N E W S L E T T E R # 8

January 1983

Aloha from Hawaii! The Soft Warehouse Newsletter provides you with information on new Soft Warehouse products, and software extensions or corrections to existing products. In addition, the newsletter is a medium for the exchange of ideas and application programs within the growing community of **muMATH** and **muLISP** users.

If you would like to subscribe, or extend your subscription to the Newsletter for three issues beyond the expiration number on your mailing label, please send $6 ($10 for orders from outside the U.S. or Canada) by check, VISA, or Master Card to The Soft Warehouse, P.O. Box 11174, Honolulu, Hawaii, 96828, U.S.A. A complete set of back issues is available on request for $15.

Announcing the AAAI-83 Conference

The third national conference of the American Association for Artificial Intelligence will be held from August 22 to 26, 1983 in Washington, D.C. Authors of substantial, original research in AI can submit papers to AAAI, 445 Burgess Drive, Menlo Park, CA, 94025 or the Arpanet address AAAI-OFFICE@SUMEX-AIM.

Recent muMATH Reviews

The October 1982 edition of Creative Computing contains a positive review of the CP/M-80 version muMATH/muSIMP-80 by **Dr. David D. Shochat** of Santa Monica College, California. It revises an earlier review of muMATH-79 that appeared in the ACM SIGPC (Special Interest Group, Personal Computers) Notes.

Bruce Powel Douglass is also a muMATH enthusiast. He has a short review of the full feature TRS-80 version of muMATH/muSIMP in the November 1982 issue of 80 Micro. Currently this full feature version for the TRS-80 Models 1 and 3 can be obtained only from The Soft Warehouse. Owners of the limited version TRS-80 muMATH may qualify for a discount when they purchase the full version.

A "Down Under" muMATH User Group

Australian muMATHematicians interested in forming a users group should write Grahame Wilson at P.O. Box 145, Glebe, N.S.W. 2037; or leave a message on The Australian Beginning net for user WILSPOBOGLEB; or leave a message on the MI Computer Club net for user #1604. Mr. Wilson will also soon have a Source user address.

The SUPER-CALCULATOR: A Scientific Calculator using muMATH

Stuart Edwards -- Honolulu, Hawaii

muMATH users normally use the system in the "calculator mode" of operation. In this mode, the user enters an expression; the system evaluates the expression and displays the evaluated result. Building on the algebraic capabilites of muMATH, I have extended the capabilities of the system to make it function as a sophisticated scientific calculator. Specifically, the package I have developed:

* has an extensive set of predefined units and physical constants;

* allows "implicit multiplication" whereby the "*" operator is assumed if no other operator is present;

* displays the numerical part of any answer separately from the non-numerical part and automatically converts it to scientific notation if it is very large or very small;

* carries units along algebraically in the calculation, converting them as necessary to standard units;

* provides an infix operator "in" which allows the user to specify in what units the answer is to be displayed.

For instance, the system calculates expressions such as

```
? 53 kilo meter / (5 liter) in mile / gallon;
@: 24.9321412 mile / gallon
```

This expression calculates mileage in mpg for someone who has driven 53 kilometers on 5 liters of gas. As another example, assuming the Earth is a perfect sphere of radius 3959 miles, then its volume in teaspoons would be

```
? 4 pi / 3 (3959 mile)^3 in teaspoon;
@: 2.1981149 10^26 teaspoon
```

The choice of teaspoons as a unit of volume here is of course arbitrary; the result may just as easily be displayed in gallons, cubic miles, bushels, etc.

The "in" operator is smart enough to know if the requested units make sense for the calculation being performed. Thus in the example, since 4, pi, 3 and 3959 are all dimensionless and "mile" is being raised to the third power, "in" can tell that the result is a volume and should be expressed in units of volume. Since "teaspoon" is a unit of volume, the dimensions are consistent.

All this checking is done by defining the units in terms of standard units. For example, the standard unit for length is the

meter, and each unit of length is defined as some number of meters. Areas are in square meters ("meter^2"), volumes are in cubic meters ("meter^3"), etc. When a calculation is performed, everything is first converted to standard units and the resulting expression is simplified algebraically. Next, the "in" operator looks at the expression on the right - the desired units - and converts it into standard units too. Then since the required result and the units in which it is to be expressed are both in standard units, it should be possible to divide them algebraically and have all units cancel out, leaving just the numerical portion. The final step is to take this numerical portion and append the required units in their original form for display.

If the units are not appropriate for the expression being evaluated, then not all the units will cancel when "in" does its algebraic division. This is readily apparent in the final result, since extra standard units will be introduced to force an algebraically correct answer. For instance, suppose we want to compute the average speed of an airplane that travels 2250 miles in 5 hours, using the expression

```
? 5 hour / (2250 mile) in meter / second;
@: (0.0049709 second^2 / meter^2) meter / second
```

Here "meter / second" is an appropriate unit of velocity, but "5 hour / (2250 mile)" is the wrong way to perform the calculation. It should be "2250 mile / (5 hour)". Since we entered the expression incorrectly, the units do not all cancel out and the extra "second^2" and "meter^2" are introduced to show us our error and to compensate algebraically for it.

This is an important feature of the calculating system because it provides a check that the formula you are using is appropriate for the units you think the answer should be in. For example, if you think you're computing a velocity and your expression evaluates to a frequency, the error is immediately apparent.

Over a hundred pre-defined units and physical constants are included with the system. And of course, you are free to add more units and physical constants to the built-in ones.

There are three pages of documentation including examples and six pages of commented source listing. To run the package, it is recommended that your system have at least 48K of main memory.

To obtain a printed listing of the source and documentation for the SUPER-CALCULATOR package send $10 (U.S.) by check or money order to The Soft Warehouse, PO Box 11174, Honolulu, Hawaii, 96828-0174, USA. Note that the SUPER-CALCULATOR package is copyrighted by Mr. Edwards and is not A Soft Warehouse product. Questions concerning the package should be directed, in writing, to Mr. Edwards, c/o The Soft Warehouse.

A muSIMP Breakpoint Debugging Facility

Steve Messick -- Fairbanks, Alaska

The file BREAK.MUS contains the code for a simple muSIMP break package. Breaking into a function while it is executing is often useful when debugging. Temporarily stopping the program allows one to examine the values of local as well as global variables. These values can even be changed before allowing the program to continue.

The package is fairly simple to use. Edit the desired functions (e.g. using an external editor or PDS) to insert a call to the function BREAK in any suspect code. When BREAK is called, it prints its arguments and calls a modified DRIVER function. If it is non-FALSE, the first argument is printed. The second is evaluated before printing. If the third argument is non-FALSE, the second is printed in list notation, otherwise the second is printed in mathematical notation. Type "RETURN" and a semicolon to exit the break and allow the program to continue.

As an example of the use of the break package, imagine you are working on a function named SCALEFACTORS. Then if

```
BREAK ("In SCALEFACTORS, VAR= ", VAR),
```

is inserted within the function, each time SCALEFACTORS is called the quoted string is printed followed by the value of the variable VAR. Typing "RETURN" after setting VAR to a different value will cause SCALEFACTORS to resume execution using the new value of VAR.

When the modified DRIVER function is called by BREAK, it exits if you type "RETURN" following the question mark prompt. If you have the PDS muSIMP editor, the easiest way to get a copy of DRIVER is to use SAVEPROG to write DRIVER to a file. Then a text editor can be used to make the necessary modifications to create the file listed below. Note that unusual variable names have been used to avoid conflict with the names of local variables.

```
% File:  BREAK.MUS            5 January 83           Steve Messick %

% BREAK prints its arguments, then calls the new DRIVER function %

SUBROUTINE BREAK ("!BREAK1", "!BREAK2", "!BREAK3"),
  NEWLINE (),
  BLOCK
    WHEN "!BREAK1",
      PRINT ("!BREAK1"),
      WHEN "!BREAK2",
        BLOCK
          WHEN "!BREAK3",  PRINT (EVAL("!BREAK2")) EXIT,
          PRTMATH (EVAL("!BREAK2")),
        ENDBLOCK EXIT EXIT ENDBLOCK,
  DRIVER (FALSE, '*),
ENDSUB$
```

```
% This version of DRIVER exits after "RETURN" is typed. %

FUNCTION DRIVER ("!BREAKX1", "!BREAKX2", "!BREAKX3", "!BREAKX4",
    "!BREAKX5"),
  "!BREAKX4": '?,
  BLOCK
    WHEN "!BREAKX2",
      "!BREAKX4": "!BREAKX2",
      "!BREAKX3": TRUE,
      "!BREAKX2": FALSE EXIT,
  ENDBLOCK,
  BLOCK
    WHEN NOT "!BREAKX1",  RDS () EXIT,
  ENDBLOCK,
  WRS (),
  READCHAR: TRUE,
  NEWLINE (2),
  LOOP
    ERR: FALSE,
    BLOCK
      WHEN ECHO (),
        PRINT ("!BREAKX4"),  SPACES (1),
        WHEN NOT RDS AND BELL,  PRINT (ASCII(7)) EXIT EXIT,
    ENDBLOCK,
    "!BREAKX1": FALSE,
    "!BREAKX1": PARSE (SCAN (), 0),
    "!BREAKX2": SCAN,
    BLOCK
      WHEN ECHO (),  NEWLINE (NEWLINE) EXIT,
    ENDBLOCK,
    BLOCK
      WHEN ERR OR NOT TERMINATOR(),
        SYNTAX (OPERATOR, "NOT FOUND"), NEWLINE () EXIT,
      WHEN "!BREAKX1" EQ 'RETURN,
        "!BREAKX5": TRUE EXIT,
      WHEN "!BREAKX2" EQ '$,
        @: EVAL ("!BREAKX1"),
        WHEN ECHO (),
          NEWLINE () EXIT EXIT,
      PRINT ('@),  PRINT (':),
      @: EVAL ("!BREAKX1"),
      SPACES (1),
      BLOCK
        WHEN "!BREAKX2" EQ ';,
          PRTMATH (@, 0, 0, TRUE) EXIT,
        PRINT (@),
      ENDBLOCK,
      NEWLINE (2),
      NEWLINE (NEWLINE),
    ENDBLOCK,
    WHEN "!BREAKX3" AND "!BREAKX5",  FALSE EXIT,
  ENDLOOP,
ENDFUN$

RDS ()$
```

A Mini-muLISP Editor

Peter D. Grogono -- Montreal, Quebec

I have written a "quick and dirty" editor for developing muLISP programs that are too large to run along with the muSTAR editor. My editor is not as easy to use as muSTAR and is somewhat limited in its capabilities. For example, you can neither delete nor insert an expression before the first element of a list. Nevertheless, I find it a useful tool that makes it possible to interactively polish up large muLISP programs. An experienced muLISP hacker could probably improve it without a major increase in size.

To use the editor, startup muLISP without muSTAR and read in the file EDIT.LIB using an RDS command. Variables, functions, and property values can be edited by entering one of the following expressions:

```
(EDIT <variable>)                  a variable's value,
(EDFUN <function>)                     a function's definition,
(EDPROP <variable> <property>) a variable's property value.
```

An edit command is invoked by entering a single letter. The command is applied to the object currently being displayed. The following is a list of the currently implemented commands.

A Get CAR of object.
D Get CDR of object.
N Get next component of a list. The previous command must have been "A" or "N".
U Undo the last A or D command (i.e. go back up the structure).
C Change the object. You enter the new value of the object as a LISP S-expression terminated by <RETURN>. Previous command must have been "A" or "N".
I Insert a component into a list. If the previous command was "A" or "N", insert the new object after the displayed object. If the previous command was "D" or "X" insert the new object at the front of the displayed list.
X Delete the first component of the displayed list. Previous command must have been "D" or "X".
R Reset: display entire modified structure.
E Exit editor.

I have also written three other muLISP programs that may be of interest to readers of the Newsletter: an interpreter for a subset of D. A. Turner's programming language SASL, implemented by combinators; an interpreter for John Backus's FP notation; and a verifier for programs written in Dijkstra's guarded command notation. The interpreters are "toy" programs, but they suffice to illustrate some of the possibilities (and difficulties) of purely applicative programming.

The verifier consists of a parser, a verification condition

generator, a simplifier, and a theorem prover. It accepts small programs adorned with pre- and post-conditions, generates theorems whose truth guarantees partial correctness of the program, and attempts to prove the theorem by natural deduction. The programs may contain assignments, if and do statements, and arrays; you have to provide loop invariants. I suspect that this program could be considerably enhanced without exceeding the limitations of a micro-processor.

If you would like to obtain these other programs in printed or machine readable form, please write to me at 4125 Beaconsfield Ave., Montreal, Quebec, H4A 2H4, CANADA or phone (514) 483-1422.

```
% File:  EDIT.LIB           November 83           Peter D. Grogono %


(PUTD DEFUN (QUOTE (NLAMBDA (NAM BDY)
  (PUTD NAM BDY) NAM)))

(DEFUN PUTQQ (NLAMBDA (NAM IND BDY)
  (PUT NAM IND BDY) NAM)))

(DEFUN EDFUN (LAMBDA (FUN)                % Edit a function %
  (PUTD FUN (EDIT (GETD FUN))) ))

(DEFUN EDPROP (LAMBDA (ATM PROP)          % Edit a property value %
  (PUT ATM PROP (EDIT (GET ATM PROP))) ))

(DEFUN EDIT (LAMBDA (OBJ                  % Edit a variable %
    % Local variables: %  CMD STACK)
  ($RESTORE)
  (PRINT OBJ)
  (SETQ READCH NIL)
  (READCH)
  (LOOP
    (SETQ CMD (READCH))       % Insert (PRIN1 CMD) to echo command %
    ((EQ CMD (QUOTE E))
      (SETQ READCH T)
      ($RESTORE) )
    ( ((MEMBER CMD (QUOTE ( A D N U C I R X )))
        (APPLY (GET CMD $EDFUNC) NIL)
        (PRINT OBJ) )     % Use a pretty-printer if you've got one %
      ($ERR) ) ) ))

% Report an error.  Could be replaced by CTRL-G to ring the bell. %

(DEFUN $ERR (LAMBDA ()
   (PRINT "Error!") ))

% Get the modified object from the bottom of the stack. %

(DEFUN $RESTORE (LAMBDA ()
  (LOOP
```

```
    ((NULL STACK))
    (SETQ OBJ (CDR (POP STACK))) )
  (PUSH (CONS (QUOTE A) OBJ) STACK)
  OBJ))

% Each action is defined as a property of the corresponding command %

(PUTQQ A $EDFUNC (LAMBDA ()
  ((ATOM OBJ) ($ERR))
  (PUSH (CONS (QUOTE A) OBJ) STACK)
  (SETQ OBJ (CAR OBJ)) ))

(PUTQQ D $EDFUNC (LAMBDA ()
  ((ATOM OBJ) ($ERR))
  (PUSH (CONS (QUOTE D) OBJ) STACK)
  (SETQ OBJ (CDR OBJ)) ))

(PUTQQ N $EDFUNC (LAMBDA ()
  ((EQ (CAAR STACK) (QUOTE A))
    ((NULL (CDDAR STACK))
      ($ERR) )
    (PUSH (CONS (QUOTE A) (CDDAR STACK)) STACK)
    (SETQ OBJ (CADAR STACK)) )
  ($ERR) ))

(PUTQQ U $EDFUNC (LAMBDA ()
  ((NULL (CDR STACK)) ($ERR))
  (SETQ OBJ (CDR (POP STACK))) ))

(PUTQQ C $EDFUNC (LAMBDA ()
  ((EQ (CAAR STACK) (QUOTE A))
    (PRIN1 "? " )
    (SETQ READCH T)
    (SETQ OBJ (CAR  (RPLACA (CDAR STACK) (READ))))
    (SETQ READCH NIL)
    (READCH) )
  ($ERR) ))

(PUTQQ I $EDFUNC (LAMBDA ()
  (PRIN1 "? " )
  (SETQ READCH T)
  (SETQ OBJ (CDR (RPLACD (CDAR STACK) (CONS (READ) (CDDAR STACK)))))
  (SETQ READCH NIL)
  (READCH) ))

(PUTQQ R $EDFUNC (LAMBDA ()
  ($RESTORE) ))

(PUTQQ X $EDFUNC (LAMBDA ()
  ((EQ (CAAR STACK) (QUOTE D))
    (SETQ OBJ (CDR (RPLACD (CDAR STACK) (CDR (CDDAR STACK))))) )
  ($ERR) ))

(RDS)
```

MACROS: Expanding Your Programming Horizons

We have synthesized work done by **Tom Yonkman** of San Diego, California and **Steve Messick** of Anchorage, Alaska to produce a macro expansion package for muLISP. The source file, MACROS.LIB, consists of muLISP functions that enable you to write macro definitions in the Maclisp style.

A macro is similar to a no-spread, call-by-name function in that it takes any number of arguments and does not evaluate them. Unlike no-spread functions the entire macro call, including the macro name, is bound to the macro's lone parameter. Also unlike other Lisp function types, the evaluated macro is not returned directly but instead is evaluated a second time. The function **DM** is used to define new macros using the form:

(DM <identifier> (<var>) <body>)

As a trivial example, the following defines the macro FIRST to be a pseudonym for CAR:

```
(DM FIRST (BODY) (CONS (QUOTE CAR) (CDR BODY)))
```

When muLISP evaluates the macro call (FIRST X), (FIRST X) is bound to BODY. The first evaluation returns the list (CAR X), the second the value of (CAR X). Macros first translate, then execute. Hence the need of double evaluation. The translation phase is commonly referred to as macro expansion.

The much more useful LET macro, as defined in MACROS.LIB, provides a convenient means of creating and initializing temporary variables within a function definition. The syntax for a LET macro call is

(LET ((<var_1> <val_1>)
(<var_2> <val_2>)
...
(<var_n> <val_n>)) <body>)

Variables are assigned their respective values and the body is evaluated within that new environment. When the body is exited the original environment is restored. The LET macro expands into the form

((LAMBDA (<var_1> <var_2> ... <var_n>) <body>)
<val_1> <val_2> ... <val_n>)

With macros you can abbreviate frequently used code, customize muLISP or create an entirely new language designed to solve a specific problem. The authors of **Artificial Intelligence Programming** (Charniak et al., Lawerence Erlbaum Associates, 1980) are strong advocates of Lisp macros. It is an excellent source book for Lisp macros and AI programming techniques in general.

```
% File:  MACRO.LIB                                        October 83 %

               %  A muLISP Macro Expansion Package  %


% DM is the macro defining function %

(DEFUN DM (NLAMBDA L
  (PUTD (CAR L) (LIST NLAMBDA *MACRO-BODY* (LIST (QUOTE MACHAC)
     (LIST (QUOTE QUOTE) (CAR L)) *MACRO-BODY*)))
  (PUT (CAR L) MACRO (CONS NLAMBDA (CONS (CAADR L) (CDDR L))))
  (SAVELIST (QUOTE FUNCTION) (CAR L))
  (CAR L) ))

%  DEX  is identical to DEFUN except that it expands macros  when  a
function is defined. %

(DM DEX (L)
   (CONS (QUOTE DEFUN) (EXPANDMACROS (CDR L))) )

(DEFUN EXPANDMACROS (LAMBDA (*L*)
  ((OR (ATOM *L*) (EQ (CAR *L*) (QUOTE QUOTE)))
    *L* )
  ((ISMACRO (CAR *L*))
    (LIST (QUOTE EVAL)
        (EXPANDMACROS (APPLY (ISMACRO (CAR *L*)) *L*)) ) )
  (EXPANDREST *L*) ))

(DEFUN EXPANDREST (LAMBDA (*N*)
  ((ATOM *N*) *N*)
  (CONS (EXPANDMACROS (CAR *N*)) (EXPANDREST (CDR *N*))) ))

(DEFUN ISMACRO (LAMBDA (X)
  ((NAME X)
    (GET X MACRO) ) ))

% MACHAC expands the macro when called. %

(DEFUN MACHAC (LAMBDA (*MACRO-FUNC* *MACRO-BODY*)
  (EVAL (APPLY (ISMACRO *MACRO-FUNC*)
        (CONS *MACRO-FUNC* *MACRO-BODY*))) ))

(DM LET (*LET*)
  (LET-AUX (CADR *LET*) (CDDR *LET*)) )

(DEFUN LET-AUX (LAMBDA (VARS-N-VALS BODY
    % Local: %  VARS VALS )
  (LOOP
    ((NULL VARS-N-VALS))
    (PUSH (CAR  (CAR VARS-N-VALS)) VARS)
    (PUSH (CADR (POP VARS-N-VALS)) VALS) )
  (CONS (CONS (QUOTE LAMBDA) (CONS (REVERSE VARS) BODY))
        (REVERSE VALS) ) ))

(RDS)
```