619

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Project MAC

Artificial Intelligence Memo 87 | Memorandum MAC-M-263

FLIP - A FORMAT LIST PROCESSOR

by
Warren Teitelman

0.0 Summary

This memo describes a notation and a programming language for expressing, from within a LISP system, string manipulations such as those performed in COMIT. The COMIT formalism has been extended in several ways: the patterns (the left-half constituents of COMIT terminology) can be variable names or the results of computation; predicates can be associated with these elementary patterns allowing more precise specification of the segments they match; the names of elementary patterns themselves may be variable or the results of computation; it is no longer necessary to restrict the operations to a linear string of characters (or words) since elementary patterns can themselves match structures; etc. Similar generalizations exist for formats, i.e. what corresponds to the right-half of the COMIT rule.

The language has been implemented as a collection of LISP functions. Used as a single subroutine, the resulting system permits a marriage of LISP and COMIT-type notation. It is designed so that it can be used by the novice with little or no information about the details of its operation. However, the modularization of the system also allows the sophisticated user to exercise partial or complete control over the way in which it operates. In addition, he can easily expand it by defining new LISP functions.

Limited comparisons indicate a LISP compiled function will run only about twice as fast as a FLIP statement to perform the same task. Much time is saved by writing and debugging in FLIP, instead of coding directly in LISP. In addition, FLIP rules are usually more economical in space.

Mac Memo 264 describes a LISP Edit function that uses FLIP to permit editing while inside of a LISP system.

1.0 Introduction

LISP is a very powerful symbol-manipulating programming language. However, there are certain types of problems for which the explicit function-oriented nature of LISP requires lengthy coding which is difficult to prepare and to understand when read. (-1-)

These problems have been described under various headings: search procedures, parsing, string manipulation, etc. Basically, these problems require locating certain substructures in a larger structure, either to ascertain their presence, to find their value, or, as is more usual, to utilize them in assembling other structures.

"Such transformations may be characterized (and caricaturized) by the following instructions for a transformation:

> Find in this string the substring consisting of the three elements immediately preceding the first occurrence of an A, and find the element just before an occurrence of a B which follows these three elements; if such elements exist, exchange the position of the three elements and the one element, delete the A, and replace the B by a C." (-2-)

The LISP formalism cannot easily express such processes, although each can be individually programmed (if only because LISP is universal).

The first attempt to deal with this type of problem in any systematic way arose in the use of computers in linguistics. This resulted in the development of the programming language COMIT, which is built around a notation for expressing transformations such as the one above. It consists of a formal method for selecting substrings from a string, and then indicating the structure of the transformed string. As an illustration, the COMIT rule for the transformation above would be:

```
     $ + $3 + A + $ + $1 + B + $ = 1 + 5 + 4 + 2 + C + 7
```

****

(-1-) Section 2.2 contains an excellent example of such a function. It performs the transformation described in the inset on this page.

(-2-) Bobrow, D. G. "METEOR: A LISP Interpreter for String Transformation," The Programming Language LISP: Its Operations and Application, P. 161.

However, COMIT is designed to operate on strings of words, i.e. linear lists, and suffers badly in dealing with more complicated structures. For example, there is no way to realize groupings such as those indicated by parentheses in LISP except by using special subroutines to separate out substrings with balanced parentheses (and subroutine linkage is particularly cumbersome). Since working in COMIT also precludes the use of the flexibility and recursive power of LISP, most users prefer to forego the convenience of the COMIT notation and work entirely in LISP when they have a problem which requires some of the powers of both.

Because this latter contingency is arising more and more frequently, especially in conjunction with problem solving systems which deal with natural language input, a language in which statements in both LISP functional notation and COMIT prototype notation are interpretable is becoming increasingly desirable.

Bobrow attempted to resolve this problem in conjunction with STUDENT (-3-) by embedding COMIT in LISP via METEOR, a LISP program which interprets COMIT-type rules. (-4-)

****

(3) D. G. Bobrow, Natural Language Input for a Computer Problem Solving-System M.I.T. Project MAC, MAC-TR-1.

(4) It is unfair to dismiss METEOR as a mere translation of COMIT formalism into LISP. Many of the features of FLIP were anticipated by METEOR, and similarly, its deficits and shortcomings suggested some of the more sophisticated some of the powers of FLIP. However, it suffers from a defect common to such first generation efforts - an overdose of ad-hocness. But for this and a lack of flexibility (METEOR was not designed for easy modifications or introducing new conventions), METEOR might well possess all of the properties that motivated the design and implementation of FLIP. As it is, it did provide a starting-point for its design and a yardstick for evaluating it.

There have also been several attempts involving the design of new languages. These include the work of McIntosh (-5-), and the proposed LISP II matching feature (-6-). As a rule, these languages are all subject to one or more of the following criticisms.

First, the formalisms are rigid and cannot easily be modified or expanded. Usually, there are certain basic primitives which may not be tampered with. In addition, none of these languages allow the user to specify arbitrary components of a pattern as the result of a computation performed while the matching process is taking place.

Secondly, these languages are usually interpretive, and accordingly slow. They therefore have little or no appeal to the sophisicated programmer who feels that he sacrifices more in efficiency and running speed than he gains from ease of writing.

Finally, it is difficult for the user to affect the internal process and exercise control over the manner in which the matching procedure is carried out.

The formalism and implementation of FLIP reflect a concern for these problems. The aim was to develop a facility within LISP which allowed the user to specify sophisticated search/parsing procedures in a way that was both flexible and fast, and which could be used without detailed knowledge of the program. The result of embedding FLIP inside of LISP is not only an increase in the scope of LISP, but it also permits FLIP to make use of the full powers of LISP.

****

(-5-) CONVERT, H.V.McIntosh, Instituto Politecnico Nacional, Centro de Investigation y de Estudios Avanzados. This is a collection of LISP functions similar in intent to FLIP.

(-6-) Bobrow, D.G. "The COMIT Feature in LISP II," Mac Memo 219. This is a proposal for a COMIT-type feature within LISP II, and contains some of the generalizations of METEOR found in FLIP.

In the current FLIP system, the novice user can remain almost totally ignorant of the nuances of its construction and operation, and still utilize many of its more sophisticated features. This is because all of the basic, more commonly used features have been standardized and made notationally simple.

At the same time, there exist many variations which provide the sophisticated programmer with more flexibility and power. There are also facilities for allowing him to control the operation of the system. By defining new operations, he can augment the language. By monitoring the execution of existing ones, he can increase the efficiency of a FLIP program as much as he desires up to and including writing portions of it in machine language.

Both the neophyte and sophisticate will realize the benefits derived from the fact that instructions to FLIP specify goals (i.e. the segments of the list structure that one wishes to locate), as opposed to conventional instructions to computers, which specify methods. This makes it much easier to detect logical errors in the program (especially when the selective trace feature is used.) The result is that the time required to construct an unfamiliar or logically complicated algorithm may still be long, but mostly for the planning steps, not for preparing the program.

The FLIP system in its current state of evolution is far from the final word in format processing languages. It includes many features we have found desirable or necessary, and is certainly developed to the point where it is useful as it stands. But it is by no means presented as a finished product, and its design is intended to encourage modification and expansion. What the user should bear in mind while reading this paper is that FLIP is intended to be a notational base and a programming system from which one can build more interesting, sophisticated, and useful functions. The operations built into FLIP are examples of what can be done, not necessarily what should be done.

The basic operation of FLIP consists of a matching process and a construction process. These correspond to calls to two functions, MATCH and CONSTRUCT, specifying for the former, a list structure and a pattern, and for the latter, a match and a format. Both the pattern and the format are assumed to be translated into internal representations used by MATCH and CONSTRUCT. Two functions are provided for this translation process, PATTRAN and FORMTRAN. Finally a top level function, FLIP, is provided which given an input list, a pattern, and a format, performs the necessary translations, matching, and constructing, and returns the value of both the match and construct. We discuss these in greater detail below.

2. The Match

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

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

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

If our input list is (A B C D) and the pattern is ($ B $), a match occurs with the $ matching [A], B matching [B], and the final $ matching [C D]. (-7-) If we did not have the final $ in the pattern, there would be no match because there is no way for the segments matched by the first $ and the B alone to make up the entire list. If the pattern were ($ D) however, we could have a match with [A B C] [D] the corresponding segments. Note that if the pattern were ($ D $), the parsing would be [A B C] [D] [], the last $ matching the null segment [].

Another example: let (A B C D E F G) be the input list and ($ $1 D $) the pattern. In this case [A B] [C] [D] [E F G] is the parsing, with $ matching the first segment, $1 the second, etc. If we were to return to the example in the introduction, where the pattern was ($ $3 A $ $1 B $), and use 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]. Note that the A pattern did not match the first A, because the $3 pattern must first find a segment of length 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.

****

(-7-) We will use the notation [..] to denote a segment of a list. The internal representation of these segments is not important here, but will be discussed in section 2.1.2.

These are some simple examples. Section 2.1, which defines the elementary patterns and discusses them in detail, will introduce some more complicated examples in the course of the exposition; Section 2.2 contains some comparisons between FLIP and LISP, including an extremely sophisticated (and complicated) example showing the power of FLIP. The remainder of section 2 contains the translation from source language into the internal representation of patterns, in 2.3; the various modes of operation, in 2.4; and finally, in 2.5, a description of the operation and purpose of each of the functions used by the match, and the method of defining new elementary patterns.

2.1 Elementary Patterns

2.1.1 An Introduction

Before proceeding to the formal treatment of the elementary patterns, we will try to give the prospective user the flavor of the match and type of patterns it uses. This will of necessity be only an incomplete introduction.

We have already introduced three of the patterns in FLIP's repertoire, i.e., the three COMIT patterns $, $n, and x. Nothing more needs to be said about $ or $n at this point except that they will be considerably extended in section 2.1.2. For example, the user will be allowed to specify n as a variable or the result of a computation, and will also be allowed to associate predicates with these patterns which the segments they match must then satisfy.

The "x" pattern in the previous section is an instance of a more general pattern. Above, x matched something equal to itself. It is sometimes useful to have x match something equal to the value of itself, i.e. to treat x as a variable. For example, if the value of x is (A B C), then if WS is (W X Y Z (A B C) D) and PATT is ($ X $ (= X) $), (-8-) the parsing is [W] [X] [Y Z] [(A B C)] [D].

****

(-8-) WS and PATT are the names of the arguments of the function MATCH. WS stands for workspace, a term inherited from COMIT, and is the list being matched. PATT is the pattern.

When just "x" appears in PATT, it is called an <sexp>, for S-expression. When one writes (= x), this is called a <form>. One can also indicate the result of a computation, by writing (= <fn> <var> ... <var>). This is also a <form>. For example, if PATT were ($ X $ (= CAR (= X)) $), and X were (A B C) as before, the result of the match would be [W] [X] [Y Z] [A] [B C D]. Remember: the value of an <sexp> is always itself; the value of a <form> is the result of a LISP computation. For example, the value of A is equal to the value of (= CAR (A B C)).

Both an <sexp> and a <form> are examples of something called a <var>. When a <var> appears by itself in PATT, it represents an elementary pattern called a <varp> (for variable pattern), and it matches something equal to the value of the <var>. If the <var> is an <sexp>, this is the <sexp> itself. If the <var> is a <form>, it is the value resulting from a computation.

In the examples above, "matching something equal to itself" was taken to mean matching a segment of length 1 consisting of a single item equal to the value of the <var>. If the value of X is (A B C), WS is (X A B C D (A B C) (B C D) X Y Z), and PATT is ($ (= X) $), then the <varp> (which in this case is a <form>) matches the (A B C), i.e. the parsing is [X A B C D] [(A B C)] [(B C D) X Y Z]. In this case, the value of the <var> is treated as a single item.

It is also possible to make FLIP treat a <var> as a segment. This is done by using the special symbol "**". Then if WS were (X A B C D (A B C) (B C D) X Y Z), and PATT ($ (** (= X)) $), the parsing produced would be [X] [A B C] [D (A B C) (B C D) X Y Z].

In general a <varp> carries with it the specification of whether the value of the <var> is to be treated as a segment or item, with ** denoting segment and * item. However, FLIP always treats an <sexp> and a <form> as items unless otherwise specified, so that, for example (= X) is the same as (* (= X)).

The reader may have noted that the definition of the <dol> elementary pattern is not complete (<dol> is the name of the elementary pattern denoted by $). For example, suppose WS is (A B C D C D E) and PATT is ($ C $). According to the above definition both [A B] [C] [D C D E] and [A B C D] [C] [D E] are possible matches. This ambiguity will not arise in practice because the operation of the match is a sequential, left-to-right process. Thus the first match will be the one that is found.

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

This latter conceptualization is not without its rewards. Since the match does proceed from left to right, at the time any particular elementary pattern is operating, all the elementary patterns to the left of its position in PATT have already been matched with acceptable segments (or this pattern would never have been reached), and these can therefore be referenced. This is done by the use of a <mark>. A <mark> refers to a particular elementary pattern in PATT, and the value of a <mark> is the value of that pattern, i.e. the segment that it matches.

A <mark> is also a <var> and thus a <mark> appearing by itself is a <varp>. This means that it matches something equal to its value. In the case of a <mark>, unless otherwise specified FLIP treats it as a segment, contrary to <form>'s and <sexp>'s so that (** <mark>) is the same as <mark> by itself. This is usually the desired interpretation. For example, suppose we wish to find the first repeated item in WS. The pattern we would use would be ($ $1 $ 2 $). "2" is a <mark>. It refers to the value of the second elementary pattern (i.e. the segment the $1 matches). Let us trace the operation of the match for WS (A B C D E B F).

Initially, <dol> (the $ pattern) matches [], and $1 matches [A]. The second <dol> also matches [] and the <mark> tries to match with [B]. The value of the <mark> is [A], because [A] is the segment matched by the $1. Since [A] does not equal [B], the match fails. Then second <dol> matches with [B]. Now the <mark> tries to match with [C]. Again the match fails. This continues until the second <dol> matches with [B C D E B]. <mark> tries to match with [F] and fails. <dol> matches [B C D E B F], and <mark> fails again (this time because there is nothing left in WS), and now <dol> also fails. What happens then is that the first <dol>, which matched [] initially now matches [A], the <doln> (the $n elementary pattern) then matches [B], and the second <dol> again starts by matching []. Now the value of <mark> is [B]. <mark> tries to match with [C], but since [B] does not equal [C] it does not succeed. This continues until the second <dol> is matching [C D E]. At this point, <mark> tries to match with [B] and succeeds, and the final parsing is [A] [B] [C D E] [B] [F].

Similarly if we used PATT ($ $2 $ 2 $) and WS (A B C A C B A C D), the <mark> would ultimately match [A C], and the parsing would be [A B C] [A C] [B] [A C] [D]. Note that the second segment, the one corresponding to <doln>, is equal to the one corresponding to the <mark>.

This latter example demonstrates why it is desirable to have FLIP treat <mark>'s as segments. Here the value of the segment matched by the <doln> was [A B]. If this had been treated as an item, <mark> would have attempted to match with the segment [(A B)], (because [A B] is represented internally as (A B)) and not the segment [A B].

For example, if WS were (A B C D E A B E F (A B) (B D)) and PATT ($ $2 $ 2 $) a match would occur with $2 matching [A B] and 2 matching [A B], i.e., the parsing would be [] [A B] [C D E] [A B] [E F (A B) (B D)]. However, if 2 were treated as an item, i.e. if we wrote ($ $2 $ (* 2) $) (recall that we use "*" to denote single items), then $2 would still match [A B] but (* 2) would match [(A B)], i.e. the segment consisting of the single item (A B). The parsing would be [] [A B] [C D E A B E F] [(A B)] [(B D)]. This will become clearer when we discuss the internal representation of segments and the evaluation of <mark>'s.

Since the segment interpretation of <mark>'s is usually intended, this is what FLIP assumes to be the case unless specified otherwise. This philosophy has prevailed throughout the development of FLIP. We provide a way of expressing the intention of the user precisely (here by using either * or **), and furthermore, have an abbreviated representation for the more common usage.

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

As an example, let WS be (A (B C) D (B E F) G) and PATT ($ ($ F $) $), then a match will occur with the first <dol> matching [A (B C) D], the <pattern> matching [(B E F)], because (B E F) matches ($ F $), and the last <dol> matching [G]. Furthermore, the internal representation of [(B E F)] reflects the fact that it has been matched by a <pattern>, and also includes the parsing with respect to that pattern. One can visualize the entire parsing as [A (B C) D] [[B E] [F] []] [G]. The segment that the <pattern> matches is still [(B E F)], and that is the value of <pattern> if a <mark> refers to it, but the parsing is available so that <mark>'s can also refer to segments corresponding to elementary patterns contained inside of a <pattern>.

If PATT is ($ ($1 $2 $) $ (/T 2 1)) and WS is (X Y Z (A B) (B C D) E F G B Y), a match occurs with <pattern> matching [(B C D)] and the <mark> (/T 2 1) matching [B]. The (/T 2 1) refers to the first elementary <pattern> inside the second elementary pattern, namely the $1 in ($1 $2 $) and this matches with [B] in (B C D). Similarly, one can write (QUOTIENT $1 (TIMES $ 2 $)) as a pattern for determining whether or not a quotient (in LISP formalism) has a common factor in it. The 2 then refers to the $1 because this is the second elementary pattern in PATT at the top level. More will be said about this later.

At this point we shall introduce a formal and concise notational definition of the elementary patterns, which have been introduced above in a very sketchy manner. Many generalizations have been omitted. For example, it is possible to indicate with a <mark> a reference to the top level <pattern> or the current level <pattern> or to count backwards from the present elementary pattern, etc. Similarly one can replace the numbers in a <mark> with the result of a computation, i.e. a <form>. A <pattern> can also be the result of computation, or even the evaluation of a <mark>, and can refer to segments as well as single items. In fact, a user who needs a particularly exotic elementary pattern has a very good chance that some combination of the generalizations in the definitions below will fulfill his purpose. Otherwise, he can always define a new elementary pattern as discussed in section 2.5.

2.1.2 Notation and Definitions

In this section we give a syntactic definition of the match portion of the FLIP language. "The definition is given in Backus notation with the addition of three dots (...) to avoid naming unnecessary syntactic types.

In Backus notation the symbols "::=", "<", ">", and "|" are used. The rule:

<S-expression> ::= <atomic symbol>|(<S-expression>.<S-expression>)

means that an S-expression is either an atomic symbol, or it is (sic) a left parenthesis followed by an S-expression followed by a dot followed by an S-expression followed by a right parentheses. The vertical bar means "or", and the angular brackets always enclose elements of the syntax that is being defined." (-9-)

i. <ep> ::= <elementary pattern>;
   <ep>$^{\dashv}$ ::= the segment matched by <ep>;
   <ep>$^{\vee}$ ::= the internal representation of <ep>$^{\dashv}$ (also called the value of <ep>).

If <ep>$^{\dashv}$ is [], <ep>$^{\vee}$ is NIL; if <ep>$^{\dashv}$ is [S], <ep>$^{\vee}$ is (S); if <ep>$^{\dashv}$ is [S1 S2 ... Sn], <ep>$^{\vee}$ is (S1 S2 ... Sn). Note that this allows the distinction between an <ep> that matches a segment [A B], whence <ep>$^{\vee}$ = (A B), and an <ep> matching a segment [(A B)], whence <ep>$^{\vee}$ = ((A B)). Similarly, if <ep> matches [], then <ep>$^{\vee}$ is NIL, while if <ep> matches [NIL], then <ep>$^{\vee}$ is (NIL).

We will define an <ep> by giving its value, i.e., the internal representation of the segment it matches. Thus in xxii <varp>$^{\vee}$ = (<var>$^{\vee}$) means <varp> matches a segment whose internal representation is (<var>$^{\vee}$), which implies that it is a segment consisting of the single item <var>$^{\vee}$. In xxiii, <varp>$^{\vee}$ = <var>$^{\vee}$ means <varp> matches a segment whose internal representation is <var>$^{\vee}$. Therefore, if <var>$^{\vee}$ is (X Y Z), <varp> will match [X Y Z]; if <var>$^{\vee}$ is ((X Y Z)), <varp> will match [(X Y Z)].

ii. <var> ::= <sexp>; <var>$^{\vee}$ ::= <sexp>$^{\vee}$

****

(-9-) LISP 1.5 Programmers Manual, p. 8

iii. <sexp> ::= <atom>; <sexp>^{v} ::= <atom>

iv. <sexp> ::= (QUOTE <S-expression>);
    <sexp>^{v} ::= <S-expression>

Definitions such as iii, iv, v, and vi are disjunctive, i.e. an <sexp> is either an <atom> or (QUOTE <S-expression>), etc.

v. <sexp> ::= <nonlist>; <sexp>^{v} ::= <nonlist>

vi. <sexp> ::= <list>; <sexp>^{v} ::= <list>,
The first member cannot be *, **, =, QUOTE, $SET, $*, $**, or a <number> (-10-)

vii. <var> ::= <form>; <var>^{v} ::= <form>^{v}

viii. <form> ::= (= <S-expression>);
      <form>^{v} ::= the value of (EVAL <S-expression> A), where A is an alist and is one of the inputs to the matching process (-11-)

ix. <form> ::= (= <fn> <var> <var> ... <var>);
    <form>^{v} ::= the value of (APPLY <fn> (<var>^{v} <var>^{v} ... <var>^{v}) A), where a modified APPLY is used so that if <fn> is an FEXPR or FSUBR the desired operation is still performed.

****

(-10-) We have already indicated on page 12 that the <ep> ($ F $) is a <pattern>. iv seems to imply that ($ F $) is also an <sexp>. The actual interpretation taken by the translator, PATTRAN, depends on the setting of the QUOTE mode, (see 2.3). (The top level <pattern>, i.e. the input to the match, is always treated as a <pattern>, regardless of the setting of QUOTE.) If QUOTE is *T*, both ($ F $) and, for example, (A (B C) D E), will be treated as S-expressions, i.e. <sexp>'s. If QUOTE is NIL, both are treated as <pattern>'s. Thus iv is correct provided QUOTE is *T*. If QUOTE is NIL, xxx applies. Note that regardless of the setting of QUOTE, (QUOTE (A (B C) D E)) is always an <sexp>, and ($* (QUOTE ($ F $))) (see xxxi) is always a <pattern>.

(-11-) The match is performed by a function MATCH1 of three arguments, WS PATT A (see section 2.5). The value of MATCH1 is NIL if WS does not match PATT, and is the parsing of WS with respect to PATT if the match succeeds.

x. <var> ::= <mark>, <var>^{v} ::= <mark>^{v}

xi. <mark> ::= <number>;
    <mark>^{v} ::= same as (/T <number>) - see xiv

xii. <mark> ::= (<number> <mark1> ...<mark1>);
     <mark>^{v} ::= same as (/T <number> <mark1> ... <mark1>) - see xiv.

xiii. <mark1> ::= <number> | <var>;
      <mark1>^{v} ::= <number> | <var>^{v}, and must be a <number> or else an error occurs.

xiv. <mark> ::= (/T <mark1> <mark1> ... <mark1>);
     <mark>^{v} ::= see the comments following xvii

xv. <mark> ::= (/C <mark1> <mark1> ... <mark1>);
    <mark>^{v} ::= see the comments following xvii

xvi. <mark> ::= (/U <mark1> <mark1> ... <mark1>);
     <mark>^{v} ::= see the comments following xvii

xvii. <match> ::= ($MATCH <list> <list> ... <list>);

A <match> is the internal representation of an object which has been matched by a <pattern>. The first <list> is the internal representation of the object, i.e. <ep>^{v} for the <pattern>, the rest of the <list>'s are the parsing, each <list> being an <ep>^{v}. Thus if WS is (A B C D), PATT ($ B $), the <match> is ($MATCH ((A B C D)) (A) (B) (C D)) corresponding to the parsing [A] [B] [C D]. If WS is (A (B C) D (B E F) G) PATT ($ ($ F $) $), <match> is ($MATCH ((A (B C) D (B E F) G)) (A (B C) D) ($MATCH ((B E F)) (B E) (F) NIL) (G)).

The evaluation of a <mark> is very similar for the three cases, xiv, xv, and xvi, listed above. The only difference lies in the starting point of the evaluation process. For /T the top level <match> is used, i.e. the one being assembled by MATCH1. For /C the current level <match> is used. This will be the same as /T if the <mark> in question is at the top level, i.e. is not inside of a <pattern>. For /U, the <match> used is the one n levels up from the current level, where n is <mark1>^{v}, and <mark1> is the first <mark1> in <mark>. (If n is too large, for example if <match> is already at the top level, an error occurs.)

Once the correct <match> is located, the evaluation process is the same for all <mark>'s. MARKVAL (the evaluation function) successively evaluates each <mark1>, and if the resulting number n is positive, takes the <list> corresponding to the nth <ep> of the <match> being used, and if negative, the <list> corresponding to the nth <ep> from the end of the <match> being used. This <list> must be a <match> and is then used as <match> for the next <mark1>. When the last <mark1> is evaluated, the corresponding

<ep>^{v} is found, and this is the value of the <mark>. If a <list> is encountered before this which is not a <match>, or if n is too large, an error occurs.

Since this process is a little difficult to view abstractly, we include here several examples illustrating facets of the above description. Although we use numbers throughout the examples, remember that a <mark1> can also be a <form>, i.e. the result of a computation.

Suppose WS is (A B C (D E F) G H (I J K L (M N (O P Q)))) and PATT is ($ $1 ($1 $ F) $ ($1 $ $2 ($ ($1 P α)))), where α is some <mark>. At the time α is evaluated, the top level <match> is:

```
($MATCH ((A B C (D E F) G H (I J K L (M N (O P Q)))))
   (A B)
   (C)
   ($MATCH ((D E F)) (D) (E) (F))
   (G H)
   ($MATCH ((I J K L (M N (O P Q))))
      (I)
      (J)
      (K L)
      ($MATCH ((M N (O P Q)))
         (M N)
         ($MATCH ((O P Q))
            (O)
            (P      )))))
```

For α	the value of α is
(/T 2), or just 2	(C)
(/T -3), or just -3	(C) - same as above
(/T 3)	((D E F))
(/T 3 1), or (3 1)	(D)
(/T 2 1)	ERROR
(/T 6)	ERROR
(/T -2 2)	(E)
(/T -2 -1)	(F)
(/T 5 2)	(J)
(/T 5 4 2 2)	(P)
(/U 1 -1)	(M N)
(/U 1)	((M N (O P Q))
(/U 3 1)	(A B)
(/U 3)	((A B C (D E F) G H (I J K L (M N (O P Q)))))
(/U 3 -1)	(G H)
(/U 4)	ERROR
(/C 1)	(O)
(/C -1)	(P)
(/C)	((O P Q))

xviii. <ep> ::= <varp>; <ep>$^{\vee}$ ::= <varp>$^{\vee}$

This is the first <ep> we have defined so far.

xix. <varp> ::= <sexp>; same as (" <sexp>)

xx. <varp> ::= <form>; same as (* <form>)

xxi. <varp> ::= <mark>; same as (** <mark>)

xxii. <varp> ::= (* <var>); <varp>$^{\vee}$ ::= (<var>$^{\vee}$)

Here <varp> matches a segment consisting of the single item <var>$^{\vee}$; hence <varp>$^{\vee}$ = (<var>$^{\vee}$).

xxiii. <varp> ::= (** <var>); <varp>$^{\vee}$ ::= <var>$^{\vee}$

Here it matches a segment of WS equal to <var>$^{\vee}$. Thus <var>$^{\vee}$ must be a list, or else an error occurs.

xxiv. <ep> ::= <dol>; <ep>$^{\vee}$ ::= <dol>$^{\vee}$

xxv. <dol> ::=.$; <dol>$^{\vee}$ ::= <list>

Thus <dol> matches any segment.

xxvi. <ep> ::= <doln>; <ep>$^{\vee}$ ::= <doln>$^{\vee}$

xxvii. <doln> ::= $n, n a <number>;
<doln>$^{\vee}$ ::= <list> of length n

xxviii. <doln> ::= ($N <var>);
<doln>$^{\vee}$ ::= <list> of length <var>$^{\vee}$,

if <var> is not a number, an error occurs.

xxix. <ep> ::= <pattern>; <ep>$^{\vee}$ ::= <pattern>$^{\vee}$

xxx. <pattern> ::= (<ep> <ep> ... <ep>);
<pattern>$^{\vee}$ ::= same as ($* (QUOTE (<ep> <ep> ... <ep>))) (-12-)

****

(12) This assumes QUOTE is NIL, otherwise this is an <sexp> and falls under vi (see also footnote 10). If QUOTE is *T*, use ($* (<ep> <ep> ...))). (Note that ($* (<ep> <ep> ... <ep>)) is <u>not</u> correct if QUOTE is NIL as (<ep> <ep> ... <ep>) will then be a <u><pattern></u> and $* expects a <u><var></u>. However, ($* (QUOTE (<ep> <ep> ... <ep>))) is always correct.

xxxi. <pattern> ::= ($* <var>);
<pattern>$^{\vee}$ ::= (<list>), where <list> matches (the translation of) <var>$^{\vee}$, in the sense defined here (-13-)

In other words, <pattern> matches a segment consisting of a single item, which is a list. Although the value of <pattern>$^{\vee}$ as an <ep> is (<list>), what is appended to the <match> being constructed by MATCH1 is actually ($MATCH (<list>) . segment1 segment2 ...), or in other words, the parsing of <list> with respect to <var> .

xxxii. <pattern> ::= ($** <var>);
<pattern> ::= <list>, where <list> matches the (translation of) <var> etc. as above

Here <list> is a segment of WS, not a single item. Again what is appended to <match> is ($MATCH <list> <seg1> <seg2>...). Thus, ($ ($** (<ep1> ... <epn>)) $) matches the same things as ($ <ep1> ... <epn> $), except that the $** has only one <ep>$^{\vee}$, (although it contains an <ep> for each of the n <ep>'s), and the S-expression (<ep1> ... <epn>) may be replaced by a <form> or a <mark>.

Again we include some examples.

****

(-13-) The list structure that is <var>$^{\vee}$ will be altered by the translation process (see section 2.3). This means that if a quoted S-expression is used, it will be translated only once. However, it also means that if it is not desirable to have the structure clobbered, one should use a <form> with the function COPY.

Suppose WS is (A B C ($1 $ (/C 1)) FOO D X A (B C D B) (B C D B A) X Y), PATT is ($ $1 FOO $ ($* (= COPY (= CAR 2))) $). When <pattern> is entered, the $1 matches [($1 $ (/C 1))], i.e. the value of <match> is ($MATCH ((A B C ($1 $ (/C 1)) FOO ...)) (A B C) (($1 $ (/C 1))) (FOO) ()).

The value of (= CAR 2) is therefore ($1 $ (/C 1)) and this is (copied and) translated and used as a pattern to match against the single item which is next in WS, namely D. The match fails (immediately because D is not a list) so the second $ extends its segment to include D (it now matches [D]). (= CAR 2) is reevaluated, translated (if we were not concerned about preserving the structure ($1 1 (/C 1)), we could omit the COPY and then only one translation would be needed) and reapplied, this time to the item X. The match fails again. Ultimately it succeeds matching with (B C D B), and the final value of <match> is ($MATCH ((A B C ...)) (A B C) (($1 $ (/C 1))) (FOO) (D X A) ($MATCH ((B C D B)) (B) (C D) (B)) ((B C D B A) X Y)). Note the <match> corresponding to <pattern>.

If WS had been (A B C ($1 $ (= CAR 1)) FOO ...), then <pattern> would have matched with (B C D B A), and <match> would have been ($MATCH ((A B C ...)) (A B C) (($1 $ (= CAR 1))) (FOO) (D X A (B C D B)) ($MATCH ((B C D B A)) (B) (C D B) (A)) (X Y)).

The $** <ep>, as illustrated by the next examples, is used to match a segment of WS. Suppose WS is as above (with (/C 1)) but PATT is ($ $1 FOO $ ($** (= COPY (= CAR 2))) $). The value of <match> here would be ($MATCH ((A B C ...))) (A B C) (($1 $ (/C 1))) (FOO) (D) ($MATCH (X A (B C D B) (B C D B A) X) (X) (A (B C D B) (B C D B A)) (X)) (Y)). Note that the value of <pattern> is a segment of WS, [X A (B C D B) (B C D B A) X] which matches ($1 $ (/C 1)). If we change WS as before to be (A B C ($1 $ (= CAR 1)) ...), then <match> is ($MATCH ((A B C ...)) (A B C) (($1 $ (= CAR 1))) (FOO) ($MATCH (D X A) (D) (X) (A)) ((B C D B) (B C D B A) X Y)).

The "Abort Predicate," "Fast $," and "Failure Predicate" extensions of <dol> described below may all be used inside of a <pattern>, either of the $* or $** variety. In the $* case, they act exactly as they do in PATT, which is itself a $* <pattern>. In the $** case, some care must be exercised involving <mark>'s and the "Failure Predicate," and the "Fast $" will not operate across the interface between the $** <ep> and the rest of the <ep>'s. We will discuss this in detail in the description of the operation of PATTERN and PATTERN1 in section 2.5.

xxxiii. <npred> ::= (<fn>)|(<fn> <var> ... <var>);
<npred>$^{\vee}$ ::= the value of (APPLY <fn> (<ep>$^{\vee}$ <var>$^{\vee}$ ... <var>$^{\vee}$) A), where <ep>$^{\vee}$ is the value of the segment matched by the <ep> with which <npred> is

associated, and a modified APPLY is used so that FEXPRS and FSUBRS will word. (-14-)

xxxiv. <ep> ::= <varf>; <ep>^{v} ::= <varf>^{v}

xxxv. <varf> ::= <varp> / <npred>;
      <varf>^{v} ::= <varp>^{v} provided <npred>^{v} is not NIL.

In other words, the segment matched by <varf> is identical to that matched by <varp> with the added constraint that the value of <fn> given <varp>^{v} as its first argument, and the evaluation of the rest of its (optional) arguments, is not NIL.

Example: If X is C and PATT IS ($ $3 (= X) / ((LAMBDA (Y Z) (NOT (MEMBER (CAR Y) Z))) 2) $), the result of matching with WS (C A B C D C E F G C H I) is ($MATCH ((C A ...)) (C A B C D C) (E F G) (C) (H I)). Note that the value of the argument Y is <ep>^{v} , and therefore one must take CAR of it for use with MEMBER.

xxxvi. <ep> ::= <doinf>; <ep>^{v} ::= <doinf>^{v}

xxxvii. <doinf> ::= <doin> / <npred>;
      <doinf>^{v} ::= <doin>^{v} with the added constraint that <doin> satisfy <npred> as above.

Example: ($ $3 $ $3 / ((LAMBDA (X Y) (EQUAL (REVERSE X) Y)) 2) $) will match with (A B C D E F D C B X) producing ($MATCH ((A B ...))) (A) (B C D) (E F) (D C B) (X)).

xxxviii. <ep> ::= <doif>; <ep>^{v} ::= <doif>^{v}

xxxix. <doif> ::= <doi> / <npred>;
      <doif>^{v} ::= <doi>^{v} provided <npred>^{v} is not NIL or (NIL).

Example: ($ $1 $ / ((LAMBDA (X) (GREATERP (LENGTH X) 5))) 2) $) will match with (A B C D A D B C B D) to produce ($MATCH ((A B ...)) (A) (B) (C D A D B C) (B) (D)).

****

(-14-) <npred> may be equal to NIL. In this case, it is the same as though it always had the value *T*. Thus $1 / NIL is the same as $1 / ((LAMBDA (X) T)), and the same as just $1.

The Abort Predicate

The <dolf> is a generalization of <dol> analagous to that of <doinf> over <doin> and <varf> over <varp>. In other words, <dolf> acts like <dol> with side constraints in what it matches. However, here the similarity ends.

A <varf> or <doinf> can determine immediately whether or not it will match, because its <ep>^{v} is determined within itself. A <dolf> however can match more than one segment, and its <ep>^{v} is really determined by the requirements of the other <ep>'s in the <pattern>. This means that the <dolf> has the unique property of being able to try other <ep>$^{-1}$'s even after finding an acceptable match. It also implies that a considerable amount of the effort spent on a match may be involved in the <dolf> search, and that it is appropriate to find ways to streamline it and make it more efficient.

Consider the case where PATT is ($ $1 $ / ((LAMBDA (X) (LESSP (LENGTH X) 5))) 2 $) and WS is (A B C D E F G A D H I J K L M ...). The <dol> initially matches [] and the $1 [A]. The <dolf> initially matches [] but [B] does not equal [A]. The <dolf> then matches [B], but [C] does not equal [A]. Finally, the <dolf> matches [B C D E], but [F] does not equal [A]. Next the <dolf> tries to match with [B C D E F]. Since the length of this segment is not less than five, it will not match, so the <dolf> tries to match with [B C D E F G]. Again the length of this segment is not less than five, so it does not match. Similarly, the <dolf> will try to match with each segment until it exhausts the list WS. Only then will it determine that it cannot match and return control to the first <dol>. This will then match with [A] and the <doin> with [B]. The <dolf> search will be repeated. Finally, a match occurs with value ($MATCH ((A B C ...))) (A B C) (D) (E F G A) (D) (H I J ...)). This is of course correct and what was intended, but it could have been found with a more efficient handling of the <dolf> search.

The first extension of <dolf> is designed to solve this problem. It is called the "abort" feature, and allows the user to specify under what conditions a <dolf> search can be terminated. If the value of <npred> applied to the appropriate inputs is ever NIL, the <dolf> fails to match, and no further search occurs. If the value is (NIL), the <dolf> does not match this segment, but the search continues as before. In the example given above, the <dolf> would not have attempted any further matches once it tried the segment [B C D E F] since the value of <npred> was then NIL. If we wanted the segment matched by <dolf> to have a length between 5 and 10, we could use (LAMBDA (X) (COND ((LESSP (LENGTH X) 5) (LIST NIL)) ((GREATERP (LENGTH X) 10) NIL) (T T))).

```
xl. <dolf>  ::= $ // <npred>;
    <dolf>v ::= same as xxxix
```

The Fast $

The second extension of the <dolf> allows a faster and more efficient search in certain instances. Consider the case where PATT is ($ $1 $ / ((LAMBDA (X) (LIST (GREATERP (LENGTH X) 10)))) 2 $). The <npred> associated with the <dolf> will be applied to the segment that the <dolf> is matching each time a match is attempted. Then control will be passed to the <varp> 2. If this fails, the <dolf> extends its segment and reapplies the predicate. It would be desirable if the predicate were only applied once the <varp> 2 found an acceptable segment. In fact, it would be nice if it were not necessary to leave <dolf> and reenter the <varp> for each attempted match, and we could essentially evaluate the <varp> and run along WS looking for something matching it and only then apply the predicates, etc. This is accomplished by means of the "fast" feature. When a // appears with <dolf> and an <npred>, if the next <ep> is a <varp>, <varf>, <doln>, or <dolnf>, the <dolf> search is handled in FAST mode. (-15-)

****

(-15-) If FAST is set to *T* at run time, all <dolf>'s will be treated as though a // were used. If FAST is set to *T* at translation time, all <dolf>'s will be treated as though a // were used regardless of what is the value of FAST at run time. This is usually the desired mode of operation except where it is necessary to have some <dolf>'s not operate in FAST mode. In this case, FAST should be NIL and these <dolf>'s written with a single /.

In order to properly explain the effect of the FAST mode, we must discuss the way in which a match normally proceeds. As we will see in section 2.3, the translation of a pattern corresponds to a sequence of function calls, one for each <ep>. At the time it is called, each <ep> function is given arguments relating to its particular operation, e.g. for a <doinf> the arguments are the <var> whose value will be the length of the segment, and the predicate which this segment must satisfy. In addition, each function is supplied with three arguments: the current WS, the current MATCH, and the current PATT. If the function determines that it matches a segment at the beginning of WS, it passes control to the next <ep> function indicating what WS it should operate on, together with the new, modified MATCH, and PATT (which is the old PATT minus the <ep> function which has just operated). If the function corresponding to the first <ep> does not match an initial segment, it returns a value to the last <ep> function indicating failure.

For example, if WS is (A B C D) and PATT is ($ $1 C $), the input to VARF the first time it is entered is (B C D) for WS, ($MATCH ((A B C D)) NIL (A)) for MATCH, ((DOLF NIL NIL NIL)) for the rest of PATT, and three arguments relating to the needs of VARF. These are (SEXP C), NIL, NIL (the first NIL indicates the <var> represented by (SEXP C) is to be treated as an item, and the second indicates there is no predicate associated with it). Since (SEXP C) has the value C, and C does not match B, the first thing in WS, VARF returns a value that indicates it did not succeed in matching.

The second time VARF is entered, DOLF having extended its <ep> , WS is (C D), MATCH is ($MATCH ((A B C D)) (A) (B)), again ((DOLF NIL NIL NIL)). This time (SEXP C) matches with the C at the beginning of the WS, so VARF passes control to the next <ep> giving it (D) as WS, (($MATCH ((A B C D)) (A) (B) (C)) as MATCH, and NIL for the rest of PATT. This <ep> is the last <dol> and it matches the rest of WS, namely [D].

This procedure will be exam"ned in greater detail when we explain the operation of each funct on in section 2.5. At this point, however, it should be clear already that the DOLF function assumes a special role. It is the only <ep> which can resume operation once it has matched a segment and passed control on to the next <ep>. In the example above, when VARF indicated it failed to the DOLNF, which was the function entered immediately before it, DOLNF then also failed. In other words, the match had determined that starting from (A B C D), $1 followed by C did not mach. Even though $1 did match, the entire match would have failed had these two <ep>'s not been proceeded by a $, or call to DOLF.

When DOLF matches an acceptable segment, it passes control on to the next <ep> function. Should everything match from then on, the segment matched by DOLF at this point is the one corresponding to reverts to the last $, and it resumes trying to find an acceptable segment, exactly as though it had never matched before.

Let us consider yet another example. Suppose WS = is (A B C B D C E) and PATT ($ $1 $2 2 $). DOLF, corresponding to the first $, would be entered initially with WS (A B C B D C E), MATCH = ($MATCH ((A B C B D C E))), and PATT = ((DOLNF 1 NIL) (DOLNF 2 NIL) (VARF *T* (MARK /T 2) NIL) (DOLF NIL NIL NIL)). Note that PATT corresponds to the translation of the rest of the pattern, namely ($1 $2 2 $). Since no predicates are associated with this DOLF, it matches the first acceptable segment, namely the null segment, and passes control to the next <ep>, a DOLNF, with WS = (VARF *T* (MARK A B C D B D C E), (the same as it was before), PATT = ((DOLNF 2 NIL) (VARF *T* etc. and MATCH = ($MATCH ((A B C B D C E)) NIL).

The DOLNF matches with (A) and passes control to the next DOLNF giving it (B C B D C E) for WS, ((VARF *T* etc.)) for PATT, and ($MATCH ((A B C B D C E)) NIL (A)) for MATCH. This DOLNF also matches, with (B C), and passes control to the VARF with WS (B D C E), ((DOLF NIL NIL NIL)) for PATT, and MATCH ($MATCH ((A B C B D C E)) NIL (A) (B C)). The VARF fails because the value of (MARK /T 2) is (A), and (A) does not match the beginning of (B D C E).

At this point, control passes all the way back to the first DOLF. This DOLF then matches with (A), and passes WS (B C D B D C E), MATCH ($MATCH ((A B C B D C E) (A)) and PATT ((DOLNF 2 NIL) (VARF *T* ...)) onto the next DOLNF and the process continues. (-16-)

****

(-16-) The user can observe the operation of FLIP in detail by setting TRACE to *T* and executing any FLIP statement (see section 2.4). Results from a partial trace are contained in section 2.2.

The FAST mode is designed to eliminate some of this process by decreasing the number of function calls necessary to complete a match. If a DOLF is entered and the next function call in PATT is to a DOLNF or VARF, a fast DOLF does not merely match up with the first acceptable segment and pass control. It will not attempt to match any segment unless the DOLNF or VARF is also going to be satisfied. In fact, the DOLNF or VARF is never actually entered (and will not be printed out if the TRACE mode is turned on). What DOLF does do is evaluate the <var> used by DOLNF or VARF, and if there is a predicate, also evaluate the inputs. This is done just once. Then DOLF searches WS to find the first segment that will satisfy the DOLNF or VARF. When this segment is found, DOLF applies its own predicate, if any, to the segment consisting of all of WS up till the segment that matched the DOLNF or VARF. If the predicate yields NIL, DOLF abandons search and reports failure, as in the "abort" discussion above. If it yields (NIL), DOLF continues searching. Otherwise, DOLF matches, and so does VARF/DOLNF, and control is passed to the next <ep> after the DOLNF/VARF.

In a previous example, where PATT was ($ $3 $ $3 / ((LAMBDA (X Y) (EQUAL (REVERSE X) Y)) 2) $), and WS was (A B C D E F D C B X), the sequence of function calls would be as follows: first DOLF would be entered and match with NIL, then DOLNF would match with (A B C). The second DOLF would match with NIL, and the second DOLNF would fail. The second DOLF would match with (D), the second DOLNF would fail again (note that it had to reevlaluate the value of the <mark> 2 which was an input to its predicate). The second DOLF would match with (D E), the second DOLNF would fail again, etc. Finally, the second DOLF, after putting in seven calls to the second DOLNF with none of them succeeding, would fail. At this point the first DOLF would match with (A), the first DOLNF with (B C D), and we would continue.

If instead the second DOLF were a fast DOLF, i.e. if it were $ // NIL, or if FAST were *T*, the sequence of function calls would now be as follows: the first DOLF would be entered and match with NIL, then DOLNF would match with (A B C). The second DOLF would evaluate the <mark> 2 getting (A B C), and determine that it was looking for a segment of length 3 which satisfied a certain function of the two arguments consisting of this segment and (A B C). It would then run through WS and determine that no such segment existed, and therefore fail The first DOLF would then match with (A), the DOLNF with (B C D). The second DOLF would evaluate the <mark> 2, getting (B C D), determine it was looking for a segment of length 3 such that this segment and (B C D) satisfied a certain predicate, and would find (D C B). Since this DOLF matches any segment (there being no predicate), it would match with (E F), and match the DOLNF with (D C B), the segment DOLF had found for it, and pass control to the next <ep>, which is the last DOLF.

The FAST mode, as with the case of the "abort" feature, does not affect the ultimate value of the matching process, (-17-) it merely affects the way in which the search for this parsing is performed. Because the search is performed differently in FAST mode, however, a few words of caution are necessary. First, since the DOLNF or VARF in question is not actualy entered, nor is the segment matched by DOLF actually appended to the <match> until both of them are satisfied, negative <marks> must be adjusted accordingly. Thus ($ $1 $ -2 $) will not operate in FAST mode with the intended meaning. What should be used is ($ $1 $ _-1_ $) because when the <mark> is evaluated, it will be done so inside of the DOLF, which will then have a parsing corresponding only to the first <dolf> and the <dolnf>. Similarly, something like ($ ($N (= LENGTH 1)) $) will not work in FAST mode because the 1 refers to the first DOLF, which does not have an <ep> in the parsing until after the <dolnf> has matched. In both cases an error will occur in MARKVAL stating that the <mark> in question is too large. The user can avoid this by bearing in mind what actually is going on when the <dolf> is running in FAST mode.

****

(-17-) Strictly speaking, this is not true. It is possible to write FLIP statements so that the "abort" feature will cause a correct parsing _not_ to be found, even where one exists. For example if we assume EVENP to be a predicate with the obvious definition, then if WS is (A B C D E F G), and PATT ($ / ((LAMBDA (X) (EVENP (LENGTH X)))) E $), there will be no <match> because when <dolf> matches with [A], its length is not even. However, note that the parsing [A B C D] [E] [F G] is correct (since the length of [A B C D] is even). Note that if this was run with a FAST $, it would work, because the DOLF would not apply its predicate until it found the E, at which time it would be satisfied. However, we can also find cases in which a particular FLIP rule will not operate in FAST mode, as it is written, even though it will in non-fast mode (see warnings above). All this means is that the user must occasionally be a little circumspect when using these features.

One special convention is available, however, which does allow a <doinf> or <varf> to refer to the value of the <dolf> immediately proceeding it even when that <dolf> is running in FAST mode. This is done by making the first input to its predicate the <mark> 0, which is then checked for specially by DOLF. For example, ($ $3 / ((LAMBDA (X Y) (SUBSET X (CAR Y))) 0) $) will match with (A B C D C B E D A D X Y); to produce [A B C D B E] [D A D] [X Y].

The actual value of the <mark> 0 is list of <dolf>$^{-1}$ (hence the use of CAR in the example above), because we wish to have <dolf> evaluate the inputs to the predicate only once, and yet still have the <mark> 0 indicate the latest segment matched by the $. This means that the <mark> must be a pointer to a list structure which the DOLF function can change. Since this list structure is initially NIL, we must list it in order to provide a structure which DOLF can then modify. (We cannot actually modify NIL without doing dreadful things to LISP.)

The Failure Predicate

The third extension of the <dolf> involves the resuming of search once the <dolf> has matched and passed control to the next <ep>. Occasionally it is necessary to base the decision to resume search on information relating to why the subsequent match failed. This is done by means of the "failure" predicate.

```
          xii. <dolf> ::= $ / <npred> / <npred>;
               <dolf>ᵛ ::= same as xxxix

          xiii. <dolf> ::= $ // <npred> / <npred>;
                <dolf>ᵛ ::= same as xxxix
```

Once the <dolf> has passed control onto the next <ep> (or the one following that in the case of a fast <dolf>), and control reverts to it, the failure predicate, which is the second <npred> in xii and xiii above is applied, with its arguments evaluated against the results of the entire match. Any <ep> that did not match is assigned an <ep>^{v} consisting of the single atom "FAILED". Other <ep>'s that did match have values as defined above. <dolf> evaluates these arguments and gives them to the "failure" <npred> (in this case, the segment matched by the <dolf> itself is not automatically the first argument of the <npred> as above - you must ask for it). If the value of the <npred> is NIL, the <dolf> does not continue searching but passes control back up the line (perhaps to another <dolf>). Otherwise, it continues searching.

Example: find an item in a list that is repeated four times. If PATT is ($ $1 2 $ 2 $ 2 $) and WS is (A B C A D E A B C B C D C), then when <doinf> matches A, the first <mark> will also match as will the second <mark>, because A is repeated three times. The third <mark> will not match. When control reverts to the third <dolf>, however, it will go on looking for an A. Since the fourth A was never found, this <dolf> will not find another third A. Ultimately it fails and passes control back to the second <dolf>. This <dolf> will continue searching for a second A, find one, namely what had been the third A, and pass control on to the next <dolf> to look for more A's, etc.

If the pattern used was ($ $1 $ / NIL / (NO 4) 2 $ / NIL / (NO 6) 2 $ / NIL / (NO 8) 2 $), (-18-) the <dolf>'s would continue searching only until they found a match for the <mark> immediately following them. When the fourth A was not found and the fourth <dolf> failed, the parsing was [] [A] [B C] [A] [D E] [A] [B C B C D C]. Control passes to the third <dolf> where the <mark> 6 is evaluated against ($MATCH ((A B C A D E A B C B C D C)) NIL (A) (B C) (A) (D E) (A) (B C B C D C) FAILED FAILED). (-19-) Since the 6th <ep> did match, its <ep> being (A), (NO 6) is NIL and the <dolf> fails. Similarly for the third and second <dolf>.

****

(-18-) NO is a function which is true only if its argument did not match, i.e. it is (EQ X (QUOTE FAILED)). YES is not NO.

(-19-) Actually, what is returned from the failure is (($MATCH ((A B C ...)) NIL (A) (B C) (A) (D E) (A) (B C B C D C))). The extra parentheses indicate that a failure has occurred. When DOLF evaluates the failure predicate, it sets a flag so that MARKVAL returns FAILED where it previously would have given an error. Note that negative marks will count backwards from wherever the <match> ended, and therefore are not really meaningful.

If fast <doif>'s were used, we could also write ($ $1 $ / NIL / (NILL) 2 $ / NIL / (NILL) 2 $ / NIL / (NILL) 2 $), where the value of NILL is NIL. In this case, the <doif> does not give up control until it finds a match for the <mark> following it. The (NILL) insures that it will not continue looking if it gives up control and then regains it.

Although this example is extremely simple, and could be done by means of a suitably designed "abort" predicate instead, it gives the general flavor of the failure predicate. The longest common substring problem in the next section uses the "failure" predicate in a more interesting fashion.

xliii. <doif> ::= $ / / <npred>; same as $ / NIL / <npred>

xliv. <doif> ::= $ // / <npred>; same as $ // NIL / <npred>

Before leaving the failure predicate, we should emphasize two features. First the failure predicate is only applied when control reverts to the <doif> after a failure. If the <doif> is not in fast mode, this will be after each attempted match by the next <ep>. If the <doif> is in fast mode this will be only after the next <ep> has matched, and some subsequent <ep> has failed. Secondly, if the failure predicate is applied and does not yield NIL, the match continues exactly as though nothing had happened. If in fast mode, this means the <doif> continues to look for a match for the next <ep> if it is a <varf> or a <doinf>, applying its "abort" predicate only when it finds a match, etc. (It first reevaluates the arguments of the predicates, and the <varf>'s involved because the alist may have been changed.) If not in fast mode it extends it segment, applies the "abort" predicate, if any, and goes on.

xlv. <$set> ::= ($SET <atom> <var>)

<$set> is not an <ep> in the strictest sense. It does not match anything nor does it have a value. When using <mark>'s pretend the <$set> is not in the pattern at all. The purpose of $set is to alter the alist, A. It binds (or rebinds if already bound) the variable <atom> to the value of <var>. It uses the function $SET1 to perform this binding. $SET and $SET1 are described in section 2.5

xlvi. <doll> ::= $*$ / <npred>

<doll> is a device for allowing the user to assume control of the match. The value of the match is the value of <npred> given as its first three arguments WS, PATT, and MATCH, and the evaluation of the rest of its arguments (if any) against the current MATCH.

The very existence of <doll> guarantees that FLIP is universal, because it means we can peform arbitrary LISP computations. For example, if FN is a LISP function of two arguments, X and Y, then ($*$ / ((LAMBDA (A B C) (FN X Y)))) is a FLIP match statement having the value of (FN X Y), assuming that X and Y are bound to their correct values on FLIP's alist.

This also means that the presence of <doll> precludes the necessity of ever writing functions such as DOLF, VARF, PATTERN, etc., since one could write each of these instead by means of a <doll> function!

For example, one can write instead of ($ $3 A $ $1 B $)

```
  ($ $*$ / ((LAMBDA (WS PATT MATCH) (COND
    ((LESSP (LENGTH WS) 3) (LIST MATCH))
    (T  (MATCH2 (CDDDR WS) PATT (APPEND MATCH (LIST
                (CAR WS) (CADR WS) (CADDR WS))))  )))
   A  $   $*$ / ((LAMBDA (WS PATT MATCH) (COND
    ((NULL WS)  (LIST MATCH))
    (T (MATCH2 (CDR WS) PATT (APPEND MATCH (LIST (CAR WS))))) )))
   B  $)          (-20-)
```

The user, observing that operations such as $3, $1 etc., often arose in practice, might wish to define a function, call it DOLNF for want of a better name, which would take n as an input and perform the same task as the <doll>'s above. He would then be able to write

```
    ($ $*$ / (DOLNF (QUOTE 3)) A $ $*$ / (DOLNF (QUOTE 1))  B  $)
```

****

(-20-) MATCH2 is the main routing function for MATCH. It will be discussed in detail in section 2.5. Its effect is to call the next <ep>, which is obtained from PATT, giving it WS, (CDR PATT), and MATCH as arguments. It checks to see when PATT is finished, or if WS has been exhausted.

To avoid repeating the $*$ notation, he might even wish to introduce a new translating convention whereby ($N 3) meant (DOLNP (QUOTE 3)), etc., and so be able to simply write

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

This is exactly the way the $N pattern, and other elementary patterns discussed above, were developed, and there is no reason why a prospective user could not similarly define new operations, should they prove desirable, or even modify the existing ones supplied with the FLIP package.

The <doll> is intended to serve in those cases too infrequent to justify writing an entirely new <ep> function. All that is necessary is to observe the few conventions relating to communication between <ep> functions (unless the <doll> is to entirely assume control for the rest of the match, in which case no other <ep> functions will be called.) In most instances, much of the labor can be performed by existing functions in the FLIP package, such as MATCH2, MATCH3, VARVAL, MARKVAL, NEXT, FIRSTN, LASTN, etc. All the user then need do is assemble the parts.

2.2 Comparison of FLIP and LISP

2.2.1 A Simple Example

Programming languages can be compared in many ways, but the choice of one language over another for a particular task usually resolves itself into a subjective decision which depends on the nature of the problem, and on space and time requirements for the program. It is impossible to give an absolute evaluation of any language that will hold for all, or even a wide class of problems.

This is also true in the case of FLIP. However, since FLIP is embedded in LISP and is intended to be used by LISP users, we will attempt to strike some comparisons between the two.

FLIP is certainly slower than LISP. Although their relative speeds vary greatly depending on the nature of the problem, it is safe to say that a LISP compiled function is anywhere from two to five times as fast as a FLIP function to perform the same task. (FLIP is usually faster than a LISP program that is run interpretively, but again this depends on the problem and the particular function used.) This is obvious because FLIP itself is a collection of compiled LISP functions, and therefore cannot be faster than a special purpose LISP compiled function. When predicates are used, FLIP runs slower than LISP compiled functions because it must use the LISP interpreter to evaluate the predicates.

This, of course, is not very discouraging. A LISP program runs slower than written in LAP (an assembly program for LISP), and MAD programs are slower than FAP programs. It is in fact very heartening that the difference in running times is so small. Even if it were greater, there would still be some justification for developing a language such as FLIP. One usually chooses a "high level" language not for its speed, but because of increased understandibility and ease of writing. This point is well exemplified by a LISP function to perform the ($ $3 A $ $1 B $ = 1 5 4 2 C 7) transformation discussed in the introduction.

```
(LISPEXAMPLE
   (LAMBDA (X) (PROG (FIRST$ $3 SECOND$)
            (COND
               ((LESSP (LENGTH X) 6) (RETURN NIL)))
      T1    (COND
               ((NULL (CDDDR X)) (RETURN NIL))
               ((EQUAL (CADDDR X) (QUOTE A)) (GO T2)))
            (SETQ FIRST$ (CONS (CAR X) FIRST$))
            (SETQ X (CDR X))
            (GO T1)
      T2    (SETQ $3 (LIST
               (CAR X)
               (CADR X)
               (CADDR X)))
            (SETQ X (CDDDDR X))
      T3    (COND
               ((NULL (CDR X)) (RETURN NIL))
               ((EQUAL (CADR X) (QUOTE B)) (GO T4)))
            (SETQ SECOND$ (CONS (CAR X) SECOND$))
            (SETQ X (CDR X))
            (GO T3)
      T4    (RETURN (NCONC (REVERSE FIRST$) (CONS (CAR X) (
NCONC (REVERSE SECOND$) (NCONC $3 (CONS (QUOTE C) (CDDR X))
))))))))
NIL
```

A LISP program to perform the transformation:

$ + $3 + A + $ + $1 + B + $ = 1 + 5 + 4 + 2 + C + 7

(See text, page 32)

This function was written and debugged by an experienced LISP programmer in approximately one half hour (real time). This is not atypical - programs seldom run correctly the first time, and occasionally even the tenth time. This LISP function, when compiled, runs about three times as fast as the ($ $3 ...) FLIP rule. However, even though the variables have been given suggestive names such as FIRST$, SECOND$, etc., and great pains were taken to make the code clear and concise, it is not immediately apparent what the function is supposed to do, even using the FLIP rule as a guide. No one can defend the stand that this LISP function is more intelligible or readable than the corresponding FLIP statement, any more than that the compiled version of LISPEXAMPLE is as understandable as the S-expression definition given above. This is the point of using LISP, or FLIP: the standardization of frequently used operations, and the adoption of a meaningful notation for them.

It is instructive to note exactly where FLIP loses the race to LISP in the example above. In the area between T1 and T2, LISPEXAMPLE is looking for the atom A. The FLIP rule does this by repeated attempts to match the fourth element in WS with A, just as LISPEXAMPLE does. However, it is done by calling the function DOLNF followed by VARF, instead of by (EQUAL (CADDDR X) (QUOTE A)).

However, this is the same rationale that led us to define the functions DOLNF and VARF in the first place: generality versus specificity. The DOLNF and VARF can be used for a wide range of tasks. The COND, two SETQs, and GO statement in LISPEXAMPLE that perform the same operation are more specialized.

Had we wished to make the FLIP rule that performs the transformation more special purpose, and correspondingly faster, there is a continuity available extending all the way to writing ($*$ / ((LAMBDA (A B C X) (LISPEXAMPLE X)))) for the entire FLIP rule. In particular, we could avoid the repeated calls to DOLNF by writing

```
($ // ((LAMBDA (X) (GREATERP (LENGTH X) 3)))  A  $ // ((LAMBDA (X)
(NOT (NULL X))))    B  $)
```

With a fast $, only three <ep> functions would be entered by this FLIP rule. Unfortunately, we would lose back some of the time gained because of the necessity of using the LISP interpreter to evaluate the predicates (again paying for the generality of FLIP). If we instead built a process of this type into FLIP, the speed would be considerably increased.

For example, let us define a function which runs along WS looking for A, and then checks to see if what led up to it was longer than 3 in length. This would be similar to a fast $, but with a $N included in it. We could call it FAST$N. We would have to define FAST$N as a function of WS, PATT, MATCH, N, and X, the last argument being the object of the search.

FAST$N turns out to be a short four-line function which uses NEXT (the function called by DOLF when it is running in FAST mode), MATCH2, LASTN, and TCONC to do most of the work. By informing the translator that ($ n x) is to be translated into (FAST$N N X), we can write (($ 3 A) ($ 1 B) $) for our new rule. This would be translated into ((FAST$N 3 A) (FAST$N 1 B) (DOLF NIL NIL NIL)), and since each FAST$N would have three <ep> s, the parsing would be the same as that for the ($ $3 A $ $1 B $) case, so we could use the (1 5 4 2 C 7) format for both. The running time however would be decreased by fifty percent for the A W X Y Z A B C D E B C D example in the text earlier, and by many factors for longer strings. We could also generalize FAST$N to allow <form>'s and <mark>'s as inputs, and perhaps even to associate predicates with them. In this way we could construct a new full-fledged elementary pattern, and expand the FLIP system - in this case towards greater efficiency.

2.2.2 Longest Common Substring

One of the advantages FLIP has over COMIT and METEOR is the ability to perform searches that are not intrinsically left to right. This is done by means of the failure predicate. The example given below illustrates a fairly complex use of FLIP, and is included along with the LISP program to perform the same task. This LISP program required several hours to construct and debug as opposed to the twenty minutes for the FLIP rule. This is because there is almost a direct translation from the algorithm used by both the LISP program and the FLIP rule into the FLIP notation. However, the LISP program runs twice as fast as the FLIP program when it is compiled, although it is about eight times as slow as FLIP when it is interpreted.

The problem is to find the longest common substring of two strings of characters. For example, if one string is (A B C D E F G) and the other (C A B C E A B C F C D E F C D E F G F X), the longest common substring is (C D E F G) (it takes the FLIP rule .5 seconds to find this string). The algorithm which is used by both programs is as follows:

Begin from the left of one of the strings (e.g. at A) and examine the other string until this character is found. Determine how long is the common substring that originates here (in this case it is A B of length 2), and if longer than the previously longest substring (in this case there wasn't any), save the result. Continue looking for this character in the second string, noting any longer strings found (in this case A B C will also be found), until the end of the string is reached. Then take the next character in the first string and continue the process. (On this pass we will find no longer strings since there isn't any B C D, although there is a B C. On the third pass, C D E F is found followed by C D E F G which is ultimately the longest string.)

The construction of the LISP program for this algorithm is straightforward (for LISP programmers) although a little detailed. We will trace the construction of the FLIP rule from the algorithm.

First, it is obvious that we cannot write a FLIP match statement for this purpose which will succeed in matching, since it must continue looking for longer strings until it exhausts the string. (We could write a single match to look for a string longer than any given number, and then write a little LISP program which kept calling this FLIP rule, but this is not as elegant, or efficient, as the FLIP rule which we will construct below.) Therefore we will want to save the partial results on the alist and use a $*$ to get them out. Thus our rule will begin with $*$ / ((LAMBDA (WS PATT MATCH) (PROG2 (MATCH2 WS PATT MATCH) STRING))), where MATCH2 does the matching with the rest of the rule, and STRING is the name of the variable which will save the longest substring. We will initially set STRING to NIL and LENGTH to 1. LENGTH will be the length of the longest substring found so far, plus one.

It will be a little easier to write the FLIP rule if we assume the two strings of characters are strung together as one list separated by some special character not in either string such as ***. Our first attempt at the FLIP rule is:

```
     $*$ / ((LAMBDA (WS PATT MATCH) (PROG2
        (MATCH2 WS PATT MATCH)  STRING))
     ($SET STRING NIL)
     ($SET LENGTH (QUOTE 1))
     $ ($N (= LENGTH))  $   ***   $   2
```

Comments: the $N locates a string of length LENGTH, since we might as well not consider any strings shorter. However, when we find a substring in the second half equal to this by means of the <mark> 2, it is possible that this substring might extend even further. Remember in the example with (A B C D E F G *** C A B D E A B C F C D E F C D E F G F X) the first substring found was just (A), but it actually included (A B). Therefore we must include <ep>'s that account for this. Furthermore, we must make sure that the $N pattern does not include the ***. If it does, we have finished the search.

```
     $*$ / ((LAMBDA (WS PATT MATCH) (PROG2
          (MATCH2 WS PATT MATCH) STRING))
     ($SET STRING NIL)
     ($SET LENGTH (QUOTE 1))
     $ / / (YES 2)   ($N (= LENGTH)) / ((LAMBDA (X) (NOT (MEMBER
              (QUOTE ***) X))))      $    ***
     $    2    $    $1 / ((LAMBDA (X Y Z) (NOT (EQUAL (CAR X)
            (NTH  Z  (LENGTH Y)))))  -1  3)
```

Comments: the first $ will go on as long as the $N pattern matches. After that it will fail, which is what we want. The <mark> 2 after the *** finds a substring equal to the $N. The next $ initially matches the null segment. The $1 pattern matches with the first character not equal to the one in the corresponding position on the $ just before the ***. It thus marks the termination of the common substring. Note that the argument 3 ignores the $*$ and two $SETs in counting which <ep> is the $ before the ***.

When $1 matches, then the <mark> 2, plus the $ after it, gives the new substring. We must bind the new string and length on the alist and continue the search.

```
$*$ / ((LAMBDA (WS PATT MATCH) (PROG2
  (MATCH2 WS PATT MATCH) STRING))
($SET STRING NIL)
($SET LENGTH (QUOTE 1))
$ / / (YES 2)  ($N (= LENGTH)) / ((LAMBDA (X) (NOT
     (MEMBER (QUOTE ***) X)))) $  ***  $  2  $
$1 / ((LAMBDA (X Y Z) (NOT (EQUAL (CAR X)
 (NTH Z (LENGTH Y))))) -1  3)
 ($SET  STRING (= APPEND  -3  -2))
 ($SET LENGTH (= ADD1 (= LENGTH (= STRING))))
 $ / (NILL)
```

All that remains now is cleaning up a few details. First, there is the possibility that the longest substring might terminate at the end of the list, e.g. (A B C *** D B C). To keep this rule simple, let us also assume that the two strings are each terminated by special markers, e.g. (A B C * *** D B C **). In this way we avoid the boundary problem. Secondly, the last $ causes a failure, we must decide at what point we wish the search to resume. Going back to the first $ each time is obviously wrong because, in the event <mark> 2 does match, there may still be longer substrings starting at the same origin in the first string. However, there is no way for the $N pattern to resume the search. We must change the $N pattern into a $ pattern which matches a string of length LENGTH, and have it resume operating if <mark> 2 matches. The rule would thus be:

```
$*$ / ((LAMBDA (WS PATT MATCH) (PROG2
  (MATCH2 WS PATT MATCH) STRING))
($SET STRING NIL)
  ($SET LENGTH (QUOTE 1))
$ / / (YES 2)  $ / ((LAMBDA (X) (COND
  ((MEMBER (QUOTE *) X) NIL)
((LESSP (LENGTH X) LENGTH) (LIST NIL))
(T T) ))) / (YES 6)   $ / / (NO 4)   ***   $ / / (NO 6)
2   $ /    / (NO 8)
$1 / ((LAMBDA (X Y Z) (NOT (EQUAL (CAR X)
 (NTH Z (LENGTH Y))))) -1  3)
($SET STRING (= APPEND -3 -2))
($SET LENGTH (= ADD1 (= LENGTH (= STRING))))
$ / (NILL)
```

The above rule would now work in this form. However, it would not be as efficient as a similar rule using fast <doif>'s. The fast rule would even be simpler, in this particular case, because most of the failure predicates used above merely check to see whether or not the next <ep> has matched (the NO predicates). Since a fast $ does not release control until the next <ep> matches (assuming it is a <varf> or <doinf>), we could replace these failure predicates by predicates which _always_ yielded NIL, i.e. the function NILL. Then the fast $ would operate, matching itself and the next <ep>, release control, and then, if control ever returned to it, automatically fail.

In addition to replacing the various NO failure predicates by NILL, there is one other change that must be made to the above rule in order that it may run in FAST mode. Since the $1 that determines the end of the common substring uses the $ immediately preceding it as an input, we must replace the -1 by a 0, and use (CAR Y) in the predicate as per the special convention on page 27. Our final rule would therefore be:

```
$*$ / ((LAMBDA (WS PATT MATCH) PROG2 (MATCH2 WS PATT MATCH)
  STRING)))
($SET LENGTH (QUOTE 1))
($SET LENGTH (QUOTE 1))
$ / / (YES 2)
$ / ((LAMBDA (X) (COND
((LESSP (LENGTH X) LENGTH) (LIST NIL))
((MEMBER (QUOTE *) X) NIL)
(T )T ))) / (YES 6)
$ / / (NILL)   ***   $ / / (NILL)  2   $ / / (NILL)
$1 ((LAMBDA (X Y Z) (NCT (EQUAL (CAR X)
   (NTH Z (LENGTH (CAR Y)))))) 0 3)
($SET STRING (= APPEND -3 -2))
($SET LENGTH (= ADD1 (= LENGTH (= STRING))))
$ / ( NILL)
```

We have included below a printout of the LISP function which performs the same operation as the rule above. By way of comparison, it runs about twice as fast as the above rule, which in turn runs five times as fast as the rule on the previous page which did not utilize fast <doif>'s. We also include a selective trace of the fast rule as an aid in visualizing it operates. The tracing mechanism has been turned on for the first <doif> following the *** <ep>, and for the final <doif> which induces a failure.

In this trace, we can see what the proposed common substring is by looking at the second <ep>. This is what the <mark> 2 will try to match. Initially, this is the string (A). When the <mark> 2 does find an (A), the next <ep> corresponds to the rest of the common substring originating at this point; here it is (B). Note that STRING is therefore bound to (A B). At this point, control goes back as far as the second <doif> which then matches with (A B C). (A B C) is also found by <mark> 2, but here the next <ep>

```
(LONGSUB
   (LAMBDA (A B) (PROG (M N W X D E C)
               (SETQ N 0)
               (SETQ X NIL)
         T1    (SETQ C B)
         T1A   (SETQ D A)
               (SETQ E C)
               (COND
                  ((NULL D) (RETURN (REVERSE X))))
         T3    (COND
                  ((EQUAL (CAR D) (CAR E)) (GO T2)))
               (SETQ E (CDR E))
               (COND
                  ((NULL E) (GO T5)))
               (GO T3)
         T2    (SETQ W (LIST
                  (CAR D)))
               (SETQ M 1)
         T4    (SETQ D (CDR D))
               (SETQ E (CDR E))
               (COND
                  ((NULL D) (GO T6))
                  ((NULL E) (GO T6))
                  ((NULL (EQUAL (CAR D) (CAR E))) (GO T6)))
               (SETQ W (CONS (CAR D) W))
               (SETQ M (ADD1 M))
               (GO T4)
         T6    (COND
                  ((GREATERP M N) (PROG2
                     (SETQ N M)
                     (SETQ X W))))
               (SETQ C (CDR C))
               (COND
                  ((NOT (NULL C)) (GO T1A)))
         T5    (SETQ A (CDR A))
               (GO T1))))
NIL
```

A LISP program for finding the longest common substring

of two strings - compare with FLIP rule on page 37.

```
DOLF
MATCH         (NIL (A) (B C D E F G *) (***))
WS            (C A B D E A B C F C D E F C D E F G F X *)
ALIST         ((G02274) (LENGTH . 1) (STRING))


DOLF
MATCH         (NIL (A) (B C D E F G *) (***) (C) (A) (B) (D))
WS            (E A B C F C D E F C D E F G F X *)
ALIST         ((G02274) (LENGTH . 3) (STRING A B))


DOLF
MATCH         (NIL (A B C) (D E F G *) (***))
WS            (C A B D E A B C F C D E F C D E F G F X *)
ALIST         ((G02274) (LENGTH . 3) (STRING A B))


DOLF
MATCH         (NIL (A B C) (D E F G *) (***) (C A B D E) (A B C) NIL
(F))
WS            (C D E F C D E F G F X *)
ALIST         ((G02274) (LENGTH . 4) (STRING A B C))


DOLF
MATCH         (NIL (A B C D) (E F G *) (***))
WS            (C A B D E A B C F C D E F C D E F G F X *)
ALIST         ((G02274) (LENGTH . 4) (STRING A B C))


DOLF
MATCH         ((A) (B C D E) (F G *) (***))
WS            (C A B D E A B C F C D E F C D E F G F X *)
ALIST         ((G02274) (LENGTH . 4) (STRING A B C))


DOLF
MATCH         ((A B) (C D E F) (G *) (***))
WS            (C A B D E A B C F C D E F C D E F G F X *)
ALIST         ((G02274) (LENGTH . 4) (STRING A B C))


DOLF
MATCH         ((A B) (C D E F) (G *) (***) (C A B D E A B C F) (C D
E F) NIL (C))
WS            (D E F G F X *)
ALIST         ((G02274) (LENGTH . 5) (STRING C D E F))


DOLF
MATCH         ((A B) (C D E F G) (*) (***))
WS            (C A B D E A B C F C D E F C D E F G F X *)
ALIST         ((G02274) (LENGTH . 5) (STRING C D E F))


DOLF
MATCH         ((A B) (C D E F G) (*) (***) (C A B D E A B C F C D E
F) (C D E F G) NIL (F))
WS            (X *)
ALIST         ((G02274) (LENGTH . 6) (STRING C D E F G))
```

matches with NIL since the character following (A B C) is F in the second half and D in the first. When the final <dolf> induces a failure, STRING is (A B C) and LENGTH 4. Again control goes back only as far as the second <dolf> which extends its segment to length 4, now matching (A B C D). However, <mark> 2 does not find this, and so control goes back to the first <dolf> which then matches with (A). (We are now midway down the page.) The second <dolf> now matches with (B C D E), which <mark> 2 again cannot find. The first <dolf> then matches with (A B), and the second <dolf> with (C D E F). This time <mark> finds it (the first C D E F), terminated by a C. The final <dolf> passes control back to the second <dolf> with STRING C D E F and LENGTH 5. This <dolf> matches with (C D E F G), which is also found by the <mark> (the second C D E F). When the final <dolf> fails and passes control back to the second <dolf>, it tries to match with (C D E F G *), and its abort predicate gives a value of NIL. Since this <dolf> does not match, the first <dolf> also fails, by virtue of its FAILURE predicate, and the value of STRING is then the value of the match, and appears at the bottom of the trace.

2.3 Translation

The purpose of making the translation process distinct from the match itself is to avoid the necessity of interpreting a FLIP rule each time it is executed, and thereby duplicating a considerable effort. Translation performs this interpretation only once, and saves the results of its labor by physically altering the list structure of its input. Whenever a FLIP rule is a constant, e.g. the call to FLIP is of the form (FLIP WS (QUOTE S-expression) (QUOTE S-expression)), this procedure saves much time because translation, and therefore interpretation, is necessary only the first time the rule is executed. Subsequently, the translator notes the fact that this S-expression represents a translated pattern.

When a FLIP rule is constructed within a LISP program and then executed, e.g. the call to FLIP is of the form (FLIP WS (LIST X Y) (LIST P Q)), translation does not cost the user, even though it will have to be performed each time, because an interpretation process of some sort must take place anyway. Translation merely saves the intermediate results, although, in this latter case, they may never be called upon again.

Once a FLIP rule has been converted into its translated form, the operation of MATCH is straightforward. Each individual <ep> of the pattern corresponds to a function call and its arguments. When one <ep> matches a segment and has finished operating, it calls the next <ep> by means of the LISP function APPLY, giving it the modified WS, PATT, and MATCH, plus its own individual arguments.

One consequence of this procedure is that it is extremely simple to introduce new operations. All that is necessary is to define the corresponding <ep> function, since LISP APPLY is completely general, and to inform the translator how to handle the notation in source language. In the next section, we discuss how this is done and introduce the $Bn pattern as an example.

The translating functions are all described in section 2.5. There are two main functions, PATTRAN and PATTRANSPK. The inputs to PATTRAN and PATTRANSPK are assumed to be <pattern>'s of syntactic type xxx. Both functions return the translated version of the input pattern, as well as altering the list structure of the input to correspond to it. Thus, if the input pattern is ($ $1 $ 2 $), the value of PATTRAN or PATTRANSPK is ($PATTERN ((DOLF NIL NIL NIL) (DOLNF 1 NIL) (DOLF NIL NIL NIL) (VARF *T* (MARK /T 2) NIL) (DOLF NIL NIL NIL))). After translation, the pointer to the list structure corresponding to ($ $1 $ 2 $) points to ($TRAN $PATTERN ((DOLF NIL NIL etc.). The special atom $TRAN is a signal to the translator that the following expression has been translated.

The only difference in the operation of PATTRAN and PATTRANSPK occurs when SPEAK is *T*. In this event, both give output regarding the translation. PATTRAN gives an abbreviated form of the translation consisting merely of a list of the names of the <ep>'s in the input pattern. PATTRANSPK prints the entire translation. Output produced by PATTRAN and PATTRANSPK for the longest common substring rules in the previous section is given at the end of this section.

In the definitions below, we have adopted the convention of using the symbol * to denote translation, e.g. <sexp>* is the internal representation of <sexp>. The translations of each of the 46 syntactic types defined in section 2.1.2 are listed by their corresponding numbers. Note that QUOTE and FAST modes affect the translation process, the former in conjunction with the interpretation of types xxx and vi, and the latter with regard to the <doif>'s.

```
            i. (not applicable)

            ii. <var> = <sexp>; <var>* = <sexp>*

            iii. <sexp> = <atom>; <sexp>* = (SEXP <atom>)

            iv. <sexp> = (QUOTE <S-expression>);
                <sexp>* = (SEXP <S-expression>)

            v.  <sexp> = <nonlist>; <sexp>* = (SEXP <nonlist>)

            vi. <sexp> = <list> whose  first  member  is  not  $SET
            etc.;
                <sexp>* = (SEXP <list>)

            vii. <var> = <form>; <var>* = <form>*

            viii. <form> = (= <S-expression>);
                  <form>* = (FORM <S-expression>)

            ix. <form> = (= <fn> <var> ... <var>);
                <form>* = (FORM <fn> <var>* ... <var>*)

            x.  <var> = <mark>; <var>* = <mark>*

            xi. <mark> = <number>; <mark>* = (MARK /T <number>)

            xii. <mark> = (<number> <mark1> ... <mark1>);
                 <mark>*   =   (MARK   /T   <number>  <mark1>*  ...
            <mark1>*), same as xiv

            xiii. <mark1> = <number>|<var>;
                  <mark1>* = <number>|<var>*
```

Since the value of a <mark1> is constrained to be a number, the <var> must of necessity be a <form>. The translator detects this and causes an error if it is not.

xiv. <mark> = (/T <mark1> ... <mark1>);
<mark>* = (MARK /T <mark1>* ... <mark1>*)

xv. <mark> = (/C <mark1> ... <mark1>);
<mark>* = (MARK /C <mark1>* ... <mark1>*)

xvi. <mark> = (/U <mark1> ... <mark1>);
<mark> = (MARK /U <mark1>* ... <mark1>*)

xvii. (not applicable)

xviii. <ep> = <varp>; <ep>* = <varp>*

xix. <varp> = <sexp>; <varp>* = (VARF NIL <sexp>* NIL)

xx. <varp> = <form>; <varp>* = (VARF NIL <form>* NIL)

xxi. <varp> = <mark>; <varp>* = (VARF *T* <mark>* NIL)

xxii. <varp> = (* <var>); <varp>* = (VARF NIL <var>* NIL)

xxiii. <varp> = (** <var>); <varp>* = (VARF *T* <var>* NIL)

We use NIL to indicate an item and *T* a segment.

xxiv. <ep> = <dol>; <ep>* = <dol>*

xxv. <dol> = $; <dol>* = (DOLF NIL NIL NIL)

If FAST is *T* at translation time, all <dolf>'s and <dol>'s are treated as fast. In this case xxv translates to (DOLF NIL NIL *T*), and is the same as the <dolf> $ // NIL.

xxvi. <ep> = <doln>; <ep>* = <doln>*

xxvii. <doln> = $n; <doln>* = (DOLNF n NIL)

xxvii. <doln> = ($N <var>); <doln>* = (DOLNF <var>* NIL)

xxix. <ep> = <pattern>; <ep>* = <pattern>*

```
xxx. <pattern> = (<ep> ... <ep>);
     <pattern>* = (PATTERN NIL (SEXP ($TRAN $PATTERN
(<ep>* ... <ep>*))))
```

Here <pattern> is an <ep> inside of another <pattern>. The value of the <sexp> is ($TRAN $PATTERN (<ep>* ... <ep>*)), so that when the <ep> function evaluates it and gives it to PATTRAN, it gets back the appropriate value.

At the top level, the input <pattern> is treated differently. Its translation is ($PATTERN (<ep>* ... <ep>*)). Compare this with xxx above.

```
xxxi. <pattern> = ($* <var>);
      <pattern>* = (PATTERN NIL <var>*)
```

Note that if QUOTE is NIL the second <ep> in ($ ($ F $) $) and the second <ep> in ($ ($* ((QUOTE ($ F $))) $) will match the same segments. However, the former will translate into (PATTERN NIL (SEXP ($TRAN $PATTERN ((DOLF NIL NIL NIL) (VARF NIL (SEXP F) NIL) (DOLF NIL NIL NIL))))) and the latter to (PATTERN NIL (SEXP ($ F $))). However, after both are executed once, their internal representations will be the same since the ($ F $) in the latter <ep> is translated in the process of execution, and therefore will be changed into the former representation.

```
xxxii. <pattern> = ($** <var>);
       <pattern>* = (PATTERN *T* <var>*)

xxxiii. <npred> = (<fn>)|(<fn> <var> ... <var>);
        <npred>* = (<fn>)|(<fn> <var>* ... <var>*)

xxxiv. <ep> = <varf>; <ep>* = <varf>*

xxxv. <varf> = <varp> / <npred>;
      <varf>* = (VARF NIL <var>* <npred>*) or (VARF *T*
<var>* <npred>*), depending on whether the <varp> was a
NIL or *T* type, i.e. item or segment.
```

We have indicated previously that when an <npred> is NIL its effects effect is the same as (LAMBDA (X) T), that is, the <ep> matches anything it would without a predicate. Here we see why. The <varf> with an <npred> that is NIL translates into the same form as a <varp>, i.e. a <varf> without an <npred>. The same will be true for <dolf> vs <dol> and <doinf> vs <doin>. Whenever an <npred> is missing, or is present but NIL, the <ep> degenerates into its simpler counterpart.

```
xxxvi. <ep> = <doinf>; <ep>* = <doinf>*

xxxvii. <doinf> = <doin> / <npred>;
        <doinf>* = (DOLNF n <npred>*) or (DOLNF  <var>*
<npred>*) depending on whether the <doin> is  a  $n  or
($N <var>)

xxxviii. <ep> = <doif>; <ep>* = <doif>*

xxxix. <doif> = <doi> / <npred>;
       <doif>* = (DOLF <npred>* NIL NIL)

xl. <doif> = <doi> // <npred>;
    <doif>* = (DOLF <npred>* NIL *T*)
```

If FAST is *T* at translation time, all <doif>'s and <doi>'s are translated as in fast mode. This means that the fourth member of the translation is *T*, as in xl above. Note that if FAST is *T*, $ by itself translates to (DOLF NIL NIL *T*).

```
xli. <doif> = <doi> / <npred1> / <npred2>;
     <doif>* = (DOLF <npred1>* <npred2>* NIL)

xlii. <doif> = <doi> // <npred1> / <npred2>;
      <doif>* = (DOLF <npred1>* <npred2>* *T*)
```

Again, setting FAST to *T* is the same as using // for the predicate marker.

```
xliii. <doif> = $ /  / <npred>;
       <doif>* = (DOLF NIL <npred>* NIL)

xliv. <doif> = <doi> //  / <npred>;
      <doif>* = (DOLF NIL <npred>* *T*)

xlv. <$set> = ($SET <atom> <var>);
     <$set>* = ($SET <atom> <var>*)

xlvi. <doll> = $*$ / <npred>;
      <doll>* = (DOLL <npred>*)
```

```
($*$ / ((LAMBDA (WS PATT MATCH) (PROG2 (MATCH2 WS PATT MATCH)
STRING))) ($SET LENGTH (QUOTE 1)) ($SET STRING NIL) $ / / (
YES 2) $ / ((LAMBDA (X) (COND ((LESSP (LENGTH X) LENGTH) (LIST NIL))
((MEMBER (QUOTE *) X) NIL) (T T)))) / (YES 6) $ / / (NO 4)
*** $ / / (NO 6) 2 $ / / (NO 8) $1 / ((LAMBDA (X Y Z) (NOT
(EQUAL (CAR X) (NTH Z (LENGTH Y))))) -1 3) ($SET STRING (=
APPEND -3 -2)) ($SET LENGTH (= ADD1 (= LENGTH (= STRING))))
$ / (NILL))
(DOLL $SET $SET DOLF DOLF DOLF SEXP DOLF MARK DOLF DOLNF $SET
$SET DOLF)
```

Output from PATTRAN when SPEAK is *T*
The first list is the input rule - taken from page 36.
Following it is a list of the top level <ep> functions.
The value of PATTRAN is the translation of the rule.

```
($*$ / ((LAMBDA (WS PATT MATCH) (PROG2 (MATCH2 WS PATT MATCH)
STRING))) ($SET LENGTH (QUOTE 1)) ($SET STRING NIL) $ / / (
YES 2) $ / ((LAMBDA (X) (COND ((LESSP (LENGTH X) LENGTH) (LIST NIL))
((MEMBER (QUOTE *) X) NIL) (T T)))) / (YES 6) $ / / (NILL)
*** $ / / (NILL) 2 $ / / (NILL) $1 / ((LAMBDA (X Y Z) (NOT
(EQUAL (CAR X) (NTH Z (LENGTH (CAR Y)))))) 0 3) ($SET STRING
(= APPEND -3 -2)) ($SET LENGTH (= ADD1 (= LENGTH (= STRING))))
$ / (NILL))
(DOLL ((LAMBDA (WS PATT MATCH) (PROG2 (MATCH2 WS PATT MATCH)
STRING))))
($SET LENGTH (SEXP 1))
($SET STRING (SEXP NIL))
(DOLF NIL (YES (MARK /T 2)) *T*)
(DOLF ((LAMBDA (X) (COND ((LESSP (LENGTH X) LENGTH) (LIST NIL))
((MEMBER (QUOTE *) X) NIL) (T T)))) (YES (MARK /T 6)) *T*)
(DOLF NIL (NILL) *T*)
(VARF NIL (SEXP ***) NIL)
(DOLF NIL (NILL) *T*)
(VARF *T* (MARK /T 2) NIL)
(DOLF NIL (NILL) *T*)
(DOLNF 1 ((LAMBDA (X Y Z) (NOT (EQUAL (CAR X) (NTH Z (LENGTH
(CAR Y)))))) (MARK /T 0) (MARK /T 3)))
($SET STRING (FORM APPEND (MARK /T -3) (MARK /T -2)))
($SET LENGTH (FORM ADD1 (FORM LENGTH (FORM STRING))))
(DOLF (NILL) NIL *T*)
NIL
```

Output from PATTRANSPK when SPEAK is *T*
The first list is the input rule - taken from page 37.
Following it is the complete translation with each <ep>
on a separate line. The value of PATTRANSPK is the translation.

2.4 Modes of Operation

FLIP has five modes of operation. These correspond to five variables which may be set to *T* or NIL: SPEAK, FAST, QUOTE, TRACE, and EDIT. Their effect on the match is described below. QUOTE, SPEAK, and EDIT also affect the construct process.

If SPEAK is NIL, PATTRAN and PATTRANSPK do not print. Otherwise PATTRAN prints out the input pattern and the list of the names of all of the top level <ep>'s. PATTRANSPK prints out the input pattern and a complete listing of the translation. This is useful for debugging.

SPEAK is initially set to NIL.

If FAST is NIL, <dol> and all <dolf>'s will be run in the normal way, except where // is used in a <dolf>. If FAST is *T*, all <dolf>'s will be treated as fast, i.e. as though written with //. Moreover, if FAST is *T* at translation time, all <dolf>'s in the input pattern will be run in fast mode regardless of what the value of FAST is when the rule is executed. Thus the only way to have a <dolf> operate in non-fast mode is to have FAST set to NIL both at translation time and run time, and to use a single / with predicates.

FAST is initially set to *T*.

If QUOTE is NIL, all lists not otherwise identified will be interpreted as <pattern>'s, i.e. syntactic type xxx. If QUOTE is *T*, all lists not otherwise identified will be interpreted as <sexp>'s, i.e. syntactic type vi.

Thus if QUOTE is NIL, ($ (A (B C)) $) becomes

```
  ($PATTERN ((DOLF NIL NIL NIL) (PATTERN NIL (SEXP ($TRAN
     $PATTERN ((VARF NIL (SEXP A) NIL) (PATTERN NIL (SEXP ($TRAN
         $PATTERN ((VARF NIL (SEXP B) NIL) (VARF NIL (SEXP C) NIL))
      )))  )))   (DOLF NIL NIL NIL)) )
```

and ($ ($1 $ 1) $) becomes

```
  ($PATTERN ((DOLF NIL NIL NIL) (PATTERN NIL (SEXP ($TRAN
     $PATTERN ((DOLNF 1 NIL) (DOLF NIL NIL NIL)
     (VARF *T* (MARK /T 1) NIL))  ))) (DOLF NIL NIL NIL)) )
```

If QUOTE is *T*, ($ (A (B C)) $) becomes

```
  ($PATTERN ((DOLF NIL NIL NIL) (VARF NIL (A (B C)) NIL)
     (DOLF NIL NIL NIL)) )
  ($PATTERN ((DOLF NIL NIL NIL) (VARF NIL (A (B C)) NIL)
     (DOLF NIL NIL NIL)) )
```

and ($ ($1 $ 1) $) becomes

```
     ($PATTERN ((DOLF NIL NIL NIL) (VARF NIL (SEXP ($1 $ 1)) NIL)
        (DOLF NIL NIL NIL)) )
```

Note that the top level pattern is always treated as xxx type.

QUOTE is initially set to *T*.

TRACE is used to follow the operation of match more closely. If TRACE is *T*, the value of WS, MATCH, ALIST, and name of the current <ep> are printed before each elementary pattern is entered. If TRACE is (DOLNF $SET), tracing will be performed only before each DOLNF or $SET is entered. If TRACE is (2 5), it will be performed whenver there are two or five functions left to be executed in PATT. One can mix function names and numbers, e.g. if TRACE is (3 DOLNF), then for PATT ($ A $ $2 B $ 4 $), tracing will occur before the third $ and the $2 are executed.

If EDIT is NIL, everything proceeds as discussed above. If EDIT is *T*, all <varf>'s that are nonatomic <sexp>'s are flattened and treated as segments, e.g. (A (B C)) becomes (VARF *T* (SEXP (L* A L* B C R* R*)) NIL). The EDIT mode also affects the operation of the construct process. It is useful for manipulating list structure as text. See the discussion of $Bn pattern in section 2.5, and Mac Memo 264.

EDIT is initially set to NIL.

2.5 Operation of Match

2.5.1 Definition of Functions in MATCH

Note: all functions described below are compiled.

2.5.1.1 General Functions

TCONC (X P)
The effect of (TCONC X P) is similar to that of (NCONC P (LIST X)), that is X is physically attached to the end of the list P. However, TCONC is designed so that it does not have to search for the end of the list P each time. This is done by making its input be a pointer in which (CAR P) is the list being changed, and (CDR P) is the end of the list. TCONC attaches X at (CDR P) using RPLACD, and returns with a new pointer in which the value of the list (now with X attached to the end) is again (CAR P) and (CDR P) points to the new end of the list, i.e. where X is. One can start with P as NIL, in which case the value of (TCONC X NIL) is ((X) X), ready for future input to TCONC, or with P as (NIL), in which case TCONC still returns with ((X) X), but physically alters its input. The latter usage has the advantage that (CAR P) is <u>always</u> the list, regardless of whether or not TCONC was ever entered (since in this case it gives NIL), and furthermore, that it is not necessary to do a SETQ each time to save the value of TCONC as initializing a variable to (NIL) will suffice, because TCONC will then make all changes upon the value of the variable itself.

LCONC (X P)
LCONC is similar to TCONC except that X is a list (or else an error occurs) and the two lists are tied together. The value of LCONC is the new P. As in TCONC, P may be NIL or (NIL).

Examples: If P is ((A B C) C), then (TCONC (QUOTE D) P) is ((A B C D) D), and (LCONC (QUOTE (D E F)) P) is ((A B C D E F) F).

FIRSTN (L N)
(FIRSTN L N) is CONS of a list consisting of the first n elements of L with the rest of L. Example: (FIRSTN (QUOTE (A B C D E)) 3) is ((A B C) D E). If L is NIL or N is too large, the value of FIRSTN is NIL. If N is not a number, an error occurs. (N may be zero, in which case (FIRSTN L N) is (LIST L). If N is negative, the program will loop indefinitely.)

LASTN (L N)
LASTN is similar to FIRSTN. (LASTN L N) is CONS of all of L <u>up to</u> the last n objects with the last n. Example: (LASTN (QUOTE (A B C D E)) 3) is ((A B) C D E). Both FIRSTN and LASTN use TCONC.

LAST (X)
If X is an atom, (LAST X) is NIL. If X is not an atom, (LAST X) is the last construct in X. Example: If X is (A B C D), (LAST X) is (D). If X is (A . (B . (C . D))), (LAST X) is (C . D).

LISTP (X)
(LISTP X) is *T* if X is a list or NIL. it uses LAST.

NILL ()
NILL is (LAMBDA NIL NIL).

SAME (X)
SAME is (LAMBDA (X) X).

YES (X)
YES is (LAMBDA (X) (NULL (EQ X (QUOTE FAILED)))).

NO (X)
NO is (LAMBDA (X) (EQ X (QUOTE FAILED))).

NEQUAL (X Y)
NEQUAL is (LAMBDA (X Y) (NULL (EQUAL X Y))).

WAPPLY (FN ARGS A)
WAPPLY is an EVALQUOTE that also works for FSUBRS. If FN is an EXPR or SUBR, it performs (APPLY FN ARGS A). If FN is an FEXPR, it performs (EVAL (CONS FN ARGS) A). If FN is an FSUBR, since all FSUBRS except QUOTE evaluate their arguments after they are called, it performs (MAPLIST ARGS (LAMBDA (X) (LIST (QUOTE QUOTE) (CAR X)))), and then performs (EVAL (CONS FN ARGS) A). It checks specially for YES, NO and NILL, and does not go through the interpreter in these cases. Also, FN may be DO, in which case it assumes the <u>second</u> ARG is the function (the first ARG would normally be the <ep>). Thus (LAMBDA (X1 ... Xn) (FN X2 ... Xn)) may be written as (DO FN X2 ... Xn). For example: one may write $1 / ((LAMBDA (X Y Z) (MEMBER (CAR Y) Z)) 2 3) as simply $1 / (DO MEMBER (= CAR 2) 3).

2.5.1.2 Translation Functions

All of the translation functions described below have careful error controls with hopefully illuminating diagnostic comments. Furthermore, they are straightforward enough so that the user can readily change them should he wish to alter or augment the conventions in a way other than that provided for by PATTRAN3.

PATTRANSPK (PATT)
Translates and alters list structure of PATT. If SPEAK is *T*, it prints the translation.

PATTRAN (PATT)
Translates and alters list structure of PATT. If SPEAK is *T*, it prints an abbreviated version of the translation.

PATTRAN1 (PATT)
Performs translation of PATT.

PATTRAN2 (PATT)
Translates next <ep> in PATT, returns CONS of translation with rest of PATT. Uses PATTRAN3. Note that next <ep> may include up to five members of the list PATT, e.g., in the case of $ / <npred> / <npred>.

PATTRAN3 (X)
If X is an atom, PATTRAN3 calls ATOMTRAN. If X is not a list, it translates as an <sexp>. If X begins with $SET, $*, $**, $N, QUOTE, = *, **, /C, /T, /U, or a number, PATTRAN3 translates it into the appropriate form as per section 2.3. Otherwise it looks on the property list of $TRAN under the property (CAR X). If (GET (QUOTE $TRAN) (CAR X)) is not NIL, it returns (APPLY (GET (QUOTE $TRAN) (CAR X)) (LIST X) NIL), i.e. it uses the value on the property list as a function which performs the translation. (This is the way one introduces new conventions.) If there is no indicator on the property list of $TRAN under (CAR X), X is translated as either a <pattern> or an <sexp>, and PATTRAN3 uses QUOTE to determine which.

NPREDTRAN (PATT PRED)
Translates the <npred> PRED. PATT is used for error diagnostics.

ATOMTRAN (X)
Translates single atom X as an <sexp> if X is not $, $*$, $n, or a number.

VARTRAN (VAR)
Calls PATTRAN3 on VAR. If the result is not an <sexp>, <form>, or <mark>, it gives an error.

NUMTRAN (X L)
L is a list of digits. NUMTRAN uses PACK and NUMOB to get the corresponding number. If it encounters a nonnumerical object, it gives an error diagnostic with X. Used for $n <ep>, which has to

be unpacked to see the $ character at the front.

MARKTRAN (X L)
L is a list of <mark1>'s. MARKTRAN translates them sequentially. Recall that each <mark1> must either be a number or have a number for its value. Therefore if any member of L is neither a number nor a <form>, MARKTRAN gives an error.

2.5.1.3 MATCH functions

MATCH (WS PATT)
MATCH is (LAMBDA (WS PATT) (MATCH1 WS PATT NIL)).

MATCH1 (WS PATT $A)
MATCH1 is the top level function for the matching operation. However, since PATT is essentially a list of function calls, once the match begins operating it does not require supervision. Thus the only purpose of the top level function MATCH1 is to set certain variables that are SPECIAL throughout the operation of the match, $A $TR and $MATCH, and then initiate the match. These variables may be accessed by means of the function SPESHUL.

$A is one of the inputs to MATCH1 and is the alist used throughout the match. In order to insure the existence of a nonempty alist so that other functions, e.g. $SET and $SET1, may alter it, if $A is NIL, MATCH1 replaces it by ((GENSYM)).

$TR is set to the value of TRACE, described in section 2.4, and is used by MATCH2.

$MATCH is used to store higher level parsings whenever a <pattern> is entered. In the example on page 16 and 17, $MATCH would consist of all but the last parsing, ($MATCH ((O P Q)) (O) (P)). It is NIL whenever the match is operating on the top level.

MATCH1 verifies that (CAR PATT) is $PATTERN and then calls (MATCH2 WS PATT ($MATCH (WS))). If match succeeds, it returns the parsing, otherwise it returns NIL.

SPESHUL (X)
If X is $A, $MATCH, $TR, $E, or $F, (SPESHUL X) is the value of X. Otherwise, SPESHUL gives an error diagnostic.

MATCH2 (WS PATT MATCH)
MATCH2 performs the linkage between elementary patterns. If WS is NIL, and PATT is also NIL, the match terminates successfully and MATCH2 returns MATCH. If WS is NIL and PATT is not, i.e. there

are still some <ep>'s left to be matched, the match fails and MATCH2 returns (LIST MATCH). Otherwise, MATCH2 performs (APPLY (CAAR PATT) (APPEND (LIST WS (CDR PATT) MATCH) (CDAR PATT)) NIL). However, MATCH2 does not use APPLY for DOLF, VARF, DOLNF, $SET, PATTERN, and DOLL as it checks for them specially so that it does not have to go through the LISP interpreter.

Tracing is also performed in MATCH2. If $TR is *T*, tracing occurs each time MATCH2 is entered and the name of the <ep>, WS, MATCH, and $A (the alist) are printed out. If $TR is NIL, no tracing occurs. Otherwise $TR may be a list containing names of functions, e.g. DOLF, $SET, or numbers which represent positions in PATT and hence in the rule. If (CAAR PATT), which will be the name of the function for the <ep> represented by (CAR PATT), is in this list, or if (LENGTH PATT) is in this list, tracing occurs. For the trace in section 2.2, TRACE, and hence $TR, was set to (1 7), indicating that the last elementary pattern, and the seventh from the last, were to be traced.

$SET (WS PATT MATCH NAME VAR)
Uses $SET1 to set NAME to (COPY (VARVAL VAR MATCH)) on $A and then continues with MATCH2.

$SET1 (VAR NEW ALIST)
If VAR appears on ALIST, it rebinds its first appearance to NEW. Otherwise it binds VAR to NEW at the end of the ALIST.

DOLL (WS PATT MATCH FN)
Uses WAPPLY with (CAR FN) and (PARGVAL (CDR FN)). FN is presumed to be the translation of an <npred>.

PARGVAL (PARGS MATCH)
Performs MAPLIST on PARGS, a list of predicate arguments, using VARVAL.

VARVAL (VAR MATCH)
Evaluates VAR. If VAR is a mark, it uses MATCH for current level parsing. Recall that VARVAL has access to the special cell $A for alist.

MARKVAL (MARK MATCH)
Evaluates MARK using MATCH as current level parsing. Also uses special cell $MATCH for higher level parsings. If $F is *T*, it returns FAILED instead of giving an error if an $<ep>^{-1}$ does not exist.

NUMVAL (X MATCH)
Used to evaluate a <mark1>. If X is a number, value is X. If X is not a number, calls (VARVAL X MATCH). If this is not a number, it gives an error.

MATCH3 (WS X)
Used by VARF. If the list X matches with a segment at the beginning of WS, MATCH3 returns (CONS X rest of WS), otherwise NIL. Example: WS is (A B C D E), X is (A B C), value is ((A B C) D E).

All of the <ep> functions, VARF, DOLF, DOLNF, and PATTERN, have a similar overall effect. They determine whether or not they match with a segment at the beginning of the WS; if not they return (LIST MATCH) to indicate failure. If successful, they proceed by calling MATCH2 giving it as arguments the modified WS, PATT, and MATCH with their <ep> appended to it.

VARF (WS PATT MATCH SEG VAR NPRED)
If SEG is *T*, VAR is treated as a segment, otherwise as an item. VARF calls VARVAL to evaluate VAR, MATCH3 to see if it matches WS, WAPPLY and PARGVAL to evaluate the NPRED if any, and then, if all indicate a match, returns (MATCH2 WS PATT MATCH) where WS has had the VARF segment removed from its front, and MATCH has had it appended at its end. If VARF fails to match, it returns (LIST MATCH).

DOLNF (WS PATT MATCH N NPRED)
Calls (NUMVAL N) to evaluate N, uses FIRSTN to get its segment, and WAPPLY and PARGVAL to evaluate the <npred>. If all indicate a match, it calls MATCH2 etc.

NEXT (DOL VAR WS)
NEXT is used by DOLF when it is operating in FAST mode. Its object is to find the next segment matched by VAR. If VAR is a number, i.e. if next <ep> after <dolf> is <dolnf> then NEXT returns (FIRSTN WS N). Otherwise, DOL is assumed to be a list in the TCONC format (it is used in DOLF to store the segment DOLF is currently matching), and NEXT searches WS until it finds a point where VAR matches. It TCONCs onto DOL any elements taken off the front of WS in this process. When it finds a match, it returns (CONS VAR rest of WS). Example: If DOL is ((A B C) C) and VAR is (G H), and WS is (D E F G H I J), the value of NEXT is ((G H) I J), and in the process, DOL is changed to ((A B C D E F) F).

DOLF (WS PATT MATCH NPRED1 NPRED2 //)
If PATT is NIL, DOLF only attempts a match with the rest of WS.

If either // or FAST is *T*, DOLF runs in FAST mode. If the next <ep> is not a <varf> or <dolnf>, it does not matter whether DOLF runs in FAST mode or not.

When DOLF is not in FAST mode, it first matches with NIL, and then continues to extend its segment by taking (CAR WS) and TCONCing it onto the previous segment. If there is an abort predicate (the NPRED1 argument), it applies this until it gets a value not NIL or (NIL). If it ever gets a (NIL) it fails and returns (LIST MATCH). If it succeeds in matching a segment, it calls (MATCH2 WS PATT MATCH) with new WS and MATCH with its segment appended, and if the result indicates a successful match, it passes this on; otherwise it sets $F to *T* and applies its failure predicate (NPRED2) if any, to decide whether or not to continue searching. If the decision is favorable, it extends its segment as above and the process goes on.

If DOLF is in fast mode, and the next <ep> is a <varf> or <dolnf>, DOLF uses NEXT to find a match for the next <ep>, and then applies the predicate for the next <ep>, and its own abort predicate. If the predicate for the next <ep> gives NIL, it looks for another match for it using NEXT again. If its own abort predicate gives NIL, it fails and returns (LIST MATCH). If its own abort predicate gives (NIL), it looks for another match using NEXT. If everything is favorable, it tries to find a match for the rest of the rule by calling MATCH2. If MATCH2 yields a successful match, it passes this on, otherwise applies its failure predicate, if any, and goes on with the search for a match for the next <ep> using NEXT. Note: in the interest of speed, DOLF only evaluates its PARGS, i.e. the inputs to its abort predicate, and the PARGS of the next <ep>, at the start of the search, and after each subsequent call to MATCH2. (In the latter case the alist may have changed.) Otherwise it knows that the arguments remain constant during the search for an acceptable match for the <dolf> and next <ep>, except for the special case where an argument, the special mark 0, refers to the segment matched by DOLF.

PATTERN (WS PATT MATCH SEG PATTLIST)
Calls VARVAL on PATTLIST, and translates the result. If SEG is NIL, it tries to match with (CAR WS). If match fails, returns (LIST MATCH), otherwise goes on using MATCH2. It tries the match with (CAR WS) by calling (MATCH2 (CAR WS) PATTLIST* ($MATCH ((CAR WS)))).

If SEG is *T*, PATTERN translates (VARVAL PATTLIST MATCH), and appends to this the function call (PATTERN1 $MATCH MATCH) followed by the rest of PATT. It sets $MATCH to (APPEND $MATCH (LIST MATCH)) and calls MATCH2 on WS, this new PATT, and ($MATCH WS). If a failure occurs, it returns LIST of the appropriate parsing. (If <pattern> was successful, but a failure occurred further on, its <ep>$^{\vee}$ will appear on this parsing. If a failure occurred inside of <pattern>, <pattern> will not have an <ep>$^{\vee}$, even if some of its <ep>'s did succeed.)

PATTERN1 (WS PATT MATCH OLDMATCH $**MATCH)
Resets $MATCH to OLDMATCH. $**MATCH is MATCH argument of PATTERN. PATTERN1 appends to this MATCH, first modifying (CADR MATCH) - it originally is set up as matching the entire WS - and calls MATCH2. If there is a failure, it sets $MATCH to (APPEND $MATCH result of failure) so that Failure Predicates inside of the <pattern> will work, and returns (LIST MATCH).

PATTERN and PATTERN1 act as an interface between the <ep>'s inside of a $** <pattern> and those on the same level outside of the $** <pattern>. PATTERN sets up $MATCH for the <ep>'s inside the $** so they can refer to previous results by means of /T or /U <mark>'s. PATTERN1 must restore $MATCH and group the <ep>$^{\vee}$s of the <ep>'s in <pattern> into a single parsing. If a failure occurs after PATTERN1, it must rest $MATCH before passing control back to the <ep>'s in the <pattern> so that their Failure Predicates may determine the caúse of failure. Similarly, if a failure occurs after PATTERN, it must restore $MATCH before passing control back to earlier <ep>'s.

Because of the operation of PATTERN and PATTERN1, ($ A ($** ($ $2)) B $) will match whatever ($ A $ $2 B $) matches - with a slightly different parsing. Also, one can write ($ A ($** ($ / / (NO (/U 1 4))) B C $), and the $ inside of the <pattern> will not continue searching if a B is found that is not followed by a C, exactly as for ($ A $ / / (NO 4) B C $).

2.5.2 Expanding FLIP

Because of the modularity of the FLIP system, it is extremely easy to introduce new operations. Since each <ep> performs its task and passes control onto the next, the <ep>'s described above can be intermixed with arbitrary <ep>'s introduced by the user. All the user need do is define a function and inform the translator how to handle the source language representation of the new <ep> (or else it will try to translate it into either an <sexp> or a <pattern>).

In this section we describe the development of an <ep> which was designed to facilitate the manipulation of list structure as if it were text. This <ep> is useful in conjunction with editing see MAC Memo 264, EDIT and BREAK Functions for LISP - and is in fact built into the current FLIP system.

2.5.2.1 Flattening Lists

Occasionally one would like to locate a particular structure in a list without concern for its depth. For example: if X is (A B (C D E) ((F G (M (N O)) P)) I J), if one is to refer to the (N O) one must write ($ (($ ($ (QUOTE (N O))) $)) $) to refer to the S-expression at the correct depth. The internal representation of LISP necessitates this sort of specification, since parentheses are not characters but structural symbols. However, if one considered X as a <u>linear</u> string of characters, locating (N O) would be trivial. Therefore, we define a function, FLATTEN, which transforms X into a linear string of characters (or atoms) substituting special atoms L* and R* for left and right parentheses respectively. This function is isomorphic in a sense to the function PRINT which takes a list structure and converts it into a linear string substituting the print-names for atoms and using the characters "(" and ")" to indicate depth. Thus (FLATTEN X) would be (L* A B L* C D E R* L* L* F G L* M L* N O R* R* P R* R* I J R*). Now to locate (N O) we need merely write ($ L* N O R* $). (-21-) We can then perform the necessary transformations and reconstruct the final structure using the function UNFLATTEN. Note that in particular we can delete and insert individual parentheses. Writing ($ L* N O R* $) for PATT and constructing with (1 L* L* N R* O R* -1) will change (N O) to

****

(-21-) The pattern ($ (** (L* N O R*)) $) will also locate (N O) and will run much faster since there is only one entry to VARF to determine whether or not a match with (L* N O R*) occurs where there are four for ($ L* N O R* $). Note that ($ (** (L* N O R*)) $) has three <ep>'s and ($ L* N O R* $) has six.

((N) O). (The CONSTRUCT feature will be discussed in detail in section 3.) In fact, if EDIT is *T*, we can simply write ($ (N O) $) and (1 ((N) O) 3) and FLIP will automatically flatten (N O) and ((N) O) during translation.

2.5.2.2 Balancing Parentheses

The above facilities are useful because they provide from within a LISP system the features of a context editor similar to the CTSS command "ED". We would like some additional sophistication for working with LISP; for example, we might like to be able to locate a structure by naming a substructure. Thus where X was (A B (C D E) ((F G (M (N O)) P)) I J) we want to be able to locate the structure (M (N O)) by specifying that it contains M, or (N O), or even just N. This is similar to asking FLIP to locate N and then back up until it finds two pairs of balanced parentheses.

The $Bn <ep> is designed to serve this purpose. In the interest of efficiency, since FLIP already has a FAST facility, we choose to have the $Bn pattern operate after the substructure has been located. Thus the fast $ could locate N and then ($BN 2) (since we want two pairs of parentheses) would say "I didn't really want to match with N but with the strcuture containing the structure containing N, so I must back up until I find two unmatched "("s and go forward until I find their corresponding ")"s, and then make the necessary changes in MATCH."

When $Bn is entered from the ($ N ($BN 2) $) pattern, the parsing is ($MATCH ((L* A B L* C D E R* ...)) (L* A B L* C D E R* L* L* F G L* M L*) (N)). After $Bn operates, it is ($MATCH ((L* A B L* C D E R* ...)) (L* A B L* C D E R* L* L* F G) (L* M L* N O R* R*)), and WS is (P R* R* I J R*). Note that the segment matched by N is gone and that that matched by $ has been reduced.

To introduce such an <ep> to FLIP, we defined a function DOLBN, of four variables, WS PATT MATCH N. DOLBN searches back through MATCH for N unmatched L*'s. If it cannot find them, it returns (LIST MATCH). If it finds them, it looks through WS for their mates. If these are found, it calls MATCH2 giving it the rest of WS after the matching R*'s, PATT, and MATCH, in which any elementary patterns that now have their segments included in that matched by $Bn no longer appear, and any portions of bordering segments included in the $Bn segment are deleted from the bordering elementary patterns. MATCH also has the segment matched by $Bn appended to it. Finally, we inform PATTRAN3 that ($BN 2) translates into (DOLBN 2) by putting on the property list of $TRAN under the property $BN the function (LAMBDA (X) (CONS (QUOTE DOLBN) (CDR X)). FLIP is now ready to operate with the $Bn pattern.

2.5.2.3 Editing Facilities now in FLIP

The functions DOLBN, FLATTEN, UNFLATTEN, and FPRINT are currently included in the FLIP package. The translation of $Bn is built into ATOMTRAN so one may write $B2 instead of ($BN 2) as above. UNFLATTEN is programmed to give an error if the parentheses do not balance, and a diagnostic relating to why they did not. FPRINT is similar to PRINT, but will work on unbalanced strings. The user can utilize this to print a flattened structure or segments of a flattened structure, without actually having to UNFLATTEN them. These functions used in EDIT mode allow the user to perform fairly sophisticated editing operations. MAC Memo 264 also describes some functions which use FLIP to perform editing of LISP functions. It should also be read by the user who will write his own editing functions, since it contains a much more detailed discussion of DOLBN, FLATTEN, UNFLATTEN, and the use of FLIP in an editing environment, and includes many illustrative examples.

3. The Construct

The purpose of the construct process is to construct a new list structure using a format and the parsing from a match. Since the flavor of the construct is very similar to that of the match, and in fact it uses many of the same functions as the match does, e.g. MARKVAL, VARVAL, etc., it is not necessary to go into it in as great detail as the previous section.

The inputs to the construct process are a parsing of the form ($MATCH <list> ... <list>), a format, and an alist. The format is a list of elementary formats, or <ef>'s, which are evaluated sequentially from left to right, their values being attached to the list structure under construction. The similarity to the matching process will be evident if we return to the COMIT example in the introduction:

```
        $ + $3 + A + $ + $1 + B + $  =  1 + 5 + 4 + 2 + C + 7
```

The FLIP pattern for the left side of this rule is ($ $3 A $ $1 B $). Similarly the format for the right side is (1 5 4 2 C 7). 1, 5, 4, 2, and 7 are all <mark>'s, and their value is computed with respect to the input parsing as described in section 2. C is an <sexp>; its value is itself, as described in section 2. However, instead of being <varp>'s, a type of <ep>, as they are when they appear by themselves in a <pattern>, they are <efvar>'s, which are a type of <ef>. However, as with <varp>'s, unless otherwise specified, an <efvar> that is a <mark> is treated as a segment, and the value of the <mark> is appended directly to the list structure being assembled. Similarly, an <efvar> that is a <sexp> is treated as an item so that (LIST <sexp>) is appended to the list structure being assembled.

If WS were (A W X Y Z A B C D E B C D) for the rule above, we saw that the parsing would be ($MATCH ((A W X Y ...)) (A W) (X Y Z) (A) (B C D) (E) (B) (C D)). The value of the <mark> 1 would be (A W). The value of <mark> 5 would be (E). The value of the <mark> 4 would be (B C D), and the value of <mark> 2 would be (C D). The value of <sexp> C would be C. The value of <mark> 7 would be (C D). Therefore the result of constructing with this parsing and this format would be (A W E B C D X Y Z C C D). Note that the <mark>'s were appended, and list of the <sexp> was appended, with the result that all were strung together at the same level.

3.1. Notation and Definitions

The definitions of such syntactic types as <sexp>, <form>, etc., which are contained in section 2.1.2 will not be repeated here: Please note that the <mark> is evaluated against the input parsing, and that negative <mark1>'s will therefore count backwards from the end of the parsing. Since the top level parsing is also the current level parsing, /U <mark>'s will give errors.

i. <ef> ::= <efvar>; <ef>^{v} ::= <efvar>^{v}

ii. <efvar> ::= <sexp>; same as (* <sexp>) in match, see xxii, p. 17

iii. <efvar> ::= <form>; same as (* <form>), see p. 17

iv. <efvar> ::= <mark>; same as (** <mark>)

v. <efvar> ::= (* <var>); <efvar>^{v} ::= (<var>^{v})

vi. <efvar> ::= (** <var>); <efvar>^{v} ::= <var>^{v}

If <var>^{v} is not a list, an error diagnostic occurs noting that an illegal segment has been used.

vii. <efvar> ::= <format>; <efvar>^{v} ::= <format>^{v}

viii. <format> ::= (<ef> ... <ef>);
<format>^{v} ::= same as ($* (QUOTE (<ef> ... <ef>)))

This assumes that QUOTE is NIL, otherwise (<ef> ... <ef>) is an <sexp> as in the case of a <pattern>.

ix. <format> ::= ($* <var>);
<format>^{v} ::= list of the result of treating <var>^{v} as a format in the sense above, i.e. treating each member of <var>^{v} as an <ef> to be evaluated sequentially and tied together.

x. <format> ::= ($** <var>);
<format>^{v} ::= the result of evaluating <var>^{v} as a <format> in the sense defined above.

Example: If WS is (QUOTIENT X (TIMES A B X Y Z)), PATT is (QUOTIENT $1 (TIMES $ 2 $)), and FORMAT is (TIMES (2 2) (2 -1)) the result is (TIMES A B Y Z).

3.2 Translation

The translation of an <ef> is similar to that of an <ep>. An <efvar> becomes (EFVAR NIL <var>*) or (EFVAR *T* <var>*) depending on whether the <var> is to be treated as an item or as a segment. A <format> becomes (FORMAT NIL <var>*) or (FORMAT *T* <VAR>*) depending on whether the <var> is to be treated as an item or as a segment. The functions FORMTRAN, FORMTRANSPK, FORMTRAN1, FORMTRAN3 are similar to PATTRAN, PATTRANSPK, PATTRAN1, PATTRAN3. There is no FORMTRAN2 since each <ef> is only one member of the input format. The similarity of FORMTRAN3 to PATTRAN3 extends to allowing the user to define new conventions by putting the function that performs the definition on the appropriate property of the atom $TRAN.

3.3 Modes of Operation

FAST and TRACE do not affect the construct process. SPEAK affects FORMTRAN and FORMTRANSPK in the same way as PATTRAN and PATTRANSPK. QUOTE is similarly used to decide whether an unidentified list is a <sexp> or a <format>. If EDIT is NIL, all proceeds as before. If EDIT is *T*, the value of each <efvar> and <format> is flattened before attaching it to the list structure being assembled. See memo on EDIT and BREAK Functions for LISP for more details of editing.

3.4 Functions in CONSTRUCT

FORMTRAN, FORMTRANSPK, FORMTRAN1, and FORMTRAN3 are analagous to their counterparts.

CONSTRUCT (MATCH FORMAT)
CONSTRUCT is (LAMBDA (MATCH FORMAT) (CONSTRUCT1 MATCH FORMAT NIL)).

CONSTRUCT1 (MATCH FORMAT $A)
Similar to MATCH1, Sets $A to ((GENSYM)) if NIL and calls CONSTRUCT2.

CONSTRUCT2 (MATCH FORMAT X)
X is in TCONC format. If FORMAT is NIL, returns (CAR X). Otherwise it does (APPLY (CAAR FORMAT) (APPEND (LIST MATCH (CDR FORMAT) X) (CDAR FORMAT)) NIL).

EFVAR (MATCH FORMAT X SEG VAR)
Does VARVAL of VAR, and if SEG is NIL, attaches this value to X by using TCONC, otherwise uses LCONC. If EDIT is *T*, it first flattens the value. Exits by calling CONSTRUCT2.

FORMAT (MATCH FORMAT X SEG VAR)
Uses (FORMTRAN (VARVAL VAR MATCH)) as a format and calls CONSTRUCT2 to perform the construction. If SEG is NIL, it treats result as item, and TCONCs it onto X, otherwise uses LCONC. If EDIT is *T* flattens result first. Exits by calling CONSTRUCT2.

4. Miscellaneous

4.1 Top Level Functions

FLIP (WS PATT FORM)
FLIP is (LAMBDA (WS PATT FORM), (FLIP1 WS PATT FORM (LIST (LIST (GENSYM))))).

FLIP1 (WS PATT FORM A)
If (MATCH1 WS (PATTRAN PATT) A) is successful, it saves this value and performs CONSTRUCT1 on it with (FORMTRAN FORM) and A, and returns CONS of this value with result of construction. Note that the same alist is used for both calls so that any variables set during the match will carry over to the construction. FLIP1 can be used with PATT equal to NIL, in which case it assumes WS is a parsing, and just calls CONSTRUCT1 on WS with (FORMTRAN FORM) and A and returns CONS of WS with the result of the construction. If FORM is NIL, it just calls MATCH1 on (PATTRAN PATT) WS and A, and if successful returns CONS of this with NIL, otherwise it returns NIL. Note that CAR of the value of FLIP1 is _always the parsing_, provided a successful match occurred, and _CDR is always the value of the construction_.

FLIPR (WS PATT FORM REP)
FLIPR is (LAMBDA (WS PATT FORM REP) (FLIPR1 WS PATT FORM REP NIL)).

FLIPR1 (WS PATT FORM REP A)
FLIPR1 is designed to repeat the application of a pattern and format to the WS for such problems as: remove all duplications from a list. The FLIP rule with PATT as ($ $1 $ 2 $) and FORMAT (1 2 3 5) would remove only the first duplication. We would like to reapply it to the segment matched by the second $ and last $. The inputs to FLIPR1 that will perform this are ($ $1 $ 2 $), (1 2), and (3 5). A match would be attempted with PATT ($ $1 $ 2 $). If successful, a CONSTRUCT with (1 2) would occur. This value would be saved. Then a CONSTRUCT with (3 5) would be performed and another match with this and ($ $1 $ 2 $) would take place. This would continue until the match failed. Then the results of all the constructions together with the WS when the pattern failed would be strung together. CONS of the number of times a succesful match occurred with this list structure would then be returned.

Example: If WS is (A B C D B C E A F), PATT is ($ $1 $ 2 $), FORMAT is (1 2), and REP is (3 5), the result will be (3 A B C D E F).

4.2 Loading FLIP

FLIP is contained in files FLIP DATA and FLIP1 DATA through FLIP8 DATA. All are in READ ONLY, LINKABLE, PROTECT mode in my files, t272 3515. You may load and compile all of these functions by loading FLIP DATA (computation time: approximately 60 seconds). The file FLIP SAVED is also in READ ONLY, LINKABLE, PROTECT mode in my files and contains FLIP already loaded and compiled.