MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

ARTIFICIAL INTELLIGENCE PROJECT
MEMO 95

MEMORANDUM MAC-M-305
April, 1966

A PROGRAM FEATURE FOR CONVERT

Adolfo Guzman and Harold McIntosh

A program feature has been constructed for CONVERT, closely modeled after the similar facility found in many versions of LISP. Since it is functional or operational in nature, it has been included as a skeleton form, together with a number of related operator skeletons. This Memo describes them, and also the RUL mode, which allows the user to specify arbitrary components of a pattern as the result of a computation performed while the matching process is taking place.

Introduction

CONVERT, described in (1), is a programming language which is applicable to problems conveniently described by transformation rules. By this we mean that patterns may be prescribed, each being associated with a skeleton, so that a series of such pairs may be searched until a pattern is found which matches an expression to be transformed. The conditions for a match are governed by a code which also allows subexpressions to be identified and eventually substituted into the corresponding skeleton.

Once a match is achieved, the skeleton which specifies the expression to be formed -- using as building blocks the subexpressions identified in the matching -- may be quite arbitrary and complex since it is possible to transform partial results with the help of additional transformation rules. This has been so far accomplished with recursive skeletons.

The program feature for CONVERT, described below, allows us to specify these transformations into a rather iterative manner, with labels and transfers of control.

We finally will talk about the RUL mode, which allows a CONVERT rule set to define a pattern. This mode brings all the power of CONVERT into the matching process, and enables the user to specify arbitrary components of a pattern as the result of a computation performed while the matching process is still taking place. For example, in the pattern (XXX A B C YYY), C may be defined to match a number only if it happens

to be equal to $a^2 + b^2$ or smaller than 27 - b, where a and b are the numbers matched by the patterns A and B, respectively.

Description of skeleton forms related to the program feature.

We will assume the reader is somewhat familiar with the CONVERT language, i.e., as described in (1).

(=PROG= (XXX) S1 S2 ... Sn) The skeleton form which introduces a program. (XXX) is a list of program variables, which are atomic skeletons declared to be in the EXPR mode during the execution of the program. They may be indicated to represent fragments by enclosure in parentheses, according to the prevailing convention. Initially their values are respectively an empty list or an empty fragment, but their values may be modified by the operators =SETQ=, =MAKE=, *SETQ*, and *MAKE*.

If a program variable is already in use as a skeleton, its old value is pushed down upon entrance to the program and automatically restored upon the completion of whatever replacement is called for by the program. S1, S2, ... Sn are statements in the program which consist of skeletons which are replaced in order, starting with the first. In addition to skeletons to be replaced, program elements may also consist of heading or location markers (labels), "go to" statements, and "return" statements.

Since a (=PROG= ...) may contain any permissible skeleton, =PROG= inside =PROG=s are permissible at any depth.

(=RETN= S)
(=GOTO= S) The skeletons S1, S2 ... of a =PROG= are REPLACEd (i.e., we use the function REPLACE to compute their value) sequentially, the resulting value being ignored, except when it is of the form (=GOTO= n) or (=RETN= S), where we transfer control to n (REPLACEd), or return S (REPLACEd) as value of the =PROG=, respectively.

If no =RETN= skeleton is found and we arrive at the end of the program, the value of the (=PROG= ...) is the value of the last skeleton found.

(=RETN= (=GOTO= n)) In particular, the value of a (=PROG= ...) may be a (=GOTO=) statement, so we may return a (=GOTO=) statement. For example, consider the skeleton

```
(=PROG= (A B C) 1 (=SETQ= A 8)

                2 (=PRNT= ...)

                3 (=REPT= ...)

                4 (=PROG= (A E) A1 (X XXX (*SETQ* A (P Q R)) ...)
                                   (=RETN= (=GOTO= 2)) )

                5 ....
                  ....)
```

Here, the value of (=PROG= (A E) ...) is (=GOTO= 2), and as a result, after restoring the variables A and E to their initial state, we transfer to 2.

The same result could be achieved if instead of (=RETN= (=GOTO= 2)) we simply say (=GOTO= 2); we may think that, since the inner =PROG= has no statement #2, we empty the program when searching for it,

and as a result we abandon that =PROG=, giving as a value the last REPLACEd skeleton (that is, our (=GOTO= 2) in question). And, since we abandon the =PROG=, all the bound variables (A and E in this case) are automatically restored to their previous state.

This works at any level. In the same way, also, skeletons such as (=RETN= (=RETN= (=RETN= (XXX =SAME=)))) are valid.

Summary:

1. Automatic Back-Up. In nested =PROG='s, we may pop up as many levels as necessary, with a skeleton of the form (=RETN= (=RETN=...(=RETN= S) ...))

2. Automatic Transfer and relocation of program variables. If, inside a =PROG=, we say (=GOTO= n), the skeleton labelled n is looked for, and, if not found in the current (=PROG= ...), this is abandoned, its value becomes (=GOTO= n), and this in turn originates a search in the higher level =PROG=, etc., until statement n is found.

This is a very convenient way to transfer from inside a =PROG= to a statement outside it, going up as many =PROG='s as necessary, and at the same time correctly restoring all the variables.

```
Example.  (=PROG= (A B C) 1 (=SETQ= A =READ=)
                            (=SETQ= B 5)

                          2 (=PROG; (A D) 5  (=SETQ= A B)
                                             (=SETQ= B 8)
                                           .....
                                         6  (=GOTO= A)
                                           .....
                                         7  (=GOTO= B)
                                           .....
                                       A10  (=RETN= (=GOTO= 5))
                                         9  (=RETN= (=RETN= 14))
```

```
5 (=PRNT= B)
....
8 (A *conc* (SAME* C) C3) R)
....   )
```

When entering 2, B has the value 5 and the value of A was read from the console; the =PROG= in 2 rebinds A, and it is set (in 5, in the innermost =PROG=) to the value of B, that is, to 5. B is set to 8, also. (B in this case is a free variable, since it was not bound by the innermost =PROG=). Therefore, assuming no other =SETQ= has occurred, the statement 6 is equivalent to (=GOTO= 5), and we go there, changing the value of A to 8, (the value of B) as indicated.

If we execute statement 7 (and assuming that the value of B continues to be 8) we transfer to the statement #8; in doing this we abandon the innermost =PROG=, and therefore A and D are rebound to their initial values, namely the S-expression read from the console, and D (since D is "undefined" in the outermost =PROG=, stands for itself).

The statement #A10 allows us to go to the #5 (in the outer =PROG=), and then we print B, that is, 8. (B, as a free variable, has been changed in the inner =PROG=, and this change still holds).

The statement #9 "unwinds" both =PROG='s, and then the value of the outer =PROG= is 14.

GENERALIZED =GOTO= and =RETN= STATEMENTS.

In addition to (=GOTO= n) and (=RETN= S), we have the long or generalized form

(=RETN= S S1 S2 ... Sm)

(=GOTO= n S1 S2 ... Sm)

A replacement is made from left to right, and after to REPLACE the last, we return S or transfer to n, respectively. For example,

(=RETN= (=SETQ= B 5) (=SETQ= A (C R E)) (=SETQ= B 8))

we assign to B the value 5, to A the value (C R E), to B the value 8. And we return 5 (not 8) as value. The value of A is still (C R E); the value of B is still 8, but, unless they are free program variables, this binding is lost when the =RETN= exits the =PROG=.

Note: Since (=RETN= ...) and (=GOTO= ...) only make sense in the "top level", we do not have the versions (*RETN* ...) and (*GOTO* ...).

PROGRAM VARIABLES.

The former examples showed informally how program variables are used. They work very much in the same way as LISP program variables; (=PROG= (A B C ...) .....) initializes the variables A, B, C, ... to (), and saves their old values;

(=SETQ= A S1) makes a REPLACEment on the skeleton S1, and its value is assigned to A; A is considered =QUOT=ed (it is not REPLACEd). The value of (=SETQ= ...) is the value of its second argument REPLACEd.

A =PROG= is exited (abandoned) when:

1. A (=RETN= S) statement is encountered-- in this case the value of the (=PROG= ...) is S REPLACEd.

2. A (=GOTO= n) statement is encountered and n is not in the range of the (=PROG= ..). In this case, the value of such (=PROG=..) is (=GOTO= n), which, if there is an outer =PROG=, originates in turn a transfer to n (or else the abandoning of that =PROG=, etc.).

3. The end of the program is encountered without having executed a (=RETN= ..) or a (=GOTO= ..) transferring outside. In this case the value of the (=PROG= ..) is the last skeleton REPLACEd (or (), if none).

In all cases, abandoning the =PROG= causes all its variables to be restored to the value they had when the (=PROG= ..) was entered, so that the (=PROG= ..) is invisible (with respect to its program variables) to the rest of the program.

=SETQ= may also modify free variables (that is, bounded outside the =PROG=), in which case the alteration subsists.

-- A word of caution. Since the value of (=SETQ= A S1) is S1 REPLACEd, a skeleton such as (=SETQ= B (=GOTO= 7)) not only gives to B the value (=GOTO= 7), but in addition transfers to 7. If this is not what we want, a simple way to avoid it is to write

((=SETQ= B (=GOTO= 7)))

since in this way the value of the skeleton is ((=GOTO= 7)) and therefore no transfer is made at this point.

(=MAKE= V S1) is like =SETQ=, but we REPLACE V first. If the value of V is A, then

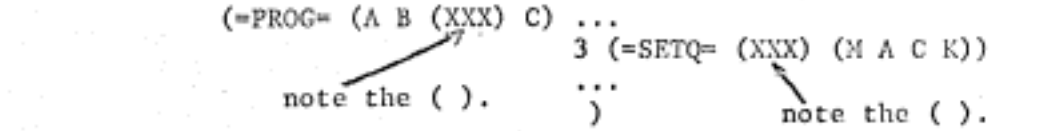
(=MAKE= V S1) is equivalent to (=SETQ= A S1)

and

(=SETQ= B C) is equivalent to (=MAKE= (=QUOT= B) C)

FRAGMENT PROGRAM VARIABLES

Since in CONVERT, fragments are a valid data type, we have also fragment program variables, which we denote by declaring them in parentheses:



Here, XXX is one of those. In this example, the value of XXX is M A C K and, for example, (=SETQ= (XXX) (1 XXX 2 XXX 3 XXX R)) gives to (XXX) the value (1 M A C K 2 M A C K 3 M A C K R), that is, gives to XXX the value 1 M A C K 2 M A C K 3 M A C K R.

As always, fragments do not have independent existences, but they only can occur inside a bigger skeleton [inside a list].

RELATION AMONG =PROG= VARIABLES AND DICTIONARY VARIABLES

Both have the same hierarchy, and in case of conflict, the latest binding takes precedence.

For example, if we delcare in our dictionary (... X EXPR (M I T) ...) then outside of (=PROG= (X Y) .....), X has the value (M I T), and inside it, X has whatever value it has acquired by use of =SETQ='s, or (), if none. Abandoning the =PROG= causes, of course, the restoration of X to its former value (M I T).

Although not necessary, we recommend:

1. To use numbers as labels.
2. Not to use the same variable as a program variable and as a dictionary variable.

Warning: The program variables only have existences in the right hand side of a rule, that is, as (or inside of) a skeleton. Therefore, they are not recognized in the pattern side, their value (in a pattern) being either themselves or its associated expression in the dictionary. (if necessary, this rule may be modified according to the experience gained by the users.)

HEADERS (labels): They are optional, and they may (must) be an atom, except perhaps NIL. A single statement (skeleton) may have more than one label, or none at all.

CONDITIONAL STATEMENTS.

Since a program (in the =PROG= sense) is a sequence of any skeletons, any one of these may be of the form

(=WHEN= S P S1 S2)

(=COND= S P S1 S2)

(=REPT= S Ck ...)

(=CONT= S Ck ...)

(=BEGN= S Ck ...)

The first two represent short conditionals, much in the same way as (IF P Q R) is used in some LISP systems, such as MBLISP or Hawkinson-Yates LISP; the reader will recognize the last three: they are the standard recursive skeletons in CONVERT, and therefore their use here does not represent a special case, but the confirmation that any valid skeleton is admissible inside a =PROG=. They play in CONVERT the same role that COND plays in LISP. Of course, in general, the collection of rules whose name is Ck may be recursive and use this =PROG= again. Also, it is possible that, as a result of the application of the set of rules Ck to S, the value will be of the form (=GOTO= ..) or (=RETN= ..), in which case the proper action is taken.

(=WHEN= S P S_1 S_2). When skeleton S (replaced) matches pattern P, its value is S_1; otherwise is S_2, or S if S_2 is missing. It is equivalent to

```
(=CONT= S * (
(P S1)
(== S2) ))
```

(=COND= ....) differs from (=WHEN= ....) in this way: when comparing S against P, =WHEN= uses the most recent dictionary, as =CONT= does, and =COND= uses the original dictionary, as =REPT= does.

HIDDEN CONTROL STATEMENTS

If we declare in the dictionary (... G SKEL (=GOTO= 5) ..)

then in a program like (=PROG= (A B) .....
.....
G
.....)

G (since it is a skeleton) stands for a transfer to 5.

This gives to us two advantages:

1. A shorter notation.

2. Since we may change the value of G (after all, G is just another skeleton) at running time, the program may modify itself during execution.

For example, a dictionary containing F SKEL =READ= gives to a program like (=PROG= (B) .....
5 .....
F
8 .....)
the ability to request data from the teletype and execute it; the users may type (=GOTO= 5) or (=SETQ= B 0.0825), or (A B). In this last case, the skeleton (A B) is REPLACEd, and then, in general* control is given to the next statement in sequence, 8 in our example, the value of (A B) being lost.

As a last example in this section, a skeleton of the form

```
(=PROG= (A B) 1 .....
              2 .....
              (=PROG= (C D) 10 .....
                            20 .....
                               ..... )
              3 .....
              4 .....
                ..... )
```

may be written as

```
(=PROG= (A B) 1 .....
              2 .....
               P
              3 .....
              4 .....
                ..... )
```

*If A happens to be declared as (A SKEL =RETN=), then a skeleton like (A B) has a value (=RETN= B) and, if found in the convenient level, causes the =PROG= to return B REPLACEd as value.

if we define P in the dictionary as
P SKEL (=PROG= (C D) 10 ..... 20 ....)
As a particular case, P may use itself

```
P SKEL (=PROG= (C D) 10 .....
                    20 .....
                      P
                    30 .....
```

Note that a definition like this last one gives to the user the possibility to give name to a program and to call it by name.

FRAGMENT PROGRAMS

In the same way as P could stand for a single skeleton i.e., a (=GOTO= ..), (=SETQ= ..), (=PROG= ..), etc., a fragment may represent any part of a program:

```
(=PROG=(M) 1 .....
           2 (=PRNT= 2)
             (=GOTO= 1)
             (=SETQ= M R)
             G
           6 (=WHEN= M 1000 (=BEGN= R) (=RETN= (X XXX YYY)))    )
```

may be written as

```
(=PROG= (M) 1 .....
            2 (=PRNT= 2) ...
              WWW
            6 (=WHEN= M ...)    )
```

if we declare WWW as (WWW) SKEL ((=GOTO= 1) (=SETQ= M R) G).

WWW may, naturally, use itself, or, for example, WWW may use G and G may use WWW.

The rule for fragments is: fragments encountered in the program are REPLACEd and its value interpreted as a part of the program. (We may think of appending the value to the rest of the program). Fragment programs cannot contain labels.

An interesting case arises when, depending upon the nature of certain data, we have to choose among several subprograms to handle it, and then return to a common merging point. For example,

```
(=PROG= (M N) 1 .....
              2 .....
              (*WHEN* DATA PROPERTY1 (WWW) (XXX))
              4 .....
              5 .....  )
```

Here, depending upon DATA having PROPERTY1 or not, we choose among subprogram WWW or subprogram XXX. The utility of this characteristic arises when WWW, XXX, etc., are used in several places in the same program, so the return address is not the same in every case. Of course, another way to make the return is to do

(=SETQ= SWITCH n) in the calling sequence, and

(=GOTO= SWITCH) as return transfer in the subprogram.

An alternative (and perhaps more elegant way) is to write

```
(=PROG= (M N) 1 .....
              2 .....
              (*WHEN*  DATA PROP1 (WWW) (XXX))
              3 .....
              4 .....
              (*WHEN*  DATA PROP1 (WWW) (XXX))
              5 .....  )
```

or, even better,

```
(=PROG= (M N) 1 .....
              2 .....
               BBB
              3 .....
              4 .....
               BBB
              5 .....  )
```

where BBB is declared in M as (BBB) SKEL ((*WHEN* DATA PROP1 (WWW) (XXX))).

Used in this fashion, fragment programs are closed, callable subroutines, with which the programmer does not have to worry about return addresses.

GENERATION OF SUBPROGRAMS. *PROG*.

As usual, there exists the * version of =PROG=, namely *PROG*. The value of (*PROG* (M N) ...) is the content of the value that (=PROG= (M N) ...) would produce.

The last sentence only makes sense if the resulting value of =PROG= is a list. Among other things, (*PROG* ...) allows us to produce (to generate) a CONVERT program, and then to run it.

The RUL mode.

P RUL S

This entry in the dictionary defines the atom P as being a pattern whose match is carried out in the following way:

When compared against an expression E, P applies to that expression the set of rules S, and watches the result of that transformation. If it is =FAIL=, P fails to match E. If it is the atom =TRUE=, a success is reported: P did match E; whatever variables were bound in this match are kept (that is, we keep the variables bound by the transformation S when it was being applied to C) and they enrich the current dictionary.

Suppose we are comparing a pattern (P_0 P_1 ... P_j P P_k ... P_n) against a certain expression, and P is defined in the dictionary to

be in the RUL mode:

P RUL S.

When we arrive at P, just before it starts its comparison, patterns P_0 through P_j have already been matched (successfully), tentatively at least, and they have identified several subexpressions; that is, certain variables have been bound: some UAR's and PAV's have changed to VAR, etc., all this information is, of course, in the current dictionary.

Now we want to compare P with its corresponding expression (call it E'). We take the current dictionary and with its help we apply the transformation S to E'; that is, variables which were bound in P_0 ... P_j retain their value inside S and are therefore available.

If the transformation is successful (its value is =TRUE=), a new dictionary has been produced; we will use it when we compare P_k through P_n.

SUMMARY

P RUL S appears in the dictionary. S is a set of rules which is applied (under the current dictionary) to E; the skeletons of S may have only two values, =TRUE= or =FAIL=.

The rules of S specify the conditions under which P matches or fails. Since the actual dictionary is available, these conditions are (can be) functions of previous matchings.

If some rule of S is successful, the variables bound by its pattern-half are kept for future comparisons.

```
Example 1.  CONVERT(    (X PAV =ATO=
                         R RUL (  (=ATO= =FAIL=)
                                  ( (=== x ---) =TRUE=)
                                  ( == =FAIL=)
                               )
                        )
                        ()
                        ((A) B (C))
                        (C1 (  (R (X X X)) ))          )
```

We compare R to ((A) B (C)); if ((A) B (C)) were an atom, would be rejected, says (=ATO= =FAIL=). Now we see if it is a list containing an atom B. Therefore, the answer is (B B B).

The same program applied to A answers A.

Applied to (() () () ()) answers (() () () ())

Applied to (() 1 2 3 6) answers (1 1 1)

Example 2. Let M --the first argument of (CONVERT M I E R)-- be

```
(A  PAV  =NUM=
 B  PAV  =NUM=

 C  RUL  (    ( =NUM= (=PROG= ()
                              (=SETQ= D (=TIMS= 4 A =SAME=))
                              (=WHEN=  (=SAME= (=PLUS= A B)) (Y Y)
                                       (=RETN= =TRUE=)
                                       (=CONT= (A B =SAME=) C8) )))

                 ( == =FAIL=)   ))
```

Let I be (Y) --the dictionary of initially undefined variables--

Let the pattern being compared in this moment be (=== A === B === C ===)

(this pattern is found in some rule of R, fourth argument of CONVERT).

C will match a number equal to A + B, or else C8 is called to transform (A B =SAME=); if that transformation gives (=RETN= =TRUE=) as value, C will match that number.

C also refuses to match anything but numbers.

In the mean-time, 4AC has been stored in D.

OBSERVATIONS:

1. It makes no sense to have a fragment version of the RUL mode, or to have *TRUE*, for instance.

2. As it is implemented, now everything which is not =TRUE= is supposed to be =FAIL=. Therefore, only the rules of success deserve to be mentioned.

AVAILABILITY

The =PROG=ram feature already forms part of the CONVERT processor.

Link CONVRT SAVED T316 4170

It runs smoothly. Nevertheless... for bugs: phone X 5866, A. Guzmán.

References: (1) Guzmán, A. and McIntosh, H. CONVERT, presented at the SIC-SAM Symposium (ACM), March 29-31, 1966, Washington, D.C. To be published C. ACM. Aug 1966.

Guzmán, A. CONVERT. Tesis Profesional. Instituto Politécnico Nacional (México). 1965. (Spanish).