Tentative Design of a New Symbolic ~~Algebraic~~ Manipulation Program ~~Language~~

STEPHEN WOLFRAM

ABSTRACT

These notes discuss tentative plans for the construction of a new algebraic manipulation computer language, based on the (UNIX) language C. The new language is intended to be both fast and capable of processing very large intermediate expressions (whose size is limited only by available computer memory), as appear in many physical problems.

Mention reverse Polish

1. Existing Languages and Introduction

A very large number of computer languages able to perform some manner of symbolic and algebraic manipulation has been constructed over the last 20, or so years (the first one of which I am aware was written by J. Mathews at Caltech in 1959 to perform tensor algebra encountered in calculating Feynman diagrams for the graviton loop corrections to the photon propagator). Some 'languages' were designed for a specific physical problem, and were abandoned when the problem was complete. Others were written primarily to research algorithms and the structure of languages, and were never applied to any unsolved scientific problem. Still others were written in languages (often machine codes) which have now become obsolete. The main languages which are in reasonably widespread use today are summarized in Table 1. Their strengths and weaknesses are all very different. The earliest of the languages were SCHOONSHIP, ASHMEDAI and REDUCE, which were designed primarily to perform traces of Dirac gamma matrices as are required in the evaluation of Feynman diagrams. Of the three only REDUCE has been used extensively outside particle physics, but for the most complicated Feynman diagram calculations, SCHOONSHIP is almost exclusively used. REDUCE, SCHOONSHIP and ASHMEDAI may all be classified as 'low-level' algebraic manipulators: their primary function is to substitute polynomials into polynomials, and perform simple expansions or reductions on the result. However, at least SCHOONSHIP, and to some extent ASHMEDAI, are capable of manipulating and simplifying huge polynomials, containing perhaps a million terms, in a realistic amount (~ minutes) of computer time. Despite its elegance, it appears that the basic structure of LISP (discussed below) is largely inappropriate and inadequate for performing such 'heavy duty' algebra. Both REDUCE and MACSYMA are therefore limited to approaches and problems in which intermediate expressions remain tolerably small (typically taking $\leqslant$ 1 megabyte of computer memory, corresponding to polynomials with a few thousand terms). This is a very severe restriction on the usefulness of LISP-based algebraic manipulators. Probably the main reason that REDUCE has enjoyed such widespread popularity is its portability and comparative simplicity: it cannot compete with, for example, SCHOONSHIP on the size of problem accessible. Despite the fundamental size restrictions, MACSYMA has the important merit of having many complicated mathematical operations (such as polynomial factoring and symbolic integration) built-in. MACSYMA was originally constructed around J. Moses' symbolic integration routine SIN, which was largely intended as a project in artificial intelligence. Over the course of 10 or so years, MACSYMA has grown considerably in sophistication and complexity, and now has some 500 built-in commands, although many of its mathematical commands have never been tested on problems much larger than those easily accessible to manual solution. One might hope that the sophistications of MACSYMA could overcome the memory space and computer time problems that it inevitably inherits from LISP; in some important cases, this is possible, but mostly the sophistications make matters worse.

It should be noted that on many practical problems, SCHOONSHIP runs between a factor of 10 to 100 (after taking out difference between intrinsic computer instruction times) faster than REDUCE or MACSYMA. Thus improvements on the basic structure of, for example, MACSYMA can clearly be very profitable. Some of MACSYMA's mathematical algorithms are, nevertheless, the result of much investigation, and are very efficient. The main examples of such algorithms are those for forming partial fractions, for factoring univariate polynomials, and for indefinite symbolic integration of elementary functions (Moses' hierarchy of algorithms and Risch's algorithm). In constructing a new language, the same algorithms would probably be used for these purposes.

The purpose of these notes is to discuss some tentative plans for the design of a new algebraic manipulation language, intended to be capable of manipulating very large expressions and performing complicated mathematical operations. Its design is intended to be simple, so as to aid debugging and extensions, to enhance portability and to permit reasonably fast construction. I shall describe in some detail the structure of the intended language; almost nothing of it has so far been implemented, and in many cases the precise method of implementation has not yet been decided.

Analytical results for physical problems are usually useful only if they are reasonably simple. In other cases, only numerical results are profitable, and can usually be derived most easily by purely numerical techniques. (There are, however, some areas in which it is useful to obtain complicated intermediate analytical results so as to ease later numerical analysis.) Despite the simplicity of many final analytical results, intermediate expressions generated in the course of obtaining them may be incredibly large and complicated. The evaluation of complex Feynman diagrams in particle physics is one of the most spectacular examples of such a case. The final analytical forms for most interesting higher-order diagrams depend on only a few parameters, and have in final answer typically less than ten terms (and hence may be written on one line). On the other hand, at intermediate stages in the calculation, literally millions of terms may be generated, nearly all of which eventually cancel out to leave the final simple result. This phenomenon may well be a signal that present methods for Feynman diagram calculation are wasteful and unnecessarily complicated: such a possibility is one of the major motivations for obtaining analytical results, since patterns in these could help to reveal better methods.

In (most) present implementations of REDUCE and MACSYMA, all parts of an expression being processed must be stored in memory locations which can be addressed using 18 bits; the maximum active region of memory is therefore usually under 1 megabyte. This technical restriction can in principle be lifted, but all indications are that the CPU time to process much larger expressions (as necessary for Feynman diagram calculations) would be totally unrealistic. The basic reason for this inadequacy probably lies with the way in which expressions are naturally stored by LISP, and attendant memory management problems. The method of storage is based on binary trees. For example, the expression

$$x^2 + 3\ x\ y + y^2 + 1/4 \qquad (1)$$

is typically stored as in Fig. 1. Each node represents one word of computer memory. The first half of the word contains a 'pointer' which consists of the absolute address of the next node on the left-hand branch down the tree, and the second half for the right hand branch (if the branches lead to symbols, then at the node is a pointer to a region of memory which describes the properties of the symbols). Figure 1 is clearly not a very natural way to represent (1). A much simpler and more direct representation is afforded by use of an n-any tree, as illustrated in Fig. 2. (This representation is similar to that used by SCHOONSHIP.) Here the nodes may consist of many contiguous words of memory: an operator (e.g., +) is stored at each node, together with pointers to each of its arguments. One might expect that the management of such semicontiguously-used memory was more difficult than for entirely scattered LISP-like usage. However, it will turn out that by arranging to signal immediately when a region of memory can be overwritten, such a storage scheme may easily be controlled.

A fundamental and oft-advertised feature of LISP is its widespread use of recursive functions. For example, the factorial function might naturally be defined in a LISP-based language by

```
FAC(x): = if x = 1 then 1 else x*FAC(x-1).                    (2)
```

In a nonrecursive language such as FORTRAN, factorial would be defined iteratively by

```
FAC(x): = (f:1, for i:1 thru x do f:f*i,f).                   (3)
```

Such an iterative definition is unnatural in LISP, and can only be given rather indirectly in it. The definition (2) may well be the more elegant, but it is almost inevitable that it is both slower and more wasteful of memory then (3). Every time the function FAC is entered in (2), the value of the program counter (sequence control register), which gives the return address, and perhaps several other registers, must be saved on a stack. There are rather few cases in doing practical algebra where recursion is necessary or desirable; that recursion should be the functional basis for a language intended primarily to perform algebra seems unnecessarily restrictive. The new language to be described below will include recursion only in a somewhat secondary manner.

Name	Base Language	Author
REDUCE	LISP	A. Hearn (Stanford/Utah)
MACSYMA	LISP	J. Moses, etc. (MIT)
SCHOONSHIP	COMPASS*	M. Veltman (CERN/Utrecht)
ASHMEDAI	FORTRAN	M. Levine (Caltech/Carnegie-Mellon)
(CAMAL		A. Norman (Cambridge U.))

Table 1: Major existing general algebraic manipulation languages in approximate order of their usage.

* CDC 6000 series assembly language.

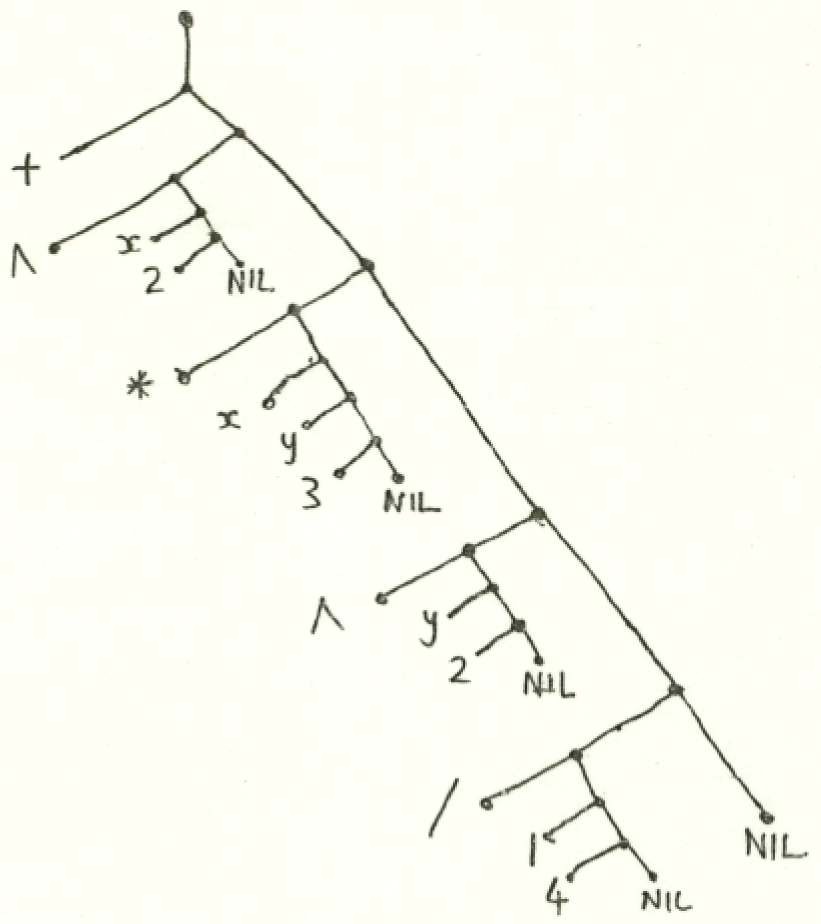

Fig. 1: The binary tree which represents the expression (1) in LISP.

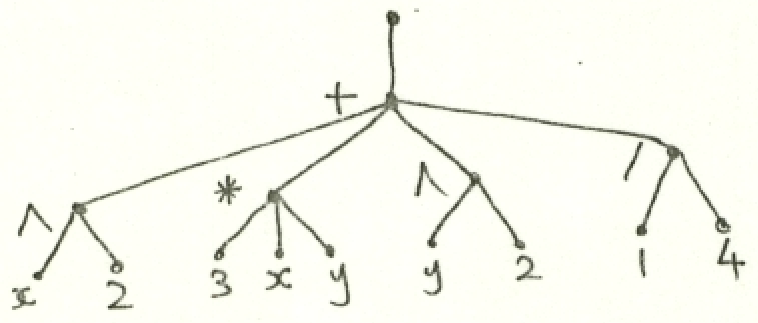

Fig. 2: An n-ary tree representation of the expression (1) similar to that intended for the new language.