Record of the First Annual Maple Workshop June 11-12, 1983 Wild Echo Lodge Port Stanton, Ontario

Edited by

B.W. Char, K.O. Geddes & G.H. Gonnet

Research Report CS-83-31

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Record of the First Annual Maple Workshop June 11-12, 1983, Wild Echo Lodge, Port Stanton, Ontario

Edited by: Bruce Char, Keith Geddes, Gaston Gonnet

Participants

Stephen Watt, Carolyn Smith, Sophie Quigley, Michael Monagan, Benton Leong, Erich Kaltofen*, J. Howard Johnson, Marta Gonnet, Gaston Gonnet, Keith Geddes, Greg Fee, Bruce Char, Robert Bell

Department of Computer Science
University of Waterloo
Waterloo, Ontario

*Department of Computer Science
University of Toronto
Toronto, Ontario

ABSTRACT

This is an edited record of the discussion at the Maple retreat during the three formal working sessions. Most of the discussion was focussed on extending the Maple language[Ged82a, Cha83a] to deal with mathematical manipulations it currently does not handle conveniently, as well as the implementation issues arising from such extensions. Three main topics were discussed: handling matrices characterized as conglomerations of matrix sub-blocks, possible ways of handling evaluation and unevaluation of expressions, and operator algebra.

1. Session 1: 9:30am-noon June 11, 1983

1.1. Matrices and arrays in Maple

The discussion started by considering the problem of manipulating matrices which are conceived as having an internal structure of submatrices of assorted sizes. This problem has been mentioned as something that the predecessors of Maple do not handle well. For example, we might be thinking of a matrix A, consisting of three interesting blocks, $A1$, $A2$, and B:

$$A = \left(\begin{array}{c|c} A1 & A2 \\ \hline \mathbf{0} & B \end{array}\right)$$

Typically, we know that

$$\text{rows}(A1) = \text{rows}(A2) \qquad \text{cols}(A1) = \text{cols}(\mathbf{0})$$

but we may not know (or choose to define) more than that. Even so, algebraic manipulation of such objects is found to be useful in such applications as the numerical analysis of linear algebra algorithms, or with Strassen's matrix multiplication algorithm.

It was noted that certain non-symbolic languages, such as PL/1, Algol68, and APL can define and name sub-blocks of a matrix. For example, Algol68 allows subscripting and trimming (collectively called slicing) as in:

```
[1:20, 1:20] real x
y := x[2:4, 18:20]
```

in which case y is a 3-by-3 matrix. Maple at this point has arrays and tables, [Wat83a] but does not have any knowledge of matrices or arithmetic on them. The obvious way to represent a matrix in Maple is by a two dimensional array.

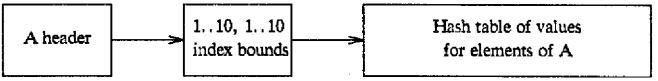

Figure 1 -- schematic representation of a 10 by 10 matrix A

Using the natural representation has the advantage that unless the "Table of values" has information to the contrary, "A[i,j]" will evaluate to "A[i,j]". If B is a submatrix of A, the indexing header/descriptor of the matrix B contains a description of how to map an address of B into the corresponding address in the Table of values for A. This is similar to PL/1 with definite array bounds.

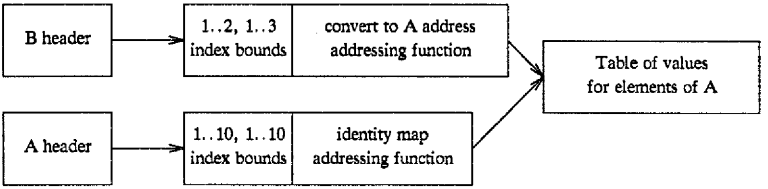

Figure 2 -- schematic representation of A and submatrix B

However, this does not handle the situation when the dimensions of A and B are not known, or intentionally symbolic. In such a case, all that may be declared to the system is that m and n are positive integers which are symbols for the number of rows and columns of A respectively.

What happens with assignments to elements of such objects, such as a[i,i]:=3, where i is a symbol? If we allow such assignments, (which all agreed would be useful), then we may want to first use i as a symbolic index, and then later give it a value. For example, consider the following sequence of assignments:

```
a[a[i]] := 4
  a[i] :=3
    i:=3
```

What happens when the value of a[i] (that is, a[3]) is requested? Is it 4 or 3? (Or something else? If this is tried for a Maple table currently, the results of an assignment to a symbolic element a[i] are only retrievable, subsequent to the assignment of a value to i, by the table reference a['i'].)

One school of thought (the "programming viewpoint") is that assignments to arrays or matrices should be time stamped, and the most recent assignment used, so that the semantics

of the above should be the same as the sequence a[3]:=4; a[3]:=3; -- 3 should result.

Another school of thought (the "mathematical viewpoint") is that when i is assigned a value, either of the previous assignments has equal "right" to be the one which actually takes effect. Thus, the result of the above is unpredictable/unspecified (either 3 or 4).

Another situation where this might arise is when the user wants to define a matrix as "all zeroes, but with the diagonal equal to one". This can be viewed as two separate assignments, one modifying the other (the programming point of view). Or one could just use a single "pseudo-mathematical" definition such as occurs in vaxima with "a[i,j] := if i=j then 1 else 0".

1.2. Brief interlude: discussion on infinite objects

Infinite objects occur in many mathematical situations. For example, one may define a set via:

$$a \leftarrow \{b\}$$
$$b \leftarrow \{a\}$$

This object can be finitely represented as:

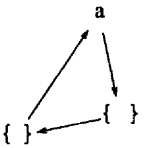

Figure 3 -- representation of "{{{{{ .... }}}}}"

There are other more useful instances of an infinite representation, such as using $x \leftarrow 1+\sqrt{x}$ as a way of representing the equivalent notion

$$x=1+\sqrt{1+\sqrt{1+\sqrt{1+\cdots}}}$$

or to represent a Taylor series by the function that describes how to produce the n-th term from n.

The point was established that users of Maple might want to manipulate and program with such things (e.g. manipulating infinite power series until one decides on a definite order, or calling upon a stream of values using "lazy evaluation" -- see [Tur82a]). While not strictly necessary, computing with infinite objects would give an expressiveness to the language not currently available.

Another observation about the infinite was made regarding Maple's current problem with the fact that x := x+1; x; causes a fault and termination. The observation was made that a student-oriented, careful checking system analogous to WATFIV would do the necessary work of marking the stack to check for recursive definition of a name, but that other versions may not. It was left unclear why such checking can't be done efficiently and why such checking should not be put into any interactive version of Maple. (Are non-students less likely to be upset when their Maple dies?).

1.3. Back to arrays and matrices

Currently, Maple arrays/tables are all implemented as hash tables -- they have no intrinsic fixed size or subscripting scheme that requires numerical declaration. It is thus easy to implement objects with symbolic dimensions based upon the current implementation of tables*. In general, the "symbolic matrix with symbolic structure" problem would start being an implementation problem when the user defined the structure of the matrix -- subblocks of a matrix A, some of whom overlap. One might imagine the following sequence of specifications:

```
a := array (1..n, 1..n)          #n a symbol
b := subarray(a, i..j, i..j)     #b is a subarray of a, row and
                                 # columns i through j. i and j are
                                 # symbols
c := subarray(a, k..l, k..l)     #c another subarray of a.  k and l
                                 # again are symbols
b := {all zeroes}
c := {identity matrix of size l=k+1}
```

followed by assignment of particular integer values to i, j, k, and l, and then usage of a particular element of a, say, a[20,19].

The most definitive suggestion for implementing submatrices was to treat the indexing and table for a matrix and all submatrices as one object, with pointers into the object made by all:

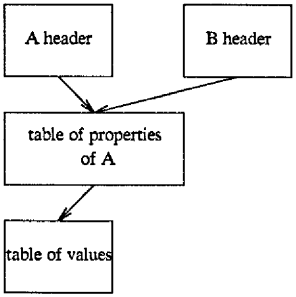

Figure 4 -- Representation of matrix/submatrix as one object

The "table of properties of A" would contain both dimensioning information and indexing information for A and all submatrices. Changes in dimensions or indexing would alter the table of properties. Presumably conflicting specifications for various submatrices would be resolved by inspecting the table. This would be okay for changing indexing functions, but not for array element assignments.

Another viewpoint is that a matrix a could be viewed as strictly the union of various

* Erich made the observation that we seem to spend a lot of time worrying about how to implement a feature, long before we are finished discussing how we want the feature to work. He noted that in the Scratchpad work, usually feature design was mainly completed before implementation details were worried about. The danger with the latter approach is that one may want to conceive of the system as being based upon features that can't be implemented efficiently. The danger of the former approach is that the system starts to take on an ad-hoc flavor -- the justification of the design is that it was easy to get to work, no matter how many problems it causes for users. Avoiding falling into one of these traps requires constant vigilance.

submatrices, some with symbolic names, and some without. The matrix m would therefore have few properties itself; all such properties would be subordinate to the properties of the submatrices, which would really do the work of defining the matrix. Implicitly this approach will only work for disjoint unions, it is not clear how this would resolve the "overlapping submatrix" problems.

Yet another view of submatrices is to view them as matrices of matrices. Thus the object in Figure 1 would be viewed as a 2 by 2 matrix, each element of which is a matrix of random size. It is not clear how this view allows one to do symbolic matrix arithmetic with two matrices where the submatrices of one matrix are not the same size as those of the other.

2. Session 2: 3pm-6pm, June 11

2.1. Parameter passing in Maple

Sentiments were expressed that parameter passing for Maple procedures is still not "right". The design should be motivated by more conceptual elegance.

For those who had never thought it through, Gaston presented a fairly precise model of parameter passing in Maple.

Originally, when a procedure was invoked, a copy of the entire procedure was made and each actual parameter (without evaluating it) was substituted into the procedure everywhere the formal parameter occurred. This was an implementation of "call by name" in one of its purest forms but there were various difficulties: (a) Some variables are environment-dependent, so something like f(a,b,") was causing the " to be substituted textually inside the procedure, producing a result which was hardly ever the desired one. (b) There is an intrinsic lack of efficiency in this approach; for example, f(very_expensive_function(x)) may produce several evaluations of the expensive function (one each time the first parameter of f is evaluated). The writer of f(..) should not be held responsible for this. (c) The copying process in itself is very expensive. Sometimes large functions do some testing and return immediately without ever executing most of the code, but the copying has to be done for each parameter, and everything "higher up" in the expression tree, which means effectively copying everything.

It should be noted that parameter passing in a symbolic language is much trickier than in normal programming languages due to the lack of a final "evaluated" form for expressions.

The current solution is to fully evaluate parameters at the time of procedure call, placing the values in an internal table to be consulted during invocation of the procedure. This has the same effect as the previous version except for parameters which depend on the environment. Most of the efficiency problems of the previous version are eliminated since parameters are evaluated only once (although parameters which are never used are evaluated needlessly). "Naive" users sometimes encounter this "feature" of Maple when one of the parameters to a procedure is an unevaluated name to which a value is to be assigned, but the user also uses that parameter like a programming variable in the procedure code as in the following example.

```
f := proc (x,y,returnval)
        returnval := x**2;
        while foo(..) do
                . . .
                . . .
                returnval := returnval + 1
        od
end;
```

Figure 5 -- Example of erroneous use of formal parameter in Maple

A legal version of the above procedure would be:

```
f := proc (x,y,returnval)
        local t;
        t := x**2;
        while foo(..) do
                . . .
                . . .
                t := t + 1
        od;
        returnval := t
end;
```

Figure 6 -- Example of correct use of formal parameter

or alternatively, forcing an evaluation by the use of " as follows:

```
f := proc (x,y,returnval)
        returnval := x**2;
        while foo(..) do
                . . .
                . . .
                returnval;
                returnval := " + 1
        od
end;
```

Figure 7 -- Alternative example of correct use of formal parameter

2.2. The uses of the notion of "variable" in symbolic computation languages

Variables in symbolic algebra are used for three main purposes: (a) as *programming variables* whose values are other expressions, constants, etc. during the process of computation; (b) as *mathematical symbols* or names with no value assigned to them; (c) as *macros* which stand for a pattern (or expression tree) of symbols without evaluation.

The concept of "lazy evaluation" can be viewed as the mode of "macros" as mentioned in (c) above. Basically, there would be a syntax for setting up macro names, such as

$$\text{p is } x^2 + 1.$$

This would build an expression tree labelled "p":

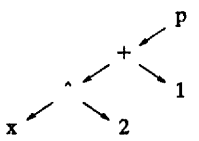

Figure 8 -- expression tree for p

A sequence of assignments (macro-definitions)

p is $x^2 + 1$
q is $p + 5$

would just build a structure for q that would include a reference to p, without doing any so-called "evaluation":

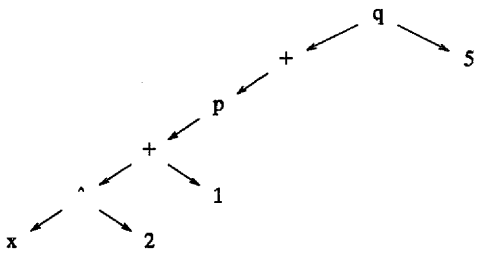

Figure 9 -- lazy evaluation expression tree for q

If one wanted to actually view the "value" of q, that would be evaluation, and require more work (in this case, just following pointers around). Unlike the current scheme of evaluation/simplification, nothing would be done until requested, in which case everything might be done. Maple does not explicitly differentiate between modes (a) and (b); the differences arise simply by the assignment (or unassignment) of names. As for mode (c), Maple supports a limited (one-step) "lazy evaluation" realized by the quoting mechanism. In some cases where mode (c) is needed for more than one step of computation, the quoting mechanism has been found to be inadequate.

Another observation was that the problem in symbolic manipulation language semantics comes from mixing evaluation with expression formation. The "lazy evaluation" proposal could explicitly separate the two; e.g.,

p is $x^2 + 1$
x is 5
q is $p^2 + 1$

would just create expressions, with the representation doing a job of representing what was typed in. It would be only on an explicit call to the evaluator, such as typing "eval(q)" after the above definitions had been entered, that the value of q as 677 would appear.

Someone then pointed out that the macro definition of "x is 5" in the above sequence should be illegal, since when the prior "p is $x^2 + 1$" was entered, x was considered to be a mathematical variable and hence should be an illegal choice as a programming variable or a macro name.

There was not total agreement that having a language with mathematical symbols, programming variables, and macro names completely disjoint was a good thing from an "ease of usage" point of view. For example, consider the problem of "undetermined coefficients". One has, say, a polynomial equation in a variable x. The polynomial's coefficients involve expressions in symbolic constants and actual numbers. Determining the coefficients means solving the system of equations that results from setting each coefficient equal to zero. Of course, when the solution is known, one wants to convert the symbolic constants into programming variables, and assign those variables the values derived. It was also noted that "lazy evaluation" would not be a desirable mode very often. An inspection of the functions coded in the Maple library shows that a majority of the steps require full evaluation.

Another proposal put on the floor was to represent expressions in terms of a pattern, and a list of variables that would name the slots of the pattern. For example, $x^2 + 1$ would be represented as:

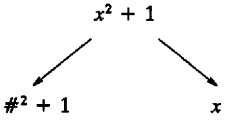

Figure 10 -- Expression tree with slots

This would seem to make things easier when pattern-directed simplification is implemented, since the pattern would be free of actual variable names. However, such pattern-matching might have a difficult time with constants, which might be in a pattern as in "look for an explicit but arbitrary constant"; i.e., it only captures one property of a symbol, namely its occurrence.

As a more complicated example, consider the following sequence of expression creation:

$$p \leftarrow x^2 + 1$$
$$q \leftarrow p^2 + y + x + 1$$

has q pointing to

$$(\#1 + 1)^2(\#2) + \#1 + \#2 + 1$$

as the pattern part, with "#1" being "x", and "#2" being y. However, the choice of which was "#2" and "#1" was arbitrary (i.e., implementation dependent).

A final point made was that in the latest profiling tests, equal time was spent in the "eval","simpl", and "search" portions of the Maple internal code during execution of the Maple tests which exercise all the features (although not necessarily in the same proportion as "real" users do). Thus, while the preceding discussion assumed that the current implementation of Maple was spending an excessive amount of time in evaluation (needlessly traversing expression trees because full evaluation is applied to each expression), the profiling results contradict this impression. Maple spends some time in its evaluator, but this amount is only about 10% of the total time, on average.

2.3. Freeze and thaw -- deferred evaluation

Discussion then turned to one of the thornier problems of evaluation in Maple, namely that because Maple always "fully evaluates" expressions before, say, giving something to a procedure as a parameter, one often finds the need to delay or totally prevent evaluation/simplification. Listed were 15 problems associated with "evaluation" that must be handled:

1 *Call-by-name*. For example, *divide(a, b, 'q')*. This case is the most common use of quotes in Maple. It involves passing a name into a procedure, and this name may be assigned a value before return from the procedure. Morven had previously pointed out that an alternative to this is to return a list of values (as would the Lisp community). One drawback of returning lists of values is the overhead of unbundling the values into individual variables after the return. Another drawback is that all values will always be computed, whereas with an extra optional parameter, the procedure can usually avoid some computation if the optional parameter is missing.

2. *Unassignment*. `x := 'x'` resets the value of x to just a symbol from programming

variable usage. Once again, the current notion in Maple is that a quoted variable is the name of the variable, and that the notion of a mathematical symbol can be simulated by a programming variable which has its own name as its value.

3. *Unassignment of concatenated names.* for i to 10 do a.i := evaln(a.i) od; This is similar to the preceding case, but the above trick does not work because 'a.i' would remain completely unevaluated. What is needed here is that i must be evaluated, then the concatenation a.i must take place, with no further evaluation of the resulting name. Maple's evaln function achieves this "evaluate to a name" construct. It should be noted that the use of quoting in both of the preceding cases could be replaced by the use of the evaln construct.

4. *FAIL or RETURN('f(paramseq)').* Returning the procedure call itself when a routine can't find the answer. This is a pure example of preventing the evaluation of an expression.

5. *Delaying expensive arithmetic.* For example, 2^{10000}. Sometimes it is expedient to delay the evaluation (or simplification) of an expression because it may not be needed in subsequent computation, but if it is needed then evaluation/simplification will take place when the expression is referred to.

6. *Unwanted simplification.* For example, $108 = 2^2 * 3^3$. This result would be produced by the ifactor routine, but if the factored result is returned normally, Maple's simplifier will immediately "simplify" the product of powers into a single number. Maple's internal simplifier will simplify 2*3 and '2*3' to 6. Using two levels of quotes "2*3" will prevent simplification because the evaluator strips off one level of quotes, so in the latter case the simplifier sees '2*3' which it leaves untouched.

7. *Preventing expansion.* For example, *expand(*$\frac{(1-x)^3+(1-x)^4}{1-x}$,$1-x$). Here, Maple's expand function has its own "freezing" mechanism by which any expression can be regarded as though it were temporarily substituted by a name, simply by listing the expression as an extra parameter.

8. *Freezing for factor, normal, gcd, etc.* For example, *factor* ($exp(x)^2-1$). Here it is desired to replace subexpressions such as exp(x) by frozen names, for the purposes of computing factor, normal, gcd, etc. This is similar to the preceding example, but unlike the expand function there is currently no solution for these other cases.

9. *String concatenation.* For example, `This is ` . 'a' . `string`. There is no separation in Maple between the concepts "name" and "string". Occasionally one gets tripped up by the fact that the string `a` is actually the same thing as the name a, which has been used as a programming variable and has a value.

10. *Automatic loading.* For example, *gcd := 'readlib('gcd')'*. Delayed evaluation is used here to get the construct which some other languages call "autoloading". The inner quotes are necessary to prevent a recursive evaluation loop and the outer quotes prevent the evaluation of the readlib function until such time as the name "gcd" gets evaluated.

11. *Premature concatenation.* For example, *sum('a.i'*x**i, i = 0 .. n).* Here we want to prevent the concatenation of a.i into ai, until within the sum function i takes on specific values.

12. *Preventing op.* In the application of Maple to the implementation of a relational data base, it is desired to construct a tuple selector. A tuple selector may look like: [op(3,X),op(8,X),op(1,X),...]. The construction of such selectors is typically done in loops, and if any of the op's evaluates prematurely it fails. This selector is then used in constructs of the form:

    map(proc(X) [selector] end, Relation) .

    The solution that was adopted for this case was to construct the selector using OP which does not evaluate to anything, and then before doing the map the assignment OP := op

is done.

13. *Tracing*. In tracing by setting the value of printlevel high, the trace of map('absLc',...) prints out the name "absLc" as it is applied during mapping. However, the name "unknown" is printed out when tracing map(absLc,...) because the name "absLc" is unknown within the map function - only its value (a procedure body) has been passed into the map function.

14. *Recurrences*. For example, *eqi := 'T.i + diff(T.(i−1), x)*x'* This is an expression from orthogonal polynomial definition. If the expression is not quoted, the diff function is evaluated at the wrong time.

15. *Substituting into an unevaluated diff*. For example, $f'(0)$ means $diff(f(x),x)|_{x=0}$. An attempt to handle this situation by using the construct

    subs(x = 0, diff(f(x),x))

    is incorrect. (We are concerned with the case when f is undefined). This is really an operator problem (operators are discussed in section 3.4 of this report).

Discussion then turned to how the various ideas suggested so far could deal with these problems.

The "p is" macro definition style might solve problems like (5), e.g.

y is 2^1000

would just create the structure. Evaluation would not occur unless macro expansion was explicitly asked for. This just defers the problem one more stage, since if all expressions are initially created with macros, one will sometimes want to macro-expand everything except structures such as 2^1000. How will one be able to specify something like that without another level of quoting or evaluation fences?

The same macro definition style would perform "unassignment" as in (2) (reset from programming variable usage to math symbol) by

x is x .

This would create a circular pointer, but presumably the implementation would be able to detect circularity at macro-expansion or printing time, and do the right thing.

For (7), it was noted that the use of expand is a little different than the proposed syntax for factor: expand(..., 1−x) has the notation for what syntactic subexpressions should be frozen with respect to the expansion, while example (8) -- factor(exp(x)**2−1) -- doesn't.

There was a consensus that there were three kinds of evaluation in normal usage:

-- evaluate now, during execution of the statement (what Maple usually does)

-- hold off evaluation "for a moment" (perhaps until the next time the expression comes under scrutiny of the evaluator - the usual effect of quoting things in Maple)

-- don't evaluate until told to: blocked evaluation during manipulation

The observation was then made that some of the examples were really not evaluation problems but actually semantic problems. For example, "diff(f(x),x) evaluated at x = 0" should not be treated as a case of controlling lexical substitution of 0 for x in the expression, but rather the concept of "evaluation at x = 0" is fundamentally different from lexical substitution.

Previous Maple meetings had brought out the preliminary concepts of "freeze" and "thaw" to handle "don't evaluate until told to". The basic proposal is that "freeze(expr)" turns the object into a special name that hides all mathematical properties of expr. For example,

freeze(1) + freeze(2) + freeze(3)

would be an expression where the arithmetic of numbers would not be performed, just as if this was the sum of three different symbols.

A "tfreeze" concept was proposed, to promulgate freezing of limited dynamic scope.

tfreeze (f, expr, vars)

would freeze the expressions in "vars" when invoking the procedure f with argument expr. The syntax for this needs to be cleaned up, but it is obvious enough that what is desired is a mechanism to freeze certain things and then automatically thaw them upon return from the procedure. There might be problems with unconditional thawing, however.

The "thaw" command proposal would remove the "hiding" of the properties of the frozen objects. For example,

thaw(expr)

would thaw everything in the expression "expr".

Erich was perturbed by the fact that the freeze concept was hiding all properties of the frozen object, as opposed to just presenting a wall through which evaluation would not pass, but rather go around. Some people viewed as a "feature" that diff(frozen-expr-in-x , x) would return 0, while others viewed this as a bug.

At this point we had reached the following solutions to the 15 problems listed above.

Problems 1-3: Use evaln, but in problems 1-2 the use of single-quotes is also valid.
Problems 4, 9-11, 13, 14: Use single-quotes to prevent evaluation.
Problems 5, 6, 12: Use the new freeze/thaw mechanism.
Problems 7, 8: Use the tfreeze (temporary freeze) mechanism, which should be a general "front-end" procedure.
Problem 15: This is an operator problem, discussed in section 3.4 of this report.

Can one assign to frozen variables? Since the implementation proposal suggests that they are just another kind of name, the answer should be "yes". This would be a "possible, but why would you want to do it?" sort of feature. The question then naturally arises, "why not make it illegal?"

A frozen "x+1" therefore would look like

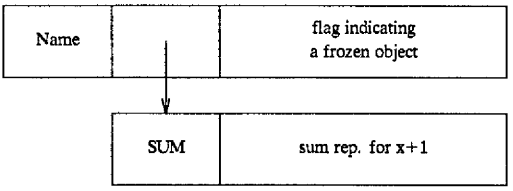

Figure 11 -- frozen object "x+1"

One could assign a value to the frozen object, via "assign(frozen(x+1), x+2)", (assign evaluates both arguments):

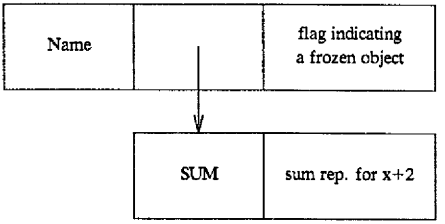

Figure 12 -- result of assign(frozen(x+1), x+2) -- a frozen(x+2)

An assignment to a frozen object changes what is being frozen.

An alternative approach to freezing would be to hide the frozen expression in its name:

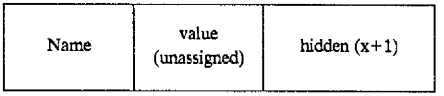

Figure 13a -- alternative representation for frozen(x+1)

where assign(frozen(x+1), x+2) produces an object whose value is x+2, but will still 'thaw' to x+1.

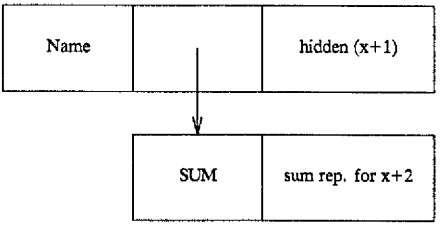

Figure 13b -- alternative result of assign(frozen(x+1), x+2)

3. Session 3, 9am-noon, June 12, 1983

3.1. More on freezing and thawing

The discussion returned to a few loose ends from the previous day's discussion. The question was asked, if you freeze an "x+1" and then subsequently another "x+1", is the same frozen object generated in both cases? The consensus was that in some applications they must be the same, and in other applications they need to be treated differently. It was eventually concluded that equivalent expressions must generate unique frozen objects, but there would have to be a concept of "levels" of freezing so that one could thaw expressions that were frozen at "level 2" without thawing expressions frozen at "level 1", for example. Both concepts (freezing with level or without) have practical drawbacks.

3.2. Short-circuit evaluation

A different observation was made about Maple's evaluation process. It was noted that not all clauses/terms in a boolean conjunction/disjunction/product need to be evaluated if, as they are evaluated in sequence, one of them is false/true/0. For example, 0*int(....,x)*.... could skip a (possibly expensive) call to "int" and recognize the result of zero immediately. Doing "short-circuit" evaluation (actually putting in a little simplification knowledge into the evaluator) has the consequence that side-effects that arise during the evaluation of the skipped terms will not occur (such as if int sets some global variables during its invocation). Thus, while attractive for efficiency purposes, it is not clear that whatever gains there are outweigh the possible change in semantics.

3.3. On the transformation 0**i → 0

Discussion turned briefly to Fateman's "language-independent bug", namely that

```
sum(a[i]*t**i, i = 0..n);
subs(t=0,");
```

returns zero, because of the (erroneous) simplification of 0**i to zero BEFORE any knowledge of the range of i is introduced. Greg mentioned that the usual convention is 0**0 → 1 rather than 0**0 → 0. (Note that for an explicit 0**0 Maple gives a "division by zero" error). Keith noted that this example is a representation of a polynomial in t with constant term a[0], and one expects the result of substituting zero for t to be the constant term a[0]. Erich was of the opinion that algebra systems make mistakes in analysis because they are algebra systems and not expert systems in analysis, but that was okay*.

3.4. Notation for operator algebra

Consider the partial derivative of f(g(x,y)) with respect to x. Unfortunately, barring a convention that assigns standard names to arguments to a function based upon their position, we have something like

```
diff(f(g(x,y)), x) --> at( partial(f,1), g(x,y)) * at(partial(g,1), (x,y))
```

(assuming that y is independent of x). The problem with the notation is that there is no name for the argument to f in the first partial, and that "evaluation at" for a function is currently not defined in maple. Attempting to use the "subs" function for the "evaluation at" operation leads to erroneous results.

The suggestions for notation for the partial derivative were:

```
diff(f,1) (x)
```

$$f_x(x)$$

$$f_1(x)$$

Given that differentiation is well-recognized as an operator upon functions, such that mathematicians like talking about "Df" without any reference to arguments of f at all, discussion turned to operators as a new kind of object in maple. It was recognized that while operators might not be an ordinary kind of symbol in maple, one still wants the ability to write mathematical expressions involving them: with E and F operators, one might write "2*E", "E+F", or "diff(E)" (whatever an operator applied to another operator does). Given that functions operate upon ordinary Maple expressions or constants, a first crack at notation, say,

```
( a + e + 1 + partial(f,1)) ( 7 )
```

shows that a constant as a function (say, "1", or a symbolic constant "a"), has a different

* Gaston thinks that above remark should be framed and placed in a prominent place at our meetings.

```
<x**2> (5)    →   25
<sin(x)> (y)    →   sin(y)
<x + cos(y)> (a,b)    →   a + cos(b)   (or  b + cos(a))
<sin> (x)    →   x
```

(Note that the latter case is an example showing that <a>, for any name a, denotes the identity operator). The underscore operator is used for the distinct concept of applying functions to arguments where no "procedurization" is involved, as in:

```
(a+b)_(x)    →   a(x) + b(x)
(sin)_(x)    →   sin(x)
```

A further refinement to the operator notation is the ability to specify which variables become "parameters" to the "procedurized" expression, and in which order. The extended syntax is:

```
<expr | x,y,...>
```

where the names x,y,... are to become the "parameters" of the procedurized expression. For example:

```
<f(x) + a | x> (b)    →   f(b) + a
<g(x) + x**2 + 3 | x> (3)    →   g(3) + 12
```

Note that the previous example <x + cos(y)> (a,b) would use indets(x + cos(y)) to determine the "parameters" (and their order) for the procedurized expression, and hence the result depends on maple's ordering of the elements in a set.

It was noted that <expr>_(args) should be equivalent to <expr>(args). More generally, the underscore operator would be required unless the left operand is an operator (i.e., in angle brackets), a name, or a procedure. It was further noted that <c>(x) → c and (c)_(x) → c for any constant expression c. The identity operator is <a> for any name a. Products (and powers) of operators will denote composition, since that is the only logical interpretation when the operands are operators. Thus there is a distinction between <expr>**2 and <expr**2>, the former denoting a composition of operators and the latter denoting ordinary squaring.

The following additional examples are presented to further clarify the concepts.

```
( <x**3>(<ln>) )_(x)    →   x
In contrast to ( <x**3>(ln) )_(x)    →   ln(x)**3
In contrast to ( <x**3>( <ln(t)> )_(x)    →   ln(ln(ln(x)))

( 1−cos**2 )_(x)    →   1 − cos(x)**2
In contrast to  ( 1 − <cos(t)>**2 )_(x)    →   1 − cos(cos(x))

sin( (1/sin)_(x) )    →   sin(1/sin(x))
In contrast to  sin( ( 1 / <sin(t)> )_(x) )    →   sin(arcsin(x))

( (<D>−a)**2 )_(y)    →   (y − a(y))**2
evalb( <expr|x>(1) = subs(x=1,expr) )    →   true

< diff(f(x),x,x,x) >_(0)    →   < diff(f(x),x,x,x) >_(0)
# (f is undefined, diff remains unevaluated, so the operator remains unevaluated)
f := sin; "";    →   −1
```

The final minutes were spent looking at the notation for ambiguities, particularly with respect to the rest of the Maple expression syntax. <a>b + c<d> can only be a Boolean operator, for example, since <a>b can't be an operator expression, it would have to be

meaning than a constant in an ordinary Maple expression. However, it is obvious from the context in the above example that the occurrences of those symbols are meant as functions, while it is not so obvious that functional notation rules should be applied (if they are different from the ordinary simplification rules) in the situation:

```
b := a + e + 1 + partial(f,1) - a ;
b(7)
```

since when the assignment to b is made, there is only the weak implication that functions are involved by the occurrence of the word "partial".

The next observation is that actually we want (f)(7) to be f(7) when f is a function. Therefore the parentheses notation (expr1) (expr2) means "expr1" is a function, evaluate it at "expr2". Thus if we want something for *operators*, we need something different such as

```
<expr1> (expr2) (expr3)
```

meaning "apply the operator expression expr1 to the functional expression expr2 to get a function as a result. Evaluate that result at expr3".

Should operators be manipulated according to the rules of standard eval/simpl algebra? (At last, we have a name for the domain that Maple does manipulation in! "The algebra of whatever eval/simpl does".) How can one define an operator? A suggestion was that there be an "option operator":

```
f := proc (x)
     option operator;
     e + 1 + partial(x,1)
     end;
```

However, there is no way in maple to do algebra on "proc"s, at least not yet. But putting the "option operator" into procs would be an easy flag for the simplifier to catch if we wanted to build in some automatic simplification to work only on operators.

An observation was made that this was somewhat similar to "procedurizing" the function used in a "map", such as

```
map( <x**2>, [a,b,c] ) → [a**2, b**2, c**2]
```

Currently there is no way to conveniently express "the squaring function" except as the somewhat clumsier

```
map( proc (x) x**2 end, [a,b,c] )
```

A warning was issued that there was a conflict between different operator notations. For example $f^2(x)$ may mean $f(x)*f(x)$, while $D^n f$ means the n-th derivative of f -- n compositions of the operator D. This confusion can be reduced by the observation that f in the first example isn't an operator, but rather the "squaring operator" working on the function f.

A concrete suggestion arose for operator notation (with later refinement to what is presented here):

```
<operator expression> (arguments)
```

with a separate concept of applying expressions to expressions in a "functional" way (see below):

```
(expression) _ (expression)
```

Thus an operator expression is denoted by angle brackets <>. The arguments to the operator expression appear immediately to the right of the angle brackets. The operator notation can be interpreted as meaning "procedurize" as in:

<a>(b).

4. Conclusions

The session adjourned with the feeling that things were just getting rolling; we could have easily spent another day in the perfect weather at Sparrow Lake talking about these issues. All were agreed, however, that the retreat provided a rare opportunity to discuss more complicated issues in a conducive environment free of other commitments.

References

Cha83a. Bruce Char, Keith Geddes, Morven Gentleman, and Gaston Gonnet, *The Design of Maple: A compact, portable, and powerful computer algebra system*, to appear in *Proceedings of Eurocal '83*, London, April 1983.

Ged82a. Keith Geddes, Gaston Gonnet, and Bruce Char, *Maple User's Manual, 2nd edition*, University of Waterloo Research Report CS-82-40, December 1982.

Tur82a. David Turner, *Programming with Infinite Data Structures*, Invited Talk given at 1982 ACM Symposium on Lisp and Functional Programming. 1982.

Wat83a. Stephen M. Watt, *Arrays and Tables in Maple: Supplement to the Maple User's Manual*, University of Waterloo Research Report CS-83-10, May 1983.