EXPERIENCE WITH THE

muSIMP/muMATH-80

SYMBOLIC MATHEMATICS SYSTEM

David D. Shochat
Santa Monica College

The following is a revision of a review of muSIMP/muMATH-79 which originally appeared in SIGPC NOTES V. 4 #1/2.

About 20 years ago, I had the opportunity to see a real computer, an IBM 709. When I asked whether it could do Calculus, I was told that a digital computer could do numerical differentiation and integration, but that symbolic manipulation of the function definition was inherently outside the domain of the computer.

The muMATH software package, developed by Albert D. Rich and David Stoutemyer, of the Soft Warehouse and available through Microsoft for 8080/Z-80 disk systems (and now APPLE][as well), can do a surprising amount of Calculus -- symbolically, the way you learn to do it in a calculus class. For example, muMATH can integrate

```
(1 + X^2)^(1/2)
```

in about 7 seconds (on my 2 MHz Z-80 CP/M system). muMATH can also do exact rational arithmetic, with (up to) 254 byte integers in any number base to 36, a great deal of algebra, some symbolic trig and even matrix algebra: muMATH will find the inverse of

```
{[1, 2],      or of      {[a, b],
 [3, 4]}                  [c, d]}.
```

Getting all of this to work on a microcomputer with only 64K (or less) is impressive, to say the least. The secret is modular construction. You only load part of muMATH at a time. It is unlikely, for example, that you will need to do matrix inversion and integration at the same time. Furthermore, the precious memory resources must be utilized as efficiently as possible, and this is reflected in the ritual of building and saving the system.

To get started, execute MUSIMP.COM. The muSIMP interpreter is loaded in, a signon message is printed, and then the muSIMP "?" prompt appears.

muSIMP is the LISP "surface language" in which muMATH is written. You may respond to the prompt in a number of ways:

(1): You may make an assignment:

```
S: 13;          (cr)
```

muSIMP's response here would be:

```
@: 13

?
```

(2): You may use an existing function:

```
D: S * 4 ;      (cr)
```

```
@: 52

?
```

(3): You may define a new function:
[every line of user input terminated with cr]

```
FUNCTION COMBS (N, K),
  WHEN K = 0, 1 EXIT,
  N * COMBS (N - 1, K - 1) / K
ENDFUN $

? COMBS (D, 5);

@: 2598960

?
```

The definition of COMBS, which computes the number of combinations of N things K at a time, shows that recursion is allowed.

```
LISTEXAMPLE: '((1 ONE) (2 TWO)
  (3 THREE)) &

@: ((1 ONE) (2 TWO) (3 THREE))

? FIRST (LISTEXAMPLE) &

@: (1 ONE)

? REST (LISTEXAMPLE) &

@: ((2 TWO) (3 THREE))

? REST (FIRST (LISTEXAMPLE)) &

@: (ONE)

? REST (REST (FIRST (LISTEXAMPLE)))
&

@: FALSE
```

[the name FALSE is equivalent to the empty list]

```
? FUNCTION MAPFIRST (LIS),
    WHEN EMPTY (LIS), FALSE EXIT,
    ADJOIN (FIRST (FIRST (LIS)),
      MAPFIRST (REST (LIS)))
  ENDFUN $

? MAPFIRST (LISTEXAMPLE) &

@: (1 2 3)
```

The use of the "&" as a termination symbol instead of ";" indicates to the "DRIVER" function, which contains the main interaction loop, that the result of the computation should be printed as a list, rather than as a mathematical expression in standard mathematical notation. A "$" means the result should not be printed at all.

```
EXPRESSION: X + Y $

? EXPRESSION ;

@: X + Y

? EXPRESSION &

@: (+ X Y)
```

Internally, only the latter (list) form really exists. But this is almost totally transparent to the user. The "DRIVER" function mentioned above gets its input through a function called "PARSE", which normally expects all of ITS input to be in non-list notation. It is this PARSE function which really defines the difference between muSIMP and normal LISP. For example, if PARSE sees "FUNCTION" in the input stream, it translates everything from there to the next occurence of "ENDFUN" into a list, which is essentially the LISP equivalent of that function definition. In LISP, everything is a list, including function definitions. In muSIMP, PARSE translates both standard mathematical notation, and

muSIMP syntax itself into list/LISP form. So the difference between muSIMP and normal LISP exists only "at the surface".

When PARSE sees a single quote symbol, two things happen. First, it reads what follows in normal list notation. Second, it adjoins the QUOTE function name onto the expression, which causes the the expression to be "taken literally", rather than being evaluated. Thus in the definition of LISTEXAMPLE above, even if we had previouly made an assignment

```
ONE: UNO $ ,
```

LISTEXAMPLE would still only contain "ONE", rather than "UNO".

Once a function definition has been made, it becomes a part of the system, which can then be called upon by other function definitions. And that's how you bring muMATH into the picture, because new function definitions (and other kinds of valid muSIMP input), can be read in from disk as well as from the keyboard. For example, if you type:

```
RDS (ARITH, MUS);
```

with the source file ARITH.MUS on the disk in drive A, all the muSIMP source code constituting ARITH.MUS will be read in and made a part of the system just as COMBS and MAPFIRST were. Now you can do:

```
1/2 + 1/3 ;

@: 5 / 6
```

muMATH is a collection of 15 muSIMP source files, all but one of which depend on certain other source files being "in" at the same time. To do definite integrals, for example, you need to load in 5 of these source files.

Any time you want you can save the state of the system on disk including (in internal form) all the muSIMP code read in so far, just by typing

```
SAVE (<name>) $
```

[with a disk with enough free space in drive A], where <name> can be any available primary file name for your DOS. muSIMP will create a special memory image file with primary name <name> and secondary name SYS. Then you can come back any time amd just by typing (while in muSIMP):

```
LOAD (<name>) $ ,
```

you will be right back where you were just before the SAVE. Or, another way to do it is to type:

```
MUSIMP <name>
```

as a DOS command, in which case muSIMP will LOAD <name>.SYS for you before entering the DRIVER loop.

The package also comes with 10 other source files which are 90% comments. These are the lesson files which make the process of learning to use the system a great deal of fun, the lessons are so well-written. The lessons files are read in like any muSIMP source file, but they are echoed at the console and consist mostly of comments, delimited by matching percent signs. Every once in a while, the lesson gives you an exercise to do, ends the comment with a "%", and then makes the assignment:

```
RDS: FALSE $
```

That stops the input from the disk and gives you control of the system to do your exercise. When you're done, you type:

```
RDS: TRUE $
```

and the lesson takes over again. The lesson may also stop being a comment long enough to do an example, providing actual muSIMP source to the system. This example may, in turn, depend upon function definitions which you added to the system while doing a previous exercise. It all works beautifully as long as you don't try to stop in the middle of a lesson and save what you've done so far. It's possible, but tricky: you have to refer to the printed copy of the lesson to see what "finishing up" operations it would have done, do them through the keyboard, and finally SAVE a memory image as usual. When you come back, you'll still have to run through the lesson file from the beginning, but now you can skip the exercises you have already done.

Five of the lessons teach the use of those features introduced to the system by the ARITH and ALGEBRA files of muMATH. After completing them, you can use the other muMATH files without much difficulty, with the aid of the printed documentation. The remaining five lessons teach the muSIMP language--to a point. The trouble is, they are so beautifully written, you'll feel somewhat lost when, after the 5th lesson, you have to start learning from the printed documentation, which is much more terse than the lessons. The lessons really "take you by the hand"; the printed material works in a way which is actually similar to the way the system is structued internally: defining itself in terms of itself. It was a frightening experience at first, trying to learn from the printed discussion of muSIMP, but I finally got used to it. After a year and a half, I think I can appreciate the concise style. The documentation has been improved consiuerably, since the original muSIMP-79 version, but there is still room for improvement. The function READ is not documented at all (probably an oversight). Considerable information is included about the various property lists used by PARSE, but the definition of PARSE itself is missing -- I had to dig out the internal (LISP) form to see how it works. And it's only by seeing how PARSE works, that some of the documentation which IS included becomes intelligible.

muMATH can handle a wide variety of symbolic math problems, so whenever it fails to come up with an expected solution or simplification, it leaves you wondering why. The only way really to find out why is to delve into the source files themselves and uncover muMATH's methods. I've also found this to be one the best ways to learn about some important techniques which are not discussed in the lessons or in the printed documentation (such as storing function definitions on property lists). But another interesting approach is simply to experiment with muMATH. In the examples which follow, the symbol " --> " will mean "simplifies to."

```
8^(1/2) --> 2^(3/2),
```

but

```
343^(1/2)
```

doesn't simplify. This is because ARITH works with a very short list of primes: (2 3 5). If you reassign:

```
PRIMES: '(2 3 5 7) $
```

then,

```
343^(1/2) --> 7^(3/2).
```

Some complex arithmetic is possible [#I is i]:

```
(1 + #I)^2 --> 2*#I,
```

but

```
(2*#I)^(1/2)
```

appears to be to much to ask, even with both trig files loaded.

With the "LOG" file loaded,

```
LOG (2^X, 2) --> X,
```

[the second argument is the base], and

```
LOG (8, 2) --> 3
```

[an improvement in the latest version].

muMATH forces its user to do a certain amount of crucial decision-making, by setting what are called control variables. For example, if the control variable NUMNUM has the value 2,

```
2 * (X + 1) --> 2 + 2*X,
```

and if NUMNUM is -2,

```
2 + 2*X -->  2 * (1 + X).
```

If NUMNUM is 30,

```
(X + 2) * (X + 3) -->  6 + 5*X + X^2,
```

but, and this is perhaps muMATH's one biggest weakness, it cannot factor even a simple quadratic such as the one above. And yet, with the equation-solving files loaded,

```
SOLVE (X^2 + 5*X + 6 == 0, X) -->

{X==-2,
 X==-3},
```

even though the former task is easily reducible to the latter. They say (in one of the lessons) that they're working on it for future releases.

You would expect:

```
X/(X*Y) --> 1/Y,
```

but one day I coulndn't seem to to make it happen. I finally realized that the problem lay with the control vartiable EXBAS which needs to be a positive multiple of 2 in order for the cancellation to work. Now EXPBAS being a positive multiple of 2 is supposed to allow such things as:

```
(X*Y)^2  -->  X^2 * Y^2.
```

In order to make sense out of all this you have to realize that internally, a fraction A/B is equivalent to A * B^-1 [actually the list: (* A (^ B -1))], so the X*Y in the denominator of the cancellation example is really a factor of the form: (X*Y)^-1, which explains why the state of EXPBAS is so crucial, in a problem which seems to have nothing to do with exponents.

The control variable DENNUM has to be a negative multiple of 15 in order to get:

```
1/X + 1/(X + 1) -->

(1 + 2*X)/(X * (1 + X)),
```

but then DENDEN has to be a positive multiple of 3 if you want the denominator multiplied out, and so it goes. Guiding muMATH through a tricky problem of adding rational expressions sometimes seems to take as much skill as doing the problem yourself! Of course, muMATH is especially impressive with a really tedious problem like (1 + X)^20 [PWREXPD must be 2].

I have found muMATH's trig and calculus capabilities to be surprisingly good: muMATH will differentiate just about any function you give it, since it

knows all the standard rules. Integration is, of course, the real test.

```
INT (1/(X^2 + 5*X + 6), X)  -->

LN ((-4 - 2*X)/(6 + 2*X)),
```

which can only be further simplified by a subsequent reevaluation of the answer, with NUMNUM and DENDEN set to -2. A careful study of the two integration source files, reveals that the above problem is done by completing the square. Since muMATH can't factor quadratics, there's no way it could do the problem by partial fractions. But it's clear, even without examining the source files, that muMATH really doesn't know anything about partial fractions, since it can't integrate

```
1 / (X^3 + 1),
```

even if you do the factoring for it. muMATH appears to be able to do integration by parts, succesfully integrating such things as

```
X^2 * #E^X
```

[#E is e], but if you look at the source code, what you find is that muMATH knows lots of those special reduction formulas, which one normally derives using parts. Surprisingly, muMATH can't integrate

```
X * (1 + X)^(1/2),
```

which is easy using parts.

The latest version of muMATH (muSIMP/muMATH-80) includes three source files which were not in the previous version: TAYLOR.DIF, LIM.DIF, and SIGMA.ALG.

The file TAYLOR.DIF, which consists of a single function definition, generates Taylor polynomials. For example,

```
TAYLOR (TAN (X), X, 0, 5) -->

X + X^3/3 + 2*X^5/15 .
```

Expansions about points other than 0 are "multiplied out" whether you like it or not. This turns out to be inherent in the algorithm used, but one can easily define:

```
FUNCTION TAYLOR1 (EX, VAR, POINT,
  ORDER),
  EVSUB (TAYLOR (EVSUB (EX, VAR,
    VAR+POINT), VAR, 0, ORDER), VAR,
    VAR-POINT)
ENDFUN $
```

Now,

```
TAYLOR1 (LN (X), X, 1, 5) -->

-1 + X - (-1+X)^2/2 + (-1+X)^3/3
 - (-1+X)^4/4 + (-1+X)^5/5 .
```

LIM.DIF, as its name suggests, calculates limits, and utilizes muMATH's differentiation machinery to do so. For example:

```
LIM ((X - SIN (X)) / X^3, X, 0) -->

1 / 6 .

LIM ((1 + 2*X)^(3/X), X, 0) -->

#E ^ 6 .

LIM (LN (X) / X, X, PINF) -->

PZERO .
```

In the first two examples, the limit is understood to be from the right (otherwise an additional argument to LIM must be used). The last example shows the use of PINF, to denote positive infinity, and PZERO, to indicate that the

limit of 0 is approached through positive values. If we ask muMATH to do a more general example, e.g.

```
LIM ((1 + A*X)^(B/X), X, 0);
```

a curious thing happens: muMATH starts asking for additional information!

```
@:
???  A  ???
ENTER SIGN (0 + -)?
```

A single character response of "+" is accepted and the computation immediately resumes, finally producing the expected answer,

```
#E ^ (A*B) .
```

As you would guess, the file SIGMA.ALG does summations (and products too). For example:

```
SIGMA (K^2, K, 1, 100) --> 338350

SIGMA (1/K^2, K, 1, 20) -->

17299975731542641 /
10838475198270720
```

In some cases, it can even handle a sum with a variable number of terms:

```
SIGMA (K^2, K, 1, N) -->

(1+N-3*(1+N)^2+2*(1+N)^3) / 6

SIGMA (1/2^K, K, 0, N-1) -->

(-2+2^(1+N)) / 2^N .
```

Incidentally, if you want the result in another form, you must RE-evaluate the answer with appropriate flag settings. For example, the above result was obtained even with DENNUM = 3. But then,

```
EVAL (@) --> 2 - 2^(1-N)
```

[The atom @ is always bound to the result of the previous computation]. If LIM.DIF is loaded along with SIGMA.ALG, the prospect of infinite series arises. As far as I can tell, this is limited to geometric and telescoping series (we can't expect miracles). Also, it's best NOT to have LIM.DIF loaded unless you need it, as it interferes in an odd way with SIGMA's finite sum activity, e.g. asking (twice!) for the sign of - LN(2) during the evaluation of a finite geometric series with common ratio 1/2.

When I first started in with muMATH, it was the symbolic math which fascinated me most. And it is indeed an amazing accomplishment, especially considering the limited memory space. Also, I understand that there are problems in physics and engineering which can only be rendered tractible by using a combination of numerical and symbolic methods. I really don't know how useful muMATH would be to a person who doesn't already know the relevant mathematics. I would love to see what would happen with muMATH in the hands of my beginning algebra students, but I haven't yet had that opportunity. I really believe a person could learn some mathematics just by learning to use muMATH, but this is only a speculation.

What excites me most now is the muSIMP language itself. Programming in muSIMP involves using, in a very concrete way, the same techniques of inductive definition, building complex structures from simple ones, and then analyzing and operating on those structures inductively, that are fundamental in theoretical work in set theory and logic. It also turns out to be much more useful as a general purpose language than I originally suspected, particularly now that you can easily link muSIMP to your

own machine language routines. muSIMP is also an excellent bridge between "traditional" programming languages and LISP. Based on my own experience, I think it would be very easy and natural for anyone accustomed to e.g. Pascal, or PL/I to start right in with muSIMP, but then when he or she subsequently (I would say inevitably) discovers LISP, it will by then seem natural too. I feel, however, that any further discussion of the muSIMP language should be conducted in the context of LISP itself and LISP surface languages generally.

The muMATH package is certainly an impressive piece of work. Although I am particularly interested in its potential as a teaching tool, it will undoubtedly find uses in other areas as well. But at any rate, the time I have spent with muMATH and muSIMP has been challenging and a great deal of fun.

AUTOMATION OF REASONING

CLASSICAL PAPERS

IN

COMPUTATIONAL LOGIC

vol I and vol II

Jörg H. Siekmann, Graham Wrightson (eds.)

It is reasonable to expect that the realtionship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last.

John McCarthy, 1963

Logic has emerged as one of the fundamental disciplines of computer science.
Computational logic, which continues the tradition of logic in a new technological setting has led to such diverse fields of application as automatic program verification, program synthesis, question answering systems and deductive databases as well as logic programming and the fifth generation computer system.

This series of volumes, the first covering 1957 to 1966 and the second 1967 to 1970, contains those papers, which have shaped and influenced the field of computational logic and makes available the classical work, which in many cases is difficult to obtain or had not previously appeared in English. The main purpose of this series it to evaluate the ideas of the time and to select those papers, which can now be regarded as classics after more than a decade of intensive research.
Contents: 60 original papers. 3 survey papers. Complete bibliography on computational logic.
To be published by Springer Verlag, Berlin, Heidelberg, New York, 1982.