SMP: A Symbolic Manipulation Program

Chris A. Cole and Stephen Wolfram
Physics Department, California Institute of Technology, Pasadena CA 91125

May 1981

SMP is a new general-purpose symbolic manipulation computer program which has been developed during the past year by the authors, with help from G.C.Fox, J.M.Greif, E.D.Mjolsness, L.J.Romans, T.Shaw and A.E.Terrano. The primary motivation for the construction of the program was the necessity of performing very complicated algebraic manipulations in certain areas of theoretical physics. The need to deal with advanced mathematical constructs required the program to be of great generality. In addition, the size of the calculations anticipated demanded that the program should operate quickly and be capable of handling very large amounts of data. The resulting program is expected to be valuable in a wide variety of applications.

Of the existing symbolic manipulation programs, SCHOONSCHIP was the only one designed to handle the very large expressions encountered, and MACSYMA the only one of any generality.

In this paper, we describe some of the basic concepts and principles of SMP. The extensive capabilities of SMP are described, with examples, in the "SMP Handbook" (available on request from the authors).

The basic purpose of SMP is to manipulate symbolic expressions. These expressions may represent algebraic formulae, on which mathematical operations are performed. By virtue of their symbolic nature, they may also represent procedures and actions.

The ability to manipulate symbolic expressions, as well as sets of numbers, allows for much greater generality and a much richer architecture than in numerical computer languages.

The structure of expressions in SMP is defined recursively as follows:

expr consists of *symbol*
or *expr*[*expr*, *expr*, ...] (projection)
or { [*expr*] : *expr*, [*expr*] : *expr*, ... } (list)

Symbols are the fundamental units. Projections represent extraction of a part in an expression. Lists allow expressions to be collected together.

To be published in the Proceedings 1981 ACM Symposium on Symbolic & Algebraic Computation.

These three fundamental forms suffice to represent all the objects, operations and procedures required in symbolic manipulations.

Symbols are labelled by a unique name (e.g. `x3` or `Mult`) which is used to represent them for input and output. Expressions may be assigned as values for symbols. A symbol to which a value has been assigned becomes essentially a short notation for its value, and is replaced by the value whenever it occurs. If no value has been assigned to a symbol, the results of (most) operations are such as would hold for any possible value of the symbol. The set of possible values represented by a symbol may be delimited by assigning a list of properties to the symbol.

The projection `f`[*expr*] represents the part of the expression `f` selected by the "filter" *expr*. If `f` is a list, the entry with index *expr* is selected. If `f` is a symbol with no value, operations performed on `f`[*expr*] hold for any value of `f`. Properties assigned to `f` may specify particular treatment. "System-defined" symbols stand for many fundamental operations. Projections from these symbols (e.g. `Plus`) yield expressions which represent the action of these operations on the filter expressions.

In projections such as `f[x1,x2,x3]` or `f[x1][x2]` with several filters, the filters are used successively or together ("curried" or "uncurried") in selection of a part of `f`.

Lists are ordered and indexed sets of expressions. The index and value of each entry in a list may be arbitrary expressions. A particular value in a list is extracted by projection with the corresponding index as a filter.

If the value of some symbol `f` is a list, the entries of the list describe parts of the "object" `f`: they give the values for projections of `f` with different filters. Entries may be introduced into such a list by assignment of values for projections of `f`.

For example, `f:{[1]:a}` (or `f[1]:a`) defines `f` to be an object whose projection with filter 1 is `a`. The values of other projections from `f` remain unspecified. `f[x+y]:(x+y)^2` then yields `{[x+y]:(x+y)^2, [1]:a}` and defines the value for a projection of `f` with filter `x+y` to be `(x+y)^2`.

Lists whose indices are successive integers (starting at 1, and termed "contiguous") are used to represent vectors. They are analogous to "arrays" in numerical languages such as FORTRAN, ALGOL or APL. Lists whose indices are fixed symbols are analogous to C, COBOL or PL/1 "structures" or PASCAL "records".

SMP incorporates many list manipulation facilities. `Ar` generates a list with a specified structure and entries according to a given "template": it alone encompasses many of the list manipulation capabilities of APL. Projections may be defined to be distributed over entries of any lists appearing as their filters, allowing lists to be used to collect a set of expressions on which the same operations may be performed. `Flat` "unravels" sublists within lists. `Sort`, `Cat` (concatenate), `Cyc` (cycle), `Rev` (reverse), `Union` and `Inter` (intersection) are also provided.

An ordinary symbol (such as `x`) is taken to stand for the same expression whenever it appears. A "generic" symbol (such as `$x`) may represent any one of a possibly infinite class ("genus") of expressions. Different occurrences of a generic symbol may stand for different members of the class. "Generic expressions" or "patterns" may stand for any one of a class of expressions in which generic symbols are replaced by suitable expressions.

List entries whose indices and values are patterns define transformations for classes of expressions. Projection of the list with an expression which is a particular case of the index pattern yields a corresponding specialization of the value pattern. The necessary replacements of generic symbols in the index are performed in the value.

For example, `g:{[$x]:$x^2}` (or `g[$x]:$x^2`) defines `g` to be the operation of transforming an arbitrary expression into its square. Thus `g[2]` becomes `4` while `g[x+y]` becomes `(x+y)^2`. This is to be contrasted with the assignment `f[x+y]:(x+y)^2` given above, which defined a value only for projection with the specific filter `x+y`.

Lists with entries of the form `{[$x]:`*expr*`}`, where *expr* is some expression containing the generic symbol `$x`, correspond to "lambda functions" in LISP, with `$x` the "bound variable". Assignments such as `f[$x]:$x-2` parallel "function definitions" familiar from FORTRAN, C and so on.

When several occurrences of the same generic symbol appear explicitly in a particular pattern, they must correspond to the same expression. Generic symbols in different patterns may represent different expressions.

List entries whose indices are arbitrary patterns define transformations for expressions with particular structures. For example, the assignment `f[$x,1-$x]:h[$x]` defines `f[a,1-a]` to become `h[a]` and `f[5,-4]` to become `h[5]`.

Two patterns are "literally equivalent" if all of their parts are identical, possibly after properties of projections such as commutativity have been accounted for. A pattern *expr2* "matches" *expr1* if the simplified form of *expr2* after suitable replacements for generic symbols is literally equivalent to *expr1*. Replacements for generic symbols must be deducible by literal comparison from at least one of the occurrences of each generic symbol in *expr2*. Thus `f[$x+$y,$x,a+$y]` is determined to match `f[5,2,a+3]`, but `f[$x+$y,$x-$y]` is not determined to match `f[5,-1]`.

If assignments are made for several patterns with overlapping domains of applicability, the assignments for more specific patterns are used in preference to those for more general cases. Hence, with `g[0]:a; g[$x]:1/$x` the value of `g` becomes `{[0]:a, [$x]:1/$x}` so that projections of `g` with the specific filter `0` are `a` but projections with other filters are the reciprocals of those filters.

A Boolean condition may be associated with a pattern to restrict expressions which it matches. *pat*`_=`*cond* represents a pattern equivalent to *pat*, but constrained to match only expressions for which *cond* is determined to be "true" after necessary replacements for generic symbols. Hence, for example, `f[$x_=(5<$x<7)]:$x^2` defines values for projections of `f` whose filters lie between 5 and 7; thus `f[6]` becomes 36 but `f[1]` or `f[x]` remain unevaluated.

The arbitrary structure of patterns used as indices in lists allows definition of "functions" whose "arguments" are constrained to be of particular "types".

SMP incorporates standard logical and relational operations such as `~` (`Not`), `&` (`And`), `~=` (`Uneq` unequal), together with character determination projections such as `Natp` (natural number) and `Polyp` (polynomial). "False" and "true" are identified with `0` and non-zero numbers respectively.

The values of entries in a list may themselves be lists. The resulting form may be pictured as an "n-ary" or "multiway" tree. Each list is a node on the tree, with branches leading to the list entries and labeled by the list indices. A particular part of the tree is selected by a projection with a succession of filters specifying the branch to be taken at each node encountered in descent from the root of the tree.

Contiguous lists of lists (with successive integer indices) represent matrices and tensors. Lists of lists with fixed indices are analogous to hierarchical data bases. Lists of lists with patterns as indices represent "functions" with several parameters.

"Multi-generic" symbols (such as `$$x`) represents sequences ("null projections") of expressions. Thus for example, `f[$$x]` stands for projections of `f` with arbitrary sets of filters; in `f[a,b,c]` `$$x` represents `[a,b,c]`. The assignment `Log[$x $$x]:Log[$x]+Log[$$x]` (space indicates multiplication) defines a logarithms to be expanded, so that `Log[a b (x+y)]` becomes `Log[a]+Log[b]+Log[x+y]`.

All SMP expressions have the structure of n-ary trees. Projections are nodes whose branches (labeled by successive integer indices) lead to the filters of the projection. Symbols form the ultimate terminals ("leaves") of the tree. Parts of an expression may be selected by projections with suitable filters, and may be modified, added or removed by assignments or deassignments for these projections. With `t:f[a^2,b]` the value of the projection `t[1,1]` is `a` and the assignment `t[1,1]:x^2` causes `t` to become `f[x^4,b]`.

Expressions input to SMP are evaluated by replacing each of their parts (starting with the smallest) by any values assigned to them. This process is carried out to the maximum extent possible: unless further assignments are made, the resulting output expressions can be evaluated no further; if input again, they will be output unchanged.

Picturing an expression as an n-ary tree, the terminals (symbols) are evaluated first, followed by the successively larger parts encountered on ascending towards the root of the tree. In evaluating projections, any system-defined procedures are invoked first. Projections with properties such as associativity or antisymmetry are cast into a canonical form, in which mathematically equal expressions are rendered syntactically equivalent. Finally, values assigned to the objects ("projectors") of the projections are scanned. Any value assigned for the required part (projection) is used. Assigning `a:3` the expression `a^2` is first evaluated to `3^2` and then simplified by the system-defined procedure associated with `Pow` to `9`. The expression `0^0` would be left unchanged by this system-defined procedure. Its value may be specified by an assignment such as `0^0:1`.

The SMP procedure for evaluation is arranged so that maximal simplifications and cancellations occur at all stages. In this way, the complexity of intermediate expressions used in generating simple final results is kept to a minimum.

Values assigned for a projection may involve further projections from the same object ("projector"), thereby representing recursive or self-referential function definitions. Evaluation of such values is performed in a sequence of passes through the complete expressions involved; in each pass the recursion is carried only one step further. Thus with the definition `g[$x]:$x g[$x-1]` the expression `g[2]` is evaluated first to `1g[0]` then simplified to `g[0]` and then evaluated to `0g[-1]` and at this stage immediately simplified to `0`. In conventional recursive evaluation schemes, `g[-1]` would be evaluated before the product `0g[-1]` was established to be zero, and the evaluation of `g[2]` would not terminate. The direct recursive definition `f[0]:f[1]:1; f[$x]:f[$x-1]+f[$x-2]` of the Fibonacci series provides a further example. The simplest recursive evaluation of `f[`*n*`]` forms a binary tree requiring exponential time and memory space; in SMP, the time and space required grow only quadratically with *n*.

There are two possible kinds of assignment of a value *expr2* to an expression *expr1*. In "immediate assignment", *expr1*`:`*expr2* specifies the value of *expr1* to be the present value of *expr2*. The resulting value is maintained in an evaluated form, and updated by any subsequent assignments when used. In "delayed assignment", *expr1*`::`*expr2* specifies that whenever the value of *expr1* is requested, the value of *expr2* found at that time is to be given. In this case, the unevaluated form of *expr2* is maintained as the value of *expr1*, and is evaluated afresh when it is used.

With the assignments `b:c a:b` the value of `a` becomes `c`. A subsequent reassignment `b:d` leaves the value of `a` unchanged. On the other hand, with a delayed assignment `a::b` the value of `a` is always the value of `b` at the time of the request. Thus, with `b:c; a::b` a request for the value of `a` would give `c` : after the reassignment `b:d` a request for `a` would however give `d`.

Expressions assigned as "delayed values" for patterns are evaluated after replacement of the necessary generic symbols.

Delayed values may represent "procedures" to be re-executed whenever "called". (A feature which exemplifies the necessity for delayed assignment is that conditionals may depend on symbolic expressions of undetermined truth value.)

Projections may have the property that some or all of their filters are to be maintained in an unsimplified form. In this way, a "procedure" assigned as the value of a symbol may be "passed by name".

SMP incorporates constructs necessary for programming: local variables, `If`, `For`, `Do`, `Rpt` (repeat) and `Swtch` (switch), together with local and non-local returns and jumps.

Assignments define values for expressions to be used whenever the expressions appear. "Replacements" such as `x->y+1` are syntactic constructs which may be applied selectively in a particular expression by an `S` (substitution) projection. Replacements may involve patterns. Note that in substitutions, as in assignments, associativity and commutativity (or other filter reordering symmetries) of projections are accounted for, so that `S[a b c^2 d, $x^2 a -> 1-$x]` yields `(1-c) b d`.

In addition to standard arithmetic operations (such as `+` (`Plus`), `^` (`Pow`), `.` (`Dot` inner product) and `**` (`Omult` outer product)) and elementary functions (such as `Log`) SMP treats a large number of the special functions of mathematical physics (such as `Chg` (confluent hypergeometric function) and `Geg` (Gegenbauer function)). Numerical values of expressions involving such functions are obtained by `N[`*expr*`]`. Simplifications and transformations are not made automatically unless the results are very simple. The necessary formulae are contained in an extensive library of "external files", and given as replacements, to be applied selectively when required. Thus, for example, `S[`*expr*`,XTrig[2,6]]` applies the half-angle relations for trigonometric functions in *expr*. The name `XTrig[2,6]` of the set of relations to apply is found from the list of formulae in the "SMP Reference Manual".

SMP incorporates a variety of facilities for effecting structural simplifications and modifications on expressions. Parts in expressions may be reassigned to have modified values. The parts may be identified using the projection `Pos` which yields a list of positions of a particular pattern in an expression. The projection `Map` may be used to apply a "template" to a "domain" or set of parts in an expression. A template is an expression used to specify an action on a set of expressions. Application of the template `f` to for example `x` and `y` yields `f[x,y]`. The template `$x^($y+$x)` has two "slots" indicated by `$x` and `$y` into which expressions are inserted: application to `x` and `y` yields `x^(x+y)`. Other facilities include `Cb[`*expr*`,`*form*`]`, which combines coefficients of terms matching the pattern *form* in *expr*.

SMP incorporates projections for performing expansions using distributivity. It also contains facilities for factorization of polynomials, for forming partial fractions, and for other polynomial manipulations.

SMP performs derivatives, sums, products, integrals, series expansions and solves equations. Assignments may be made to define derivatives, integrals, inverses and so on, for new mathematical functions. For example, `D[f[$x],{$x,1,$y}]:g[$y]` defines the first derivative of the "function" `f` of one "argument" to be `g`.

Input and output in SMP conform as closely as possible to standard mathematical notation. Input syntax may be modified and new forms introduced. Arbitrary output forms may be defined. Numerical values of expressions may be plotted.