level list which was matched. For example, matching (A B C D E) with ($ $1 'C $) and constructing with (#2 *) will yield (B (A B C D E)). The alternative is of course (#2 (#1 #2 #3 #4)).

FORMTRAN

FORMTRAN is the CONSTRUCT counterpart of PATTRAN. It is a function of one argument, a format to be translated. Its value is the translated version of the format. If the variable $TRAN is set to T, FORMTRAN physically changes its input format so that CAR of the input is the atom $FORMTRAN, and CDR of the input is the translated version.

FDEFAULT

FDEFAULT is the counterpart of PDEFAULT. If the value of FDEFAULT for an atom is NIL, * (LIST DEFAULT atom) is translated instead. If the value of FDEFAULT for a list is NIL, the list is translated as a subformat. FDEFAULT is initially defined as (LAMBDA (X) NIL).

CONSTRUCT2

CONSTRUCT2 corresponds to MATCH2. The variables used by CONSTRUCT2 are MATCH, FORMAT, $A, and $D. There is no tracing option. NOCONS does not affect the operation of CONSTRUCT or CONSTRUCT2.

---

* The atom .. is translated the same as #1 except where it is the last elementary format in a format, in which case it translates the #-1. Thus matching with (.. 'D ..) and constructing with (.. 'E ..) is the same as matching with ($ 'D $) and constructing with (#1 'E #-1).

FLIP

FLIP is a function of five arguments. The first argument is a list to be matched, the second is a pattern, and the third argument is a format. The fourth argument is an optional dictionary and the fifth an optional Alist. FLIP calls MATCH with arguments 1, 2, 4, and 5, and returns NIL if MATCH fails, or else returns the value given by calling CONSTRUCT with the value of MATCH and arguments 3, 4, and 5.

TRANSFORM

TRANSFORM is a function of four arguments. The first argument is a list to be transformed, and the second is a set of rules. The third and fourth arguments are optional dictionaries.

A rule set is a list of rules and optional (atomic) labels. Each rule consists of a pattern, a format, and an optional GOTO label.

TRANSFORM evaluates a rule set by starting with the first rule and calling match with its pattern and TRANSFORM's first argument as workspace. If a match succeeds, the parsing is given to CONSTRUCT with the format. The value of TRANSFORM is the value of CONSTRUCT. If the match fails, control passes to the next rule.

Special Cases:

1. If an atomic GOTO occurs in a rule, and the match is successful, control transfer to that labelled rule after performing the indicated CONSTRUCT operation. The result of CONSTRUCT is used as the new workspace. Note: do not use NIL as a label.

2. If a GOTO label is listed, control goes to that labelled rule if, and only if, the match fails. In this case, no CONSTRUCT operation is performed, and the workspace is unchanged.

3. The special GOTO label, TOP, is the label for the first rule of the rule set.

4. The special label * is the label for the very next rule.

5. The special label ← is the label for the current rule.

6. The special labels EXIT and BOTTOM are labels for the end of the rule set. For example: the rule

   ```
   (pattern format (EXIT))
   ```

   means EXIT from TRANSFORM if this match fails. Otherwise, CONSTRUCT is called and control passes to the next rule. No label is treated the same as EXIT.

7. The user may effect a computed GOTO by setting the variable LABEL to the label he wishes control to transfer. If LABEL is set during the match, transfer will occur regardless of whether or not the match was successful. If LABEL is set during the construct, control will only occur if the match

was successful, since otherwise CONSTRUCT would not have been entered. LABEL is automatically reset to NIL before each rule.

RTRAC

RTRAC controls a tracing option for rules. When RTRAC is set to T, TRANSFORM prints out the label of each rule entered, and the value of the list being processed. If RTRAC is a list, tracing occurs only for rules whose label is in the list.

ADDRULE

ADDRULE is a function designed to facilitate adding rules to rule sets at arbitrary locations. It is a function of four arguments. The first argument is the rule set to be modified, * and the second argument is the rule or label to be added. The third argument specifies where the change is to take place, with TOP, EXIT, and BOTTOM specifying locations consistent with TRANSFORM. Numbers specify the corresponding numbered entry with numbering proceeding from the top of the rule set and <u>both rules and labels</u> being counted. Other atoms are treated as ordinary labels and specify the location of that label.

* For functions of the form (LAMBDA & (TRANSFORM & 'ruleset &)), the name of the function may be used as the first argument to ADDRULE. ADDRULE will obtain the rule set from the definition.

There are three different modifications allowed at the specified location: insertion before, insertion after, or replacement. The normal operation is insertion after. To specify insertion before, the label or number must be listed. For replacement, the fourth argument to ADDRULE must be T. Replacement takes precedence over insertion. In all cases, the changes are destructive.

Example: (ADDRULE RULES RULE (3)) will insert RULE before the third entry in RULES.

Example: (ADDRULE RULES RULE 3 T) will replace the third entry in RULES with RULE.

Example: (ADDRULE RULES RULE FOO) will insert RULE just after the label FOO.

Example: (ADDRULE RULES RULE TOP) will insert RULE just after the label TOP, in other words before the first entry in RULES.

PRETTYFLIP

PRETTYFLIP is a function of three arguments. If the second and third arguments are NIL, PRETTYFLIP acts just like PRETTYPRINT. If the second argument is present, PRETTYFLIP acts like PRETTYDEF, i.e., it can be used to write a file complete with DEFINEQ and STOP.

The output produced by PRETTYFLIP is identical with the original sequence of characters typed in to REED except for the fact that carriage returns and spacing are introduced to produce an aesthetic format. In some cases, the output may differ slightly between translated and untranslated versions, but in either case, the output produced, if read back in again, would result in the exact same function definition. The differences occur because PRETTYFLIP cannot tell that a list will be used as a rule set, or a dictionary, until after it has been translated, and therefore cannot use a format designed especially for these types.

PF

The function PF plays the role of SUPERPRINT for PRETTYFLIP. PRETTYFLIP calls PF with the definition of a function. However, PF can be used to print output that is not a function definition, for example, a property list, or a portion of a function definition being edited.

B. THE DICTIONARY FEATURE

The use of the dictionary is one of the more interesting aspects of CONVERT, another pattern-driven programming language embedded in LISP [7]. The CONVERT dictionary offers an alternate way of specifying operations that can already be specified in FLIP using a somewhat different notation. However, some operations that can be written very simply using the dictionary are much harder to write in FLIP without it. The dictionary feature described below has been added to FLIP in order to provide it with this increased notational flexibility. Wherever possible, the CONVERT notation has been transferred intact to FLIP, and the interpretation of dictionary operations within the FLIP environment is consistent with the corresponding operations in CONVERT.

The CONVERT Dictionary

The CONVERT dictionary is a list which contains the "definitions" for some or all of the elements appearing in a pattern or skeleton, the CONVERT counterpart of a format. Whenever an atom is encountered during the operation of RESEMBLE or REPLACE, the CONVERT counterparts of MATCH and CONSTRUCT, the definition of the atom is obtained from the dictionary, and the appropriate action is taken. This extra level of interpretation costs the program in efficiency, but in return the ability to modify definitions dynamically can be an extremely powerful asset.

Each definition consists of three parts: the variable being defined, its MODE, and a third parameter containing additional information required by the mode. The dictionary itself is a list of the form

```
    (VAR1 MODE1 PAR1 VAR2 MODE2 PAR2 etc.)
```

consisting of the definitions strung out at the top level. For example, the mode PAT indicates that a variable represents an entire pattern. Thus using the dictionary (X PAT ($ $1 $ #2 $)), the FLIP pattern ($ X $ X $) would be equivalent to the pattern

```
    ($ ($ $1 $ #2 $) $ ($ $1 $ #2 $) $).
```

Note that if the value of X is ($ $1 $ #2 $), the latter pattern could also be written as ($:X $:X $).

The following is a list of some of the MODEs in CONVERT:

X VAR G	the variable mode. The letter X is used to represent an expression however complicated.
X UAR *	the undefined variable mode. X will match anything, but the entry in the dictionary is changed to read X VAR E, so that as a consequence if X appears as part of a more complicated pattern it will have to match the same quantity each time it occurs.
X PAT P	the pattern mode. The letter X represents an entire pattern.

X PAV P	the pattern variable mode, which is a combination of the modes PAT and UAR. Not only must the pattern P match E, but the dictionary is altered to read X VAR E so that if X occurs several times it will always match the same expression E.
X BUV (P ...)	the bucket variable mode. This mode is similar to the PAV mode, but rather than requiring that the same expression match every occurrence of X, we simply make a list of these expressions. Thus X in the BUV mode will match any expression matched by P, and the dictionary is modified to read X BUV (P E ...).
X CUV (P K)	the counting variable mode. This mode is similar to BUV, but rather than listing the matching expressions, we simply count them. The dictionary is modified to read X CUV (P K+1) after each match.

If a variable in the dictionary is enclosed in parenthesis, it is called a fragment variable, and matches with a fragment, i.e., a segment of a list. This corresponds to using the prefix operator "*" in FLIP.

The FLIP Dictionary

The basic differences in the FLIP and CONVERT versions of the dictionary feature arise from the difference in philosophy between the two languages. CONVERT is a richer, more recursive language than FLIP, and quite complicated operations can be specified very concisely. This is partly because nearly every expression is interpreted, broken down into its composite subexpressions, until atomic elements are encountered, and each of these are looked up in the dictionary. This means that definitions can be built upon definitions in a very sophisticated way.

FLIP on the other hand is oriented more towards efficiency of operation, occasionally at the expense of elegance. Consistent with these goals, the dictionary has been implemented in such a way that its presence does not penalize a program that did not utilize it. As an example, the translating functions determine which atoms are dictionary entries, and only these are looked up at run time. Thus if Y is a dictionary entry, the pattern (X Y Z) might translate to ((VAR X) (DICVAR Y) (VAR Z)), and only Y would then be looked up at run time. This places the burden on the programmer of declaring which elements are dictionary variables, but eliminates much of the interpretive aspects of the dictionary.

The dictionary is given to the functions TRANSFORM, FLIP, MATCH and CONSTRUCT as an optional argument. For MATCH and CONSTRUCT, the dictionary is the third argument. It is assumed to be previously translated. For FLIP and TRANSFORM, the dictionary is the fourth argument. Both of these functions will first translate it using the function DICTRAN, which modifies its input in a manner analagous to PATTRAN and FORMTRAN discussed earlier. In addition, TRANSFORM will accept as its third argument a dictionary consisting of just a list of variables. Each of these variables is automatically given the MODE UAR. Thus (TRANSFORM X Y '(A B C)) is equivalent to (TRANSFORM X Y NIL '(A UAR -- B UAR -- C UAR --). *

* It is unimportant what follows UAR in the dictionary, merely that something does follow it to preserve the periodicity of the dictionary.

The dictionary is available throughout the course of matching and construction, and the value of dictionary variables can be referenced by means of the prefix operator "@". The value of a dictionary variable is usually the parameter which comprises the third part of its definition. However, for variables of mode BUV, the value is the list, in reversed order, of the expressions matched by the bucket variable, and for variables of mode CUV, the value is the count. Thus, if X has mode BUV, =(CADR @X) will match with the second element in the bucket corresponding to X. "@" is read by REED as a call to a function which looks up the value of X. Thus the form (CADR @X) can be evaluated directly without further interpretation, and any requests for dictionary variables will still be handled correctly. This is similar to the treatment of # for marks.

It is unnecessary to use the prefix operator @ for a dictionary variable that is being used as an elementary pattern or format, because the translators check the dictionary before applying the default declaration. Thus the pattern ($ X $) is equivalent to ($ @X $), provided X is defined in the dictionary at the time of translation. Note that if X has CUV mode, the elementary pattern X differs from the elementary pattern =@X. The former will match with the pattern specified in the definition of X, and index the count. The latter will match with a single number which is equal in value to the current count, and will not make any changes in the dictionary.

The translation of the dictionary consists of converting dictionary entries into their corresponding function calls, which can then be inserted into the pattern or format at the appropriate point. For example, the translation of (X) UAR --, which specifies that X is to match a segment of the list, consists of calls to two functions: $ and SETD. When the elementary pattern X is first encountered, its (current) definition is obtained from the dictionary, and attached to the front of the pattern list. Then the match continues and the function $ is entered. $ uses the fast search strategy described earlier. It looks through the SETD function, and any housekeeping or assignment functions it may encounter until it finds something upon which it can concentrate its search. After $ finishes operating, SETD is entered. The arguments to SETD, which were determined at translation time, indicate that a new definition must be entered in the dictionary. This entry will define the variable X as having mode VAR and value whatever the $ matched. The actual function call generated for this entry is simply (VAR (QUOTE s) SEGMENT), where s is the segment matched by $. If X is subsequently encountered, it will be treated the same as though it were initially defined as a fragment variable with VAR.

Whenever a failure occurs in the match, the dictionary is restored before the match continues operating. This involves restoring all variables having MODE VAR, BUV, CUV and PAV to the values they had at the point the match is to be resumed. For example, if X and Y have VAR MODE, then Y will be restored in the pattern

```
    ($ X $ EITHER[Y -- ; --] $)
```

if the first alternative of EITHER fails, but X will not be affected.

If the match is successful, the variable NEWDIC is set to the new dictionary before exiting from MATCH. NEWDIC is already in translated form since any changes made to the old dictionary were performed by SETD. FLIP and TRANSFORM both automatically transmit NEWDIC to CONSTRUCT as its third argument. Thus changes made in the dictionary during matching can be used in constructing.

Example: the following FLIP expression embodies a standard trigonmetric formula.

```
    (FLIP W '(COS X COS Y - SIN X SIN Y)
            '(COS =(PLUS @X @Y))                 *
            '(X UAR - Y UAR -))
```

If W is (COS 23 COS 19 - SIN 23 SIN 29) the value of the FLIP expression will be (COS 42).

Modes Available in the FLIP Dictionary

There are currently ten modes available in the FLIP dictionary: VAR, UAR, PAT, PAV, BUV, CUV, EXPR, SKEL, REPT, and CONT. Variables defined with modes VAR, UAR, PAT, PAV, BUV, and CUV may be used as elementary patterns, and variables defined using modes EXPR, SKEL, REPT, and CONT may be used as elementary formats. The interpretation for each mode is given below, along with the value of dictionary variables defined with that mode. Note that =@X is always a legitimate elementary pattern or format, regardless of the mode of X. However, if a variable is defined using modes EXPR, SKEL, REPT, or CONT, it cannot be used as an elementary pattern directly. Because of the implementation, if X is defined using modes VAR, UAR, PAT, PAV, BUV, and CUV, it may be used as an elementary format. The effect is the same as =@X, or *=@X. Essentially, each of the modes VAR, UAR, PAT, PAV, BUV, and CUV are interpreted as EXPR when in CONSTRUCT.

The segment-item distinction for dictionary variables used as elementary patterns or elementary formats is based on whether or not the variable was enclosed in parentheses in its dictionary definition, as in the CONVERT dictionary. The prefix operator * has no effect on dictionary variables.

* This assumes that DEFAULT is set to (QUOTE -). Otherwise, it would be necessary to substitute 'COS, 'SIN, and '- for COS, SIN, and -, respectively.

X VAR G	value is G itself (not the value of G), and X matches G. Using X is thus exactly the same as using the expression (QUOTE G).
X UAR *	value of X is NIL. X matches the same as $1 (or $ if X is enclosed in parentheses), and then a new entry is added to the dictionary corresponding to X VAR S, where S is the item matched by $1 (or (X) VAR S, where S is the segment matched by $).
X PAT P	value of X is P. X matches the same as the elementary pattern P.
X PAV P	value of X is NIL. X matches the same as the elementary pattern P and then a new entry is added to the dictionary corresponding to X VAR S, where S is the item matched by P (or (X) VAR S, where S is the segment match by P).
X BUV P	value of X is <u>initially</u> NIL. X matches the same as the elementary pattern P. Each time P matches, a new entry is added to the dictionary corresponding to a BUV mode variable whose value is the result of CONSing the item or segment matched by P to the previous value. Thus at each point, the value of X is a list, in reversed order, of the expressions that were matched by P.
X CUV P	value of X is <u>initially</u> zero. X matches the same as the elementary pattern P. Each time P matches, a new entry is added to the dictionary corresponding to a CUV mode variable whose value is one more than the previous value. Thus at each point, the value of X is the number of times that P matched.

X EXPR G	format counterpart for VAR mode. Value of X is G.
X SKEL F	analogue of PAT. The value of constructing with F as an elementary format is added to the list being constructed. The value of X, i.e., @X, is F.
X REPT R	When used as an elementary format, X should not appear by itself but in the form (X A1 A2 ... An). The value of this entire expression is computed by first constructing with the format (A1 A2 .. An), and then transforming this result using the rule set R. (If R is either atomic or an expression of the form ((LAMBDA ..) ..), the value of R is used as the rule set.) The <u>original</u> dictionary is used. Basically, the REPT mode is a mechanism for defining functions internal to a particular transformation. The value of X, i.e., @X, is R.
X CONT R	Same as REPT except <u>modified</u> dictionary is used during transformation.

<u>Example</u>

The following is a FLIP program to perform symbolic differentiation and simplification. It is a straightforward adaptation of the CONVERT program appearing on page 613 in [7], except that only some of the simplification rules are shown. The basic idea of DERIV is that the connective, +, -, *, /, or ↑ is identified in the rule set and then the appropriate simplification rules are called by means of the dictionary variables SPLUS, SMINUS, STIMES, SDIV, SEXPT. <BEGN> is explained below in section on CONVERT primitives. Essentially, it means apply entire process to what follows <BEGN>.

```
(DERIV
  (LAMBDA (EXPR VARBLE) (TRANSFORM
      EXPR
      '((=VARBLE 1)
        ($1[ATOM *] Ø)
        ((.. + ..) (SPLUS DL DR))
        ((.. - ..) (SMINUS DL DR))
        ((.. * ..) (SPLUS (STIMES LL DR) (STIMES
              DL
              RR)))
        ((.. / ..) (SDIV (SMINUS (STIMES RR DL)
              (STIMES LL DR)) (SEXPT RR 2)))
        ((.. ↑ ..) (STIMES RR (STIMES (SEXPT
               LL
               (SMINUS RR 1)) DL))))
      NIL
      '(K          PAV       $1[NUMBERP *]
        L          PAV       $1[NUMBERP *]
        LL         SKEL      =(UNLIST #1)
        RR         SKEL      =(UNLIST #3)
        DL         SKEL      (<BEGN> =(UNLIST #1))
        DR         SKEL      (<BEGN> =(UNLIST #3))
        SPLUS      REPT      (((K L) =(PLUS @K @L))
                              (($1 Ø) #1)
                              ((Ø $1) #2)
                              (($1 #1) (2 * #1))
                              (($1 $1) (#1 + #2)))
        SMINUS     REPT      ((($1 Ø) #1)
                              ((K L) =(PLUS @K (MINUS @L)))
                              ((Ø $1) (- #2))
                              (($1 #1) Ø)
                              (($1 $1) (SPLUS #1 (- #2))))
        STIMES     REPT      (((.. Ø ..) Ø)
                              (($1 1) #1)
                              ((1 $1) #2)
                              ((K L) =(TIMES @K @L))
                              (($1 #1) (#1 ↑ 2))
                              (($1 $1) (#1 * #2)))
        SDIV       REPT      ((($1 1) #1)
                              ((Ø $1) Ø)
                              (($1 #1) 1)
                              (($1 $1) (#1 / #2)))
        SEXPT      REPT      ((($1 1) #1)
                              (($1 Ø) 1)
                              ((1 $1) 1)
                              (($1 $1) (#1 ↑ #2)))
        ))))

(UNLIST
  (LAMBDA (X) (COND
      ((CDR X) X)
      (T (CAR X)))))
```

Segments and Items

Consider the dictionary (X PAT $3 (Y) PAT $1). According to the dictionary, X is to match the same as the elementary pattern $3 and Y is to match the same as the elementary pattern $1. According to the CONVERT conventions, however, X is to match an item and Y a segment. However, the elementary pattern $3 always matches a segment, and the elementary pattern $1 always matches an item. This ambiguity is resolved by adopting the convention that when the dictionary variable is enclosed in parentheses, the translation of the accompanying pattern indicates it is to match a segment, if possible. Thus elementary patterns that always match segments do so regardless of whether the corresponding dictionary variable is listed. Elementary patterns that always match items are similarly unaffected by the form of the corresponding dictionary variable.

However, it is best if the user follow the convention of enclosing variables in parentheses in the dictionary whenever they are to match segments, and not enclosing them in parentheses when they are to match items, even where it seems unnecessary to make the distinction, i.e., where the corresponding FLIP pattern always matches the same way. One case where difficulty might occur, otherwise, is with the mode PAV. PAV uses the original form of the dictionary variable to determine the form for the new entry that it adds to the dictionary. Thus, with the dictionary X PAV $3, X will match the same as the elementary pattern $3. However, the new entry added to the dictionary after $3 matches the segment S, will be X VAR S. This means that the pattern (X X) will not match with the list (A B C A B C). (It will match with the list (A B C (A B C)).) However, if the corresponding dictionary entry had been (X) PAV $3, then (X X) would match with (A B C A B C).

<BEGN>, <REPT>, and <CONT>

In addition to the dictionary, three elementary formats have been borrowed from CONVERT and added to FLIP. These are <BEGN>, <REPT>, and <CONT>. * They complement and are similar to the CONT and REPT modes of the dictionary.

(<BEGN> S)	this form can only be used while inside of TRANSFORM. Its value is computed by first constructing using the format S, and then transforming this using the original rule set and original dictionary, i.e., starting the TRANSFORM program all over again.
(<REPT> S R)	value is the result of first constructing using the format S, and then transforming this using the rule set R and the original dictionary. If R is an atom or an expression of the form ((LAMBDA ...) ..) the value of R is used for the rule set.
(<CONT> S R)	same as REPT except modified dictionary is used.

Each of these forms may take an extra argument whose value is treated as a starting label for the accompanying rule set. Thus (<BEGN> S 'TOP) is the same as (<BEGN> S), and (<REPT> S R 'EXIT) has the same value as constructing with the elementary format S.

* The CONVERT notation is =BEGN=, =CONT=, and =REPT= for items, and *BEGN*, *CONT*, *REPT* for fragments. Since =X and *X have a special meaning in FLIP, we use a different notation.

The value of each of these three elementary formats is added to the list being constructed as an item, unless the entire elementary format is preceded by the prefix operator "*", in which case it is added as a segment.

Example: the following FLIP function can be used to merge two lists.

```
     (MERGE (LAMBDA (X Y) (TRANSFORM (LIST X Y)
           '((((($1 ..) ($1 ..))
                     (#[1,1] #[2,1] *(<BEGN> ((#[1,2]) (#[2,2])))))
             ((NIL NIL) NIL)     ))))
```

DICTIONARY FUNCTIONS

DICTRAN

DICTRAN is a function of one argument, the dictionary. Its value is the translated version of this dictionary. If $TRAN is T, its normal setting, DICTRAN also physically modifies its input.

TRANSFORM1

TRANSFORM1 is the function that does the work for TRANSFORM. Its first argument is the list to be transformed. Its second argument is the rule set to be used in this transformation. Its third argument is a dictionary. Its fourth argument is an optional starting label in the rule set. TRANSFORM1 is useful in conjunction with the dictionary because it assumes that its dictionary is already translated, and therefore can be used to perform transformations from inside of TRANSFORM, using the modified dictionary. TRANSFORM1 is used by <BEGN>, <REPT>, and <CONT>.

ADDICT

ADDICT is a function designed for modifying dictionaries either before or after they have been translated. It can be used either to add a new entry to the dictionary, or to modify a rule set for an entry defined in mode CONT or REPT. Used in the first way, it takes two arguments. The first is the dictionary to be modified, or the name of a function whose definition contains the dictionary. (In the latter case, the function definition must be of the form (LAMBDA & (TRANSFORM & & &))). The second argument is the entry to be added, which is then placed at the end of the dictionary.
Example: (ADDICT 'FOO '(X PAV $1))

When ADDICT is used to modify rule sets in the dictionary, it may take up to five arguments. The first argument is the dictionary as described above, and the second is the modification. The third argument is the <u>dictionary variable</u> whose rule set is to be changed. This rule set is located and is given to <u>ADDRULE</u> as its first argument, with the original second argument of ADDICT as the second argument to ADDRULE. The last two arguments of ADDICT are the last two arguments of ADDRULE. Thus to add the simplification rule corresponding to ((X ↑ 2) ↑ 3) = (X ↑ 6) to SEXPT, perform

```
    (ADDICT 'DERIV '((($1 ↑ K) L) (#[1,1] ↑ =(TIMES @K @L)))
          'SEXPT 3)
```

Variables

RULES	the complete rule set given to TRANSFORM1 as its second argument.
ORIGDIC	the original dictionary, i.e., originally given to the last call to TRANSFORM1.
R	the current rule set, e.g., if three rules have already been processed, R would be (CDDDR RULES).
$D	the current dictionary - always in translated form.

NOCONS and the Dictionary

The dictionary feature itself does not require the parsing to be saved. Therefore, the user can set NOCONS to T provided he does not use any marks in his patterns or formats.

C. OTHER FEATURES

Aborting the Search

The predicate on the $ elementary pattern also allows the user to indicate to the $ function when to abandon searching. Consider the pattern

```
($[LESSP (LENGTH *) 5] $1 'A $).
```

This pattern requires that an A be found in the first six elements of the input list. If no A is found, there will be much wasted searching before the end of the list is encountered, because $ will keep extending its segment and evaluating its predicate. Thus for the list (Z X Y W V U ... C B A), $ will attempt 26 unsuccessful matches before quitting. This can be avoided by changing the first elementary pattern to

```
$[COND ((GREATERP (LENGTH *) 5) (QUOTE FAIL)) (T T)]
```

Whenever the value of the predicate on the $ is FAIL, $ immediately abandons its search and reports a failure to the previous elementary pattern. If the value of the predicate is NIL, $ continues searching as before. If the value is T, or some value other than FAIL or NIL, the match succeeds, at least as far as $ is concerned. Note that for patterns such as ($[pred] 'A $), the predicate will not be applied until an A is found, because of the fast search techniques. For cases such as this, the abort feature also works for $$, the slow $ elementary pattern.

Failure Predicate

The failure predicate extension to the $ elementary pattern allows the user additional control of the search. Basically, the failure predicate allows the user to specify under what conditions $ is to resume searching following a failure. It differs from the abort predicate in that the abort predicate operates <u>before</u> the $ matches, whereas the failure predicate operates after the $ has matched, passed control to the next elementary pattern function, and then been given back control due to a failure later in the match. The failure predicate is written as another predicate, in square brackets, immediately following the first one, e.g., $[pred][failure pred]. If the value of the failure predicate is T, the search continues, exactly the same as though there had been no failure predicate.

MARKS are handled in a special way when the failure predicate is being evaluated. Normally, a reference to a nonexistent portion of a parsing produces an error. However, the value of a MARK that refers to an elementary pattern that did not match is FAILED, if that MARK is evaluated from a failure predicate. Furthermore, the parsing that is used contains the matches of <u>all</u> successful elementary patterns, not just those that precede the $. Thus in the elementary pattern

```
    ($[][EQ #4 'FAILED] 'A $1 'B $1 'C 4)
```

the search will continue if and only if the reason for failure was that a B was not found following the A. The above pattern will <u>not</u> match the list (1 2 3 A 4 B 5 6 7 A 8 B 9 C), even though the same pattern without the failure predicate would match it.

If the $ appears inside of an EITHER, REPEAT, or SUBPATTERN, MARKs can be used in the failure predicate to reference segments or items matched at higher levels, by using the full bracket notation with ↑. If the corresponding elementary pattern did not match, the value of the MARK will again be FAILED.

Because of the fast search strategies, the $ does not pass on control until it finds a suitable point in the workspace. This means that the elementary pattern

```
    ($[][NILL] 'A $ 'B $)                                *
```

will fail if a B is not found following the first A. However, the first $ <u>will</u> search for and find an A <u>before</u> passing control to the rest of the match. The failure predicate is evaluated only when a failure occurs at some point after the second elementary pattern. Compare this with the elementary pattern

```
    ($[][NILL] $1 'A $ 'B $)
```

This latter pattern will succeed only if an A is the second element in the workspace.

---

* NILL is a function of no arguments whose value is NIL.

SIDE Conditions

The SIDE elementary pattern allows the user to exercise the same controls over the matching process as that offered by the abort and failure predicates of the $ elementary pattern. However, SIDE does not match, and aside from controlling the matching process, does not affect the final parsing. The form of the SIDE elementary pattern is

```
     (SIDE pred1 pred2)
```

The first argument to SIDE, if present, acts exactly as the abort predicate for $. It is evaluated when the SIDE elementary pattern is entered. If its value is NIL, the SIDE elementary pattern reports a failure. Otherwise, SIDE allows the match to continue. If a failure occurs subsequently, the second argument to SIDE, if present, is evaluated. MARKS are treated the same way as for the failure predicate on the $. If its value is NIL, SIDE reports failure. Otherwise, a match is again attempted.

$ will look through SIDE in the same way as it does to assignment and housekeeping functions. After $ finds a suitable point in the workspace, the SIDE elementary pattern will be entered as before.

NOT

The elementary pattern (NOT form) matches a single item provided that item is _not_ equal to the value of _form_. (NOT form) is thus equivalent to $1[NOT (EQUAL * form)], but (NOT form) will operate faster.

Reentrant Subpattern

Normally, a subpattern which matches an item is not reentrant. Thus, in the pattern ($ ($ $1 $) $ #[2,2] $), if no match is found for #[2,2], control will return to the very first elementary pattern. Using the normal subpattern, there is no way to cycle through the list matched by the subpattern before continuing to the next list, because once a subpattern successfully matches as an item, it is finished operating. However, a subpattern-item can be made reentrant by means of the prefix operator "+". Thus using the pattern ($ +($ $1 $) $ #[2,2,] $), a match would occur with the list ((A B C) (D E F) (G H I) X Y H). Similarly, the pattern (+($ $1 $) ($ #[2,2] $)) could be used to match a list of two lists whenever the two lists have a common element.

Reentrant REPEAT

REPEAT is similar to the subpattern-item with respect to the reentrant question: once REPEAT has finished matching, it is never reentered after a failure. *

* Of course, if control reverts back to an elementary pattern previous to the REPEAT, which then continues the match, such as $, REPEAT will be entered again, the same as subpattern-item. However, it is not <u>reentered</u>, i.e., allowed to continue on a match that it had started previously.

Since REPEAT also attempts to match as many times as it can, without regard to elementary patterns that follow it, it may have a different effect than intended. For example, the pattern (REPEAT[EITHER [A;B]] A $) can never match because the final A would always be included in the segment matched by REPEAT. Similarly, (REPEAT[$1] $1) will never match.

However, REPEAT [E] can be made reentrant by writing instead

```
REPEAT[E / T]
```

This REPEAT will match repetitively using the sequence of elementary patterns E. However, if a failure occurs subsequently, it will delete the last repetition, reset the workspace to the value it had before the last repetition, and try again. Essentially, REPEAT operates as before, except that after a failure, it backs up. Using this notation, the elementary patterns (REPEAT[$1 / T] $1) and (REPEAT[EITHER[A;B] / T] A $) will work correctly.

The full form for the reentrant REPEAT is

```
REPEAT[E / N1 N2 P]
```

where N1 and N2 proscribe bounds on the number of times REPEAT is to match, and P is evaluated after a failure to determine whether REPEAT is to back up. REPEAT will continue to back up until either the lower bound N1 has been reached, or until the value of P becomes NIL. MARKs that appear in P can refer forward in the parsing and are evaluated in the same way as those appearing in the failure predicates for $ and SIDE.

Variable Patterns and Formats

The elementary pattern :X was defined to mean that X is evaluated and then treated as a subpattern, i.e. a list of elementary patterns. This definition is extended to allow the value of X to be an arbitrary elementary pattern. Thus it is permissible for X to evaluate to $1, or EITHER [$1 'A;], as well as a list of elementary patterns, which is itself an elementary pattern, namely the subpattern. The : operator simply evaluates its argument, translates it, and then attaches the resulting functions to the list of elementary pattern function calls. It is thus exactly the same as the PAT mode in the dictionary, and in fact, the translation of the two are indistinguishable.

As in the case of the PAT mode, $ is "smart" enought to evaluate the argument of the : operator, translate it, and continue searching whenever it encounters this type of elementary pattern. Thus the pattern ($:X $), where X has the value A, will match essentially as fast as ($ 'A $).

In the case of the elementary pattern *:X, X may also be an arbitrary elementary pattern, which then matches a segment. However, since EITHER, REPEAT, and $ always match segments, the presence or absence of "*" will not affect their operation. Similarly, if X evaluates to $1, :X and *:X have the exact same effect, because $1 must match an item. The prefix operator "*" only makes a difference when the elementary pattern can match <u>either</u> a segment on an item, i.e., for the two elementary patterns VAR and SUBPATTERN. Since this determination is made at <u>translation</u> time, i.e., the first time the pattern is used, one cannot use both :X and *:X.

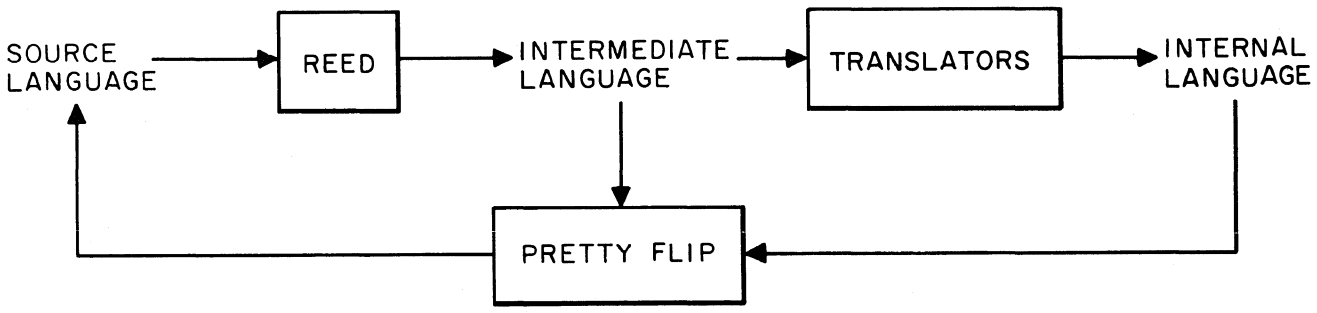
An analogous extension has been made for variable formats.

Atomic Patterns and Formats

MATCH has been extended to accept single elementary patterns of the VAR, $N, $, and NOT types, as well as a list of elementary patterns. No parsing is produced, but * will be bound to the correct expression when CONSTRUCT is called with this parsing. As a result of this extension, it is also possible to use MATCH on atomic workspaces, for example, MATCH(1 $1[NUMBERP *]). A similar extension is allowed for formats.

APPENDICES

The three appendices below describe and summarize the three levels of FLIP expressions. The first appendix treats the external syntax of FLIP expressions, sometimes called source language. This represents the outermost level. The second appendix describes the operation of REED which converts source language to intermediate language, which is the form of FLIP expressions before they are translated. The third appendix discusses the representation of FLIP entities after translation, the innermost level. PRETTYFLIP converts (by printing) expressions of level two or level three to level one. Thus, we have the following relationship:



Appendix 1: Source Language

This appendix summarizes the FLIP syntax. Individual FLIP expressions should be separated by at least one space or a carriage return. Although the syntax is divided into expressions used in MATCH and CONSTRUCT, the REED program makes no distinction. The expression #[1,2] is read the same whether it is to be used in a pattern or format, although the two interpretations are different.

The following conventions are used below:

> A is an atom; L is a list; X, Y, Z, X1 ... XN are arbitrary LISP *forms*;
>
> [x] and [y] mean that (x) and (y) are LISP forms;
>
> E, E1 ... EN are sequences of elementary patterns; F is an elementary format, R is a ruleset.

MATCH

$	
..	same as $, no predicates allowed
$$	$ without search strategies
...	same as $$, no predicates allowed
$[x]	$ with predicate
$$[x]	$$ with predicate
$[x][y]	$ with failure predicate
$$[x][y]	$$ with failure predicate
$X	matches segment of length X
$X[y]	$X with predicate

=X	VAR, item type
'X	same as =(QUOTE X); X can be an arbitrary LISP <u>expression</u>, i.e., (A . B) is allowed.
A	if A is in dictionary, same as @A; otherwise same as 'A, =A, or @A depending on setting of DEFAULT and definition of PDEFAULT.
=X[y]	VAR with predicate
*=X	VAR, segment type
*'X	same as *=(QUOTE X)
*A	if A is in dictionary, same as @A; otherwise same as *'A or *=A, depending on setting of DEFAULT and definition of PDEFAULT.
*=X[y]	VAR with predicate
#X	MARK
#[X1,X2,..Xn]	MARK
#[↑,X1,X2,..Xn]	MARK, from top level
#X[y]	MARK with predicate
#[X1,X2..Xn][y]	MARK with predicate
#[↑,X1,X2,..Xn	MARK from top level with predicate
@X	Dictionary variable
L	Subpattern-item
*L	Subpattern-segment
+L	reentrant subpattern
:X	variable pattern-item
*:X	variable pattern-segment

EITHER[E1,E2,..En]	EITHER
REPEAT[E]	REPEAT
REPEAT[E / T]	Reentrant REPEAT
REPEAT[E / X]	match at least X times
REPEAT[E / X Y]	but not more than Y times
REPEAT[E / X Y Z]	Reentrant REPEAT
(SET A Y)	assigns A to value of Y
A←Y	where Y is an elementary pattern, same as Y followed by (SET A #-1). U←V←X←Y is allowed.
(NOT X)	$1[NOT (EQUAL * X)]
(SIDE X Y)	Side condition

CONSTRUCT	
=X	VARF, Item type
'X	same as =(QUOTE X), X can be an arbitrary expression
A	if A is in dictionary same as @A; otherwise same as 'A,=A, or @A, depending on DEFAULT and definition of PDEFAULT.
*=X	VAR, segment type
*'X	same as *=(QUOTE X)
*A	if A is in dictionary same as @A, otherwise same as *'A, *=A, or @A, depending on setting of DEFAULT and definition of PDEFAULT.
#X	MARK
#[X1,X2..Xn]	MARK
#[↑ X1,X2,..Xn]	MARK, from top level
..	same as #1 unless it is last in a format, subformat, EITHERF, or REPEATF format, in which case it is same as #-1
L	Subformat-item
*L	Subformat-segment
:X	variable format-item
*:X	variable format-segment
EITHER[E1;E2..En / X]	EITHER elementary format
EITHER[E1;E2;..En]	EITHER elementary format, search for corresponding EITHER parsing automatically performed
REPEAT[E / X]	REPEAT elementary format

REPEAT[E]	REPEAT elementary format, search for corresponding REPEAT parsing automatically performed
(SET A Y)	assigns A to value of Y
A←Y	where Y is an elementary format, assigns A to the value of Y.
@X	Dictionary variable
(<BEGN> F)	<BEGN> elementary format
(<BEGN> F X)	<BEGN> with GOTO label
(<CONT> F R)	<CONT> elementary format
(<CONT> F R X)	CONT with GOTO label
(<REPT> F R)	REPT elementary format
(<REPT> F R X)	REPT with GOTO label
*(<BEGN> F)	<BEGN>-segment
*(<BEGN> F X)	<BEGN>-segment with GOTO label
*(<CONT> F R)	<CONT>-segment
*(<CONT> F R X)	<CONT>-segment with GOTO label
*(<REPT> F R)	<REPT>-segment
*(<REPT> F R X)	<REPT>-segment with GOTO label

The dictionary syntax is summarized on pages 82-83.

Appendix 2: Intermediate Language

This appendix describes the intermediate language, the form of FLIP expressions before they are translated. The transformation from source language to intermediate language is performed by the FLIP read function, REED. The intermediate language representation is designed for convenience of translation: separate FLIP expressions correspond to separate LISP expressions, with prefix operators attached at the front of the corresponding LISP expressions, and predicates attached at the back. For example, the intermediate language representation for $[GREATERP (LENGTH *) 5] is ($ (GREATERP (LENGTH *) 5)); the representation of *=(CAR X) is (* = CAR X). The intermediate language is presented by describing the operation of REED.

REED operates in the same way as READ (except for control characters, see pages 58-60), until it encounters one of the characters ',*,?,:,+,=,@,#,$,←,[, or either of the two atoms REPEAT or EITHER. The action taken for each of these cases is described below. X, X1 ... Xn denotes an expression read by REED. Thus 'X is read as (QUOTE X) means that '''FOO is read as (QUOTE (QUOTE (QUOTE FOO))). However, if a space carriage return,), [,], ; or / follows the character in question, no special action is taken, e.g.,'' is read as (QUOTE ').

'X	(QUOTE X)
*X	(* . X), i.e., (CONS (QUOTE *) (REED))
?X	(? . X)
:X	(: . X)
+X	(+ . X)
=X	(= X)
@X	(DICTIONARY X)

#X	(MARK (X)), e.g., #1 is (MARK (1))
#[X1,X2,... Xn]	(MARK (X1 X2 ... Xn)), e.g., #[↑,1,2] is (MARK (↑ 1 2))
$	($)
$X	($N X), e.g., $1 is ($N 1); $$ is ($N $) but the translators make a special check for this
X1←X2	read as two LISP expressions, (SET X1) and X2. Spaces between X1, ←, and X2 are ignored, i.e., X←Y is the same as X ← Y.
[	treated the same as with READ, i.e., REED continues reading until it finds a matching) or]. After REED completes reading this expression, it attaches the expression at the end of the previously read expression. Thus, $1[MEMB * #2] is read as two expressions ($N 1) and (MEMB * (MARK (2))), and then the second expression is attached at the end of the first giving ($N 1 (MEMB * (MARK (2)))). Spaces before "[" are ignored. If there is no place to attach the expression, no action is taken. For example, '[A B C] is read as (QUOTE (A B C)), because there is no place to put (A B C). If the previous expression is atomic, the setting of DEFAULT is used. For example, if DEFAULT is =, A[EQUAL #2 #3] is read as (= A (EQUAL (MARK (2)) (MARK (3)))). Using a predicate with an atom, however, is very rarely necessary.

EITHER, REPEAT

If the next character read, excluding spaces and carriage returns, is [, it is treated as (((and an expression is read using REED. During the reading of this expression, ; is interpreted as)(, and / as)). EITHER or REPEAT is then CONSed onto the resulting expression.

Examples:

```
EITHER[A; B]       (EITHER ((A)  (B)))

REPEAT[$1 / 3 4]    (REPEAT (($N 1)) 3 4)
```

Spaces before or after ; or / are optional. If next character is not [, EITHER or REPEAT is read as any other atom.

For all other FLIP expressions, the intermediate representation is the same as the LISP representation, e.g., (SIDE NIL (EQUAL #3 'FAILED)) is read and represented as (SIDE NIL (EQUAL (MARK (3)) (QUOTE FAILED))).

Appendix 3: Internal Language

The translated form of patterns, formats, and dictionaries consists of sequences of function calls which are evaluated at the proper time. The task of the translating functions, PATTRAN, EITHERTRAN, FORMTRAN, and DICTRAN, is to convert FLIP expressions to their corresponding function calls. Rather than describe these transformations by listing again all of the FLIP entities of Appendix 1 and their translated forms, the corresponding FLIP functions are listed below with an explanation of their arguments. From this, the individual translations may be inferred. For example, given that the function $ has two arguments, an abort predicate and a failure predicate, then the translation of $ is ($), of $[x] is ($ (x)), and the translation of $[x][y] is ($ (x) (y)).

Translations are made permanent by altering the input list structure whenever variable $TRAN is set to T, its normal setting.

PATTRAN

The basic difference between the translation of elementary patterns and elementary formats is that the arguments of elementary format functions are always evaluated, while the decision as to when and if an argument is to be evaluated for an elementary pattern function is left to the function itself. (This is necessary because predicates cannot be evaluated until an elementary pattern has tentatively matched. Also, failure predicates cannot be evaluated until the rest of the match is tried.)

There are twelve elementary pattern functions: $N, VAR, NOTT, SIDE, PATTERN, PAT, SET1, $, SLOW$, EITHER, REPEAT, and DICVAR. The arguments to the elementary pattern functions specify how it is to match. PATTERN, EITHER, and REPEAT all require special housekeeping functions after they match. Every other elementary pattern translates into one, and only one, function call.

<u>$N</u> is a function of two arguments, N and PRED. The value of N is the length of segment matched by $N. PRED is the predicate, if any.

<u>VAR</u> is a function of three arguments, V, SEG, and NPRED. The value of V is matched by VAR as an item if SEG is NIL, otherwise as a segment. (SEG is T if V is a MARK.) NPRED is the predicate.

<u>NOTT</u> is a function of one argument, X. NOTT matches provided (CAR WS) is not equal to the value of X.

<u>SIDE</u> is a function of two arguments, PRED1 and PRED2. The first predicate, if present, is applied before matching. The second predicate, if present, is applied after a failure.
(SIDE NIL NIL) is effectively a NOP.

<u>PATTERN</u> is a function of two arguments, PATTLIST and SEG. PATTLIST is given to PATTRAN to be translated and is matched as an item if SEG is NIL, as a reentrant item if SEG is T, and as a segment if SEG is any other value.

<u>PAT</u> is a function of two arguments, XPAT and *. The value of XPAT is translated as an elementary pattern. * is used by PATTRAN for the translation. PAT is the function corresponding to elementary patterns that use the : operator. It differs from PATTERN in that XPAT can be any elementary pattern, <u>including</u> a subpattern.

<u>SET1</u> is a function of two arguments, NAME and V. NAME is set to the value of V. If V is identical to NAME, the segment-item last matched is used for V, as in the case of elementary patterns such as FOO←$1, which translates as (($N 1) (SET1 FOO FOO)).

<u>$</u> is a function of two arguments, PRED1 and PRED2, the two predicates on the $ elementary pattern.

<u>SLOW$</u> is a function of two arguments, PRED1 and PRED2, the two predicates on the $$ elementary pattern.

EITHER is a function of one argument, $OR. $OR consists of a list of patterns, and is translated by EITHERTRAN, as described below.

REPEAT is a function of four arguments, $RPT, N1, N2, and PRED. $RPT is a list of elementary patterns, and is translated by PATTRAN. A call to a special function REPEAT1 is attached at the end of the translation. The value of N1, if present, is the lower bound on the number of repetitions. The value of N2, if present, is the upper bound. PRED is a predicate used to decide whether or not the REPEAT is to be reentered.

DICVAR is a function of one argument, X, the name of a dictionary variable.

The translation of a pattern consists of the list of function calls corresponding to the elementary patterns, and is headed by the rejection information, described on pages 43-45.

Example: The translation of

```
     ($ $2 A ($ $1) $[][NILL] *=(GET 'FOO #[4,2])$) is

          ((NIL NIL . A)
           ($)
           ($N 2)
           (VAR A NIL NIL)
           (PATTERN ($ $1))
           (PATTERN1 NIL NIL NIL NIL)
           ($ NIL (NILL))
           (VAR (GET (QUOTE FOO) (MARK (4 2))) SEGMENT NIL)
           ($))
```

PATTERN1 is the housekeeping function associated with PATTERN and EITHER. Its arguments are set by PATTERN, or EITHER, by actually changing the expression (PATTERN1 NIL NIL NIL NIL) in the translation. Since the arguments to PATTERN1 contain the complete parsing, the translation of a pattern may appear very complicated after it has been run a few times.

The translation is kept in pointer-pair format: CAR of the translation is the list of function calls and rejection information; CDR is a pointer to the tail of this list.

EITHERTRAN

The argument of EITHERTRAN is a list of patterns corresponding to the alternatives for an EITHER elementary pattern. EITHERTRAN translates this list for subsequent use by the elementary pattern function EITHER. This translation consists of translating in turn each of the patterns, making a list of all of their translations, and heading this list by special rejection information for EITHER. This rejection information consists of a list of the rejection information for each individual pattern, provided the rejection information indicated that the corresponding pattern began with a VAR. If each of the individual patterns fell into this category, the EITHER rejection information is headed by T, otherwise by NIL. This rejection information is primarily for use by the fast $. (see page 52)

Example: the translation of the elementary pattern

```
     EITHER['A $2 ; 'C $] is

     (EITHER (((QUOTE A) ($N 2))
              (QUOTE C) ($)))
```

The translation of the argument to EITHER is

```
     ((T (NIL QUOTE A) (NIL QUOTE C))
      ((T NIL QUOTE A) (VAR (QUOTE A) NIL NIL) ($N 2))
      ((T NIL QUOTE C) (VAR (QUOTE C) NIL NIL) ($)))
```

except that the translation of the two patterns is in pointer-pair format.

FORMTRAN

There are nine elementary format functions: VARF, FORMAT, SETF, EITHERF, REPEATF, <BEGN>, <CONT>, <REPT>, and DICVARF. Each elementary format translates into one and only one function call.

VARF is a function of two arguments, X and SEG. X is attached to the list being constructed as an item if SEG is NIL or if X is an atom. Otherwise, X is attached as a segment.

FORMAT is a function of two arguments, X and SEG. X is used to construct a list which is then attached as an item if SEG is NIL, otherwise as a segment. Both subformat and variable format translate as calls to FORMAT.

SETF is a function of two arguments, NAME and V. NAME is set to V. If V is identical to NAME, NAME is set to the value of the next elementary format function, e.g., the translation of FOO←#3 is

```
((SETF (QUOTE FOO) (QUOTE FOO)) (VARF (MARK (3)) T)).
```

EITHERF is a function of two arguments, X and Y. Y specifies the corresponding EITHER elementary pattern. If Y is NIL, EITHERF searches the parsing as described in the text. X is a list of formats used for constructing.

REPEATF is a function of two argument, X and Y. Y specifies the corresponding REPEAT elementary format, or a number. X is the format used in constructing.

<BEGN> is a function of three arguments, X, SEG, and W. X is a format for constructing, SEG specifies whether the result is to be attached as item or segment, and W is an optional GOTO label. <REPT> and <CONT> are similar except their third argument is a rule set and W is the fourth argument.

DICVARF is a function of one argument, X, the name of a dictionary variable.

DICTRAN

The translation of a dictionary is of a list of dotted pairs, consisting of the name of a dictionary variable and the form which is its translation. The translation of modes VAR, PAT, EXPR, and SKEL is straightforward into the corresponding functions VAR, PAT, VARF, and FORMAT. Modes CONT and REPT translate to functions of the same name that are very similar to <CONT> and <REPT>. The translation of modes UAR, PAV, BUV, and CUV involves a special function DICVAR1.

DICVAR1 is a function of three arguments, X, Y, and Z. X is the value of the dictionary variable. This is the argument that is changed when dictionary variables with these modes match as elementary patterns. The second argument of DICVAR1 is (T) if the dictionary variable was enclosed in parentheses, otherwise it is NIL. The third argument is the list of function calls for matching. For PAV, BUV, and CUV, this list consists of a call to PAT followed by a call to the function SETD. For VAR, it consists of a call to either $N or $, followed by a call to SETD. SETD is the function that modifies the dictionary.

The first argument to SETD is the mode of the variable. The second a pointer to its translated definition so that SETD does not have to look it up again. Thus, the translation of any dictionary containing mode, VAR, PAV, BUV, or CUV is always circular.

BIBLIOGRAPHY

1. Berkeley, E. C. and Bobrow, D. G. "The Programming Language LISP: Its Operation and Applications," Information International Inc., Cambridge, Massachusetts, 1964.

2. Bobrow, D. G. "METEOR: A LISP Interpreter for String Transformations," in Berkeley and Bobrow (see above).

3. Bobrow, D. G. and Murphy, D. L. "The Structure of a LISP System Using Two-Level Storage", BBN Report No. 1467. Comm. ACM (in press).

4. Bobrow, D. G. and Teitelman, W. "Format-Directed List Processing in LISP", AFCRL-66-302, BBN Report No. 1366, April 1966.

5. Bobrow, D. G. and Weizenbaum, J. "List Processing and the Extension of Language Facility by Embedding", Trans. IEEE PGEC, August 1964.

6. Bobrow, D. G. *et al.*, "The BBN-LISP System", BBN Scientific Report No. 1, October 1966.

7. Guzman, A. and McIntosh, H. V. "CONVERT," Comm. ACM, Vol. 19, No. 8, August 1966, pp 604-615.

~~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
DESIGN AND IMPLEMENTATION OF FLIP, A LISP FORMAT DIRECTED LIST PROCESSO

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Scientific. Interim.

5. AUTHOR(S) *(First name, middle initial, last name)*
Warren Teitelman

6. REPORT DATE 15 July 1967	7a. TOTAL NO. OF PAGES 113	7b. NO. OF REFS 7

8a. CONTRACT OR GRANT NO. AF-19(628)-5065 ARPA Order No. 627, Amendment No. 2 b. PROJECT NO. 8668 c. DoD Element 6154501R d. DoD Subelement n/a	9a. ORIGINATOR'S REPORT NUMBER(S) BBN Report No.1495 Scientific Report No. 10 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* AFCRL-67-0514

10. DISTRIBUTION STATEMENT
1- Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

11. SUPPLEMENTARY NOTES This research was sponsored by the Advanced Projects Agency	12. SPONSORING MILITARY ACTIVITY Air Force Cambridge Research Laboratories (CRB) L. G. Hanscom Field Bedford, Massachusetts 01730

13. ABSTRACT
This paper discusses some of the considerations involved in designing and implementing a pattern matching or "COMIT" feature inside of LISP.[1],[2] The programming language FLIP is presented here as a paradigm for such a feature. The design and implementation of FLIP discussed below emphasizes compact notation and efficiency of operation. In addition, FLIP is a modular language and can be readily extended and generalized to include features found in other pattern driven languages such as CONVERT [6] and SNOBOL. This makes it extremely versatile.

The development of this paper proceeds from abstract considerations to specific details. The syntax and semantics of FLIP are presented first, followed by a discussion of the implementation with especial attention devoted to techniques used for reducing the number of conses required as well as improving search strategy. Finally FLIP is treated as a working system and viewed from the user's standpoint. Here we present some of the additions and extensions to FLIP that have evolved out of almost two years of experimentation. These transform it from a notational system into a practical and useful programming system.

DD FORM 1 NOV 65 1473 (PAGE 1)
S/N 0101-807-6811

Unclassified
Security Classification

A-31408

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
LISP						
List Processing Language						
Symbol Manipulating Language						
String Transformations						
Pattern-Driven Language						
Format Directed List Processing						