A I Memo 76
February 18, 1965

MAC Memo 219
LISP II Project Memo 2

The COMIT Feature in LISP II

Daniel G. Bobrow

The COMIT Feature in LISP II

I. Purpose

The purpose of the COMIT feature is to facilitate certain types of list manipulations in LISP II. This feature is a syntactic convenience, rather than an extension of the semantics of LISP. It permits the programmer to test directly whether a piece of list structure matches a certain pattern, and if so, to construct another structure utilizing subsegments of the original structure which matched parts of the given pattern.

II. The Match and Construct Interpreters

The COMIT feature can be implemented by programming two interpretive functions in LISP,

match [pattern; workspace]

and construct [format; array]

The match interpreter has two arguments, a pattern of the type described below, and a workspace which is the list which is to be matched against the pattern. If the match fails, the value of the function is NIL. If it succeeds, the value is a symbolic array which gives the segmentation of the workspace which allowed a match.

The construct interpreter uses a format, to be described below, and an array which is an output of a successful match. It constructs a new structure according to this format.

XII. Patterns

A pattern is a list of elementary patterns. In the source language, a ⟨pattern⟩ is of the following form

(⟨ep⟩ ⟨ep⟩ ... ⟨ep⟩)

where each ⟨ep⟩ is an elementary pattern, as described below. A space delimits each ⟨ep⟩. In internal language it appears as

(PATTERN ⟨ep⟩* ⟨ep⟩* ... ⟨ep⟩*)

where we have used the * to indicate the transformation of each elementary pattern to its internal representation. In general * will be used to indicate the transformation to internal language as defined for all of LISP II. A pattern matches a list if each elementary pattern matches a segment of this list, and the matched segments, in order, constitute the entire list. The following are the elementary patterns:

1) Source Language: $

Internal Language: (DOL)

This will match any segment of the workspace, including a null segment, if the rest of the pattern matches the remainder of the workspace. The smallest possible segment which allows the match is found. The one exception to this rule is that $ as a last element of a pattern will match the remainder of the workspace.

2) Source Language: $ ⟨integer⟩ e.g. $1, $2, ..., $n,...

Internal Language: (DOLN ⟨integer⟩*) (DOLN 1), (DOLN 2), ...

For any integer n, $n matches n consecutive elements of the workspace.

3) Source Language: '<s-exp> e.g. 'BOY, '(A B), '4, 'THE '*AB*

Internal Language: (QUOTE <s-exp>)

A quoted s-expression matches an identical element in the workspace.

4) Source Language: <variable> e.g. A, ALPHA[4]

Internal Language: <variable>*

A variable matches an item in the workspace which is equal to the value of the variable.

5) Source Language: <pattern>

Internal Language: <pattern>*

A pattern matches an item in the workspace which is a list, and which matches the pattern in the sense defined here. If a match is found the array for this match is placed in the appropriate position of the array for the total match.

6) Source Language: $/<pred-atom>

Internal Language: (DOLP <pred-atom> *)

Matches anything that $ matches, with the additional condition that the segment found must also satisfy the predicate of one variable named by the atom <pred-atom> .

7) Source Language: $/(<pred>)

Internal Language: (DOLP <pred>*)

Exactly the same as above except that <pred> is any non-atomic predicate of one variable.

8) Source Language: $n/ <pred-atom> or $n/(<pred>)

Internal Language: (DOLNP n <pred-atom>*) or (DOLNP n <pred>*)

As above, but matches a segment of n items which satisfies the predicate of one variable named <pred-atom> or <pred> .

9) Source Language: n or nmp e.g. 4 or 1$2$4 ; in general <ar-mark> ::= <integer> | <integer>$<ar-

Internal Language: (ARMARK n*) (ARMARK n* m* p*)

An integer n refers to the contents of the nth element of the array for this match. It matches an item in the workspace identical to this element in the array. 1$2 would refer to the second item in the array stored at the first array position. This type of <ep> is called an array mark or <ar-mark>.

10) Source Language: $//(<n-pred> <p-arg>$^{n-1}$)

Internal Language: (DOLF <n-pred>* <p-arg>*$^{n-1}$)

Matches anything matched by $ with the additional condition that the segment matched satisfies the predicate of n-arguments named by <n-pred> . The first argument of <n-pred> is implicitly the list matched by the $. The other n-1 arguments are all <p-arg>'s. A <p-arg> can be an <armark> , a <variable> or a '<s-exp> . They reference previously matched items, variables and constants.

11) Source Language: $n//(<n-pred> <p-arg>$^{n-1}$)

Internal Language: (DOLNF n* <n-pred> <p-arg>*$^{n-1}$)

Same as above but matches a segment of n consecutive items in the

workspace. Similarly, we will allow:

12) Source Language: variable //(<n-pred> <p-arg>$^{n-1}$)

Internal Language: (VARF <variable>* <n-pred>* <p-arg>*$^{n-1}$)

13) Source Language: <armark>//(<n-pred> <p-arg>$^{n-1}$)

Internal Language: (ARF <armark>* <n-pred>* <p-arg>*$^{n-1}$)

14) Source Language: $$/<c-fcn>

Internal Language: (DOLL <c-fcn>*)

This is an escape mechanism to allow the user to construct his own matching function. The <c-fcn> is a function of 4 arguments, A, N, W and FN. At run time this function will be given an array A, a number N which represents the position of the <ep> , $$, in the pattern, a list W which is the workspace that the <c-fcn> should try to match, and a functional argument FN which should be used to match the rest of the workspace beyond the point matched by the <c-fcn> . If FN succeeds, and <c-fcn> matches, <c-fcn> should insert the segment it matches into the array at position N and return the array. If not <c-fcn> should return NIL.

Assignment to variables of matching segments of the workspace can be done automatically within a pattern for <ep>'s $, $1, and $$ in all their forms. This is done by preceding the <ep> by the variable, with no space between the variable and the following $. In internal language this is

represented by

(SET <variable>* <mark>*)

where <mark> is the form of $, $1, or $$ following the variable name.

IV. Formats

A format is a list of elementary formats <ef> . In source language it appears as

(<ef> <ef> .... <ef>)

where successive <ef>'s are separated by spaces. In internal language it appears as

(FORMAT <ef>* <ef>* .... <ef>*)

The list denoted by each <ef> is concatenated into the list denoted by the format. If the <ef> denotes an atom, then this atom is cons-ed into the list. The elementary formats are:

1) Source Language: '<s-exp> e.g. 'A '(A B)

Internal Language: (QUOTE <s-exp>)

The quoted <s-exp> is concatenated or cons-ed in as described above. This implies that 'A and '(A) will be treated identically.

2) Source Language: <variable>

Internal Language: <variable>*

The value of the variable is inserted in the reconstructed list.

3) Source Language: <armark> e.g. 3, 4$2

Internal Language: <armark> * e.g. (ARMARK 3) (ARMARK 4 2)

The element of the array at the position given by the <armark> is inserted into the reconstructed list. If the <armark> specifies a position in which there is an array, the elements of this array are concatenated, and this new list inserted in the reconstruction.

4) Source Language: <format>

Internal Language: <format> *

A <format> can be inserted as an <ef> within a <format> . The result of reconstructing with this inner <format> and the original array is cons-ed into the higher level list, thus allowing construction of arbitrarily complex list structures.

5) Source Language: fn(<p-arg> ... <p-arg>)

Internal Language: (FORM fn* <p-arg> * .... <p-arg> *)

This <ef> allows the insertion into the reconstruction of a list which is an arbitrary function of the matched subsegments, and other variables, and constants.

V. A Translator For Match and Construct

Suppose we are given any function of two variables f [u;v] for which we fix a value $\bar{u}$ for u. It is now effectively a function of one variable. We can define a function f* [v] which utilizes this constant in its definition. When f [$\bar{u}$;v] and f* [v] are called, the latter will run more efficiently.

Both match and construct are functions of the type discussed above. The pattern for match, and the format for construct are often known at

compile time. In LISP II we will compile calls to match and construct which have constant patterns and formats, respectively, by first translating them into LISP functions of one variable. I have written a preliminary version of this translator for match.

VI. The COMIT Statement

The COMIT statement in LISP II will provide a convenient way to call match and construct implicitly. In source language, the form of the most general COMIT statement will be:

COMIT W, U⇐<pattern> , A ⇐ <format>,....,D⇐<format> $S(\ell_1)$, $F(\ell_2)$

In internal language this would be represented as

(COMIT W, ((U <pattern> *) (A <format> *)...(D <format>*)) ℓ_1, ℓ_2)

W is a locative expression for a list which will be the workspace for the pattern match. U is an array name which will be assigned the value of the match if it succeeds. An implicit call is made to match with the pattern and W as arguments. A, B, ...D are locative expressions which are assigned values according to their respective formats if the match is successful. If the match is successful, after assignments to A, ...,D control goes to the statement labeled ℓ_1 (indicated by $S(\ell_1)$). If the match fails, a transfer is made to ℓ_2 (indicated by $F(\ell_2)$).

Many elements of this statement are optional. If "U⇐" is omitted, a local array is created but not named. If "A⇐" (the first locative expression with a format) is omitted, it is assumed to be W. No other formats need appear. $S(\ell_1)$ and/or $F(\ell_2)$ may be omitted; transfer will be made to the next statement for an omitted condition label. If either $S(\ell_1)$ or $F(\ell_2)$ are missing, they appear in internal language as NIL.