JOURNAL OF COMPUTATIONAL PHYSICS 4, 171–194 (1969)

Computer Algorithms for Dirac Algebra[1,2]

STANLEY M. SWANSON

Institute of Theoretical Physics,
Department of Physics, Stanford University, Stanford, California 94305

Received September 30, 1968

ABSTRACT

We present a collection of algorithms written in ALGOL 60 to illustrate the manipulation techniques used in machine language programs designed for symbolic evaluation of algebraic expressions. Such algorithms have been used by theoretical physicists in quantum electrodynamics calculations. A subset of these methods is useful in the reduction of complicated perturbation expansions which involve only ordinary (commutative) algebra. It is hoped that this will provide a more understandable and accessible archive for these methods than scattered listings of assembly code for specific computers. We consider the input and distribution of arbitrarily parenthesized algebraic expressions, Chisholm's reduction of the product of traces with intercontracted indices to single traces, the removal of contracted indices in single traces to produce simple traces which are then reduced to invariant dot products (and determinants if γ_5's are present). Several storage schemes are discussed.

INTRODUCTION

The calculation of traces arising from perturbation theory expansions in quantum electrodynamics involves repetitive applications of a few simple rules. In complicated calculations, such tedium is best relegated to a digital computer which is less apt to make manipulative mistakes than a theoretical physicist. Likewise, a theoretician's desire to expand an algebraic expression to the first few orders in a small quantity can require a prohibitive amount of bookkeeping. There is considerable prior art in this field [1–8] and there will undoubtedly be subsequent effort due to vagaries in the dissemination of techniques, the differences between computers on which such programs are implemented, and the greater capabilities of more modern machines.

---

[1] Research sponsored in part by the Air Force Office of Scientific Research, Office of Aerospace Research, U.S. Air Force, under Contract F 44620-68-C-0075.

[2] This work is part of the author's Ph.D. Thesis, Stanford University, May, 1968.

There are two extremes in the implementation of algebraic manipulations on computers: machine language programs written to do specific tasks at maximum efficiency and maximum modification effort, and programs written in higher level languages more or less suited to list manipulation. The latter type is easy to modify, but usually sacrifices some efficiency. The machine language techniques, with which we shall be concerned, take advantage of the idiosyncracies of specific computers, but generally hide their basic algorithms in a mass of detail. I shall try to illustrate the methods in the ALGOL 60 language [9], without optimizing the program for the Burroughs B 5500 on which it was tested, so that the ideas of internal representation, sorting on identifying bit patterns, and various permutation schemes are more accessible than they would be through reading assembly code listings. In this type of programming, one defines the meaning of bit patterns within words and of groups of words through the operations performed upon them, in contrast to the case of numerical computation where registers and memory locations can be treated logically as numbers, relying on hardware to interpret the bit patterns within them.

Although the input (and occasionally the output) of algebra programs is fairly innocuous, the intermediate manipulations generate a morass of terms which combine and cancel in various ways. The efficient utilization of rapid access storage for these intermediate manipulations will be less of a problem for the current generation of computers (with ca. 10^7 bits of immediate storage) than it was on

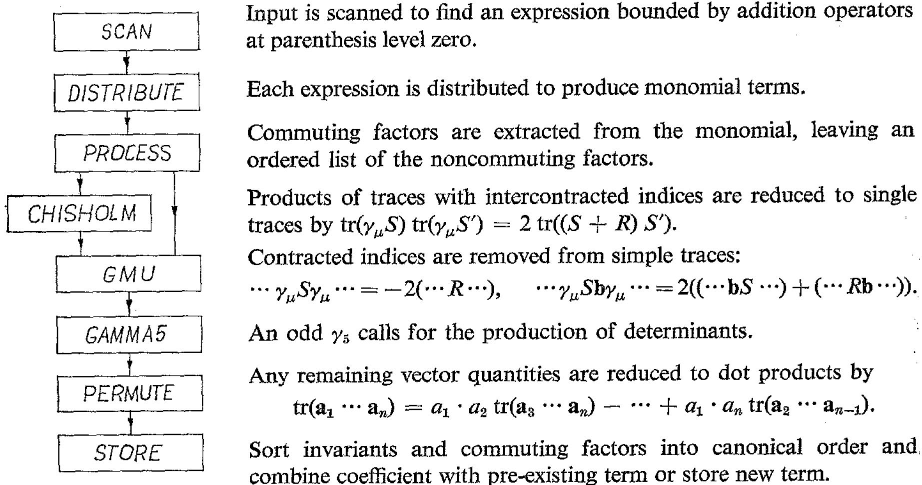

FIG. 1. Synopsis of the trace reduction scheme. Names in the boxes are of procedures which will be discussed subsequently. The symbol S or S' denotes the product of an odd number of gamma matrices: $S = \mathbf{a}_1\mathbf{a}_2 \cdots \mathbf{a}_n$, where n is odd; and R denotes a product with the same factors as S, but written in reversed order: $R = \mathbf{a}_n \cdots \mathbf{a}_2\mathbf{a}_1$.

computers such as the IBM 7090 (with only 10^6 bits available). Concern will shift to the major problem of reducing and comprehending the complicated expressions generated as output from calculations. The ALGOL program to be discussed here has adopted the philosophy of its predecessor FTRACE, an assembly language program for the IBM 7090: to reduce pieces of an expression to the final output invariants before wholesale storage is attempted. This is accomplished by nested procedure calls corresponding to a hierarchy of reduction steps in Dirac trace algebra. An overview of the main procedures and of this reduction scheme is given in Figure 1.

We will now focus on the details of the algorithms, considering in turn the input and distribution of algebraic expressions to produce monomial terms, the reduction of traces with contracted indices to simple traces, the production of invariants (dot products and determinants) from simple traces, and the storage and collection of the output expression.

Input and Distribution of Expressions

Input and output depend ultimately on a hardware representation of characters and rely on machine language coding tied to a specific correspondence between print symbols and bit sequences. Moreover, sophisticated algebra programs generally recognize control words which distinguish between various types of tasks, such as evaluation of expressions, performing substitutions, and determining sorting priorities. The recognition of multiple character symbols, task assignments, and elaborate error detection and correction are problems which will be left to the reader. We will treat the input of algebraic expressions in a somewhat schematic manner and then move on to their evaluation, introducing techniques which can be applied to other problems, such as substitutions.

My presentation of the algorithms will deviate slightly from the ALGOL 60 reference language [9]. Logical operators will be spelled out (**not**, **and**, **or**). We introduce an additional integer arithmetic operator "**mod**" which gives the remainder in division (A **mod** $B = A - B \times (A \div B)$). Finally, short strings will stand for integer values derived from the value assigned by the input routine *CHAR* to single input characters. *CHAR* is defined in terms of the standard [10] input routine *insymbol*:

```
integer procedure CHAR;
begin integer Z;
A: insymbol
      (1,‘×)+−(.0123456789␣#ABCDEFGHIJKLMNOPQRSTUVWXYZ;’,Z);
   if Z = 0 or Z = ‘␣’ then go to A; CHAR := abs(Z);
end CHAR;
```

Here *insymbol* interrogates external channel 1 for the next input character and assigns to Z an integer value corresponding to the position of the input character in the string '$\times) + \cdots Z;$' or sets Z to zero if the character does not appear in the string. (The order of the symbols in this string has been chosen to aid in the syntactical analysis of the input by the procedure *SCAN*, to be discussed next.) Thus *CHAR* ignores spaces (␣) and characters not appearing in the reference string; it gives the value 1 for a multiplication sign, 5 for a left parenthesis, 19 for the letter '*A*', and so on. In the ALGOL, single character proper strings assume values corresponding to these assignments (e.g., the expression $3 \times$ ')' + '2' equals 15) and n-character proper strings form n-digit numbers of radix 64 (e.g., '*ABC*' = $64 \times (64 \times$ '*A*' + '*B*') + '*C*' = 79125).

Another matter to be dealt with is the definition of nonlocal variables. A number of arrays and simple variables used in common by several procedures have been defined in the outside program block, nonlocally to the respective procedure blocks. This was partly a matter of efficiency and necessity under the ALGOL compiler on the B 5500 [11]. In a more involved program which uses procedures like *DISTRIBUTE* in several contexts, it might be desirable to treat these nonlocal variables and called procedures as formal parameters. The program block head starts

begin comment program block;
  integer array *V*,*VL*,*OPL*[0:100], *INDEX*,*CNUM*[0:30], *LIM*[0:1,0:10];
  array *QNUM*[0:30]; **real array** *CO*[0:100]; **Boolean array** *MULT*[0:100];

The upperbounds on these arrays set size limitations on expressions the program can handle. Since provision for real coefficients was something of an afterthought, a warning about the types of some of the arrays is in order. The convenience of real coefficients comes in the algebraic evaluation of series and in substitution for invariants, rather than in taking traces where the numerical operations are limited to change of sign, multiplication by small integers and combining terms. To facilitate transfer of information to scratch arrays, both a coefficient (possibly real) in *QNUM* [0] and symbolic information about noncommuting invariants (which is intrinsically integer or logical) in *QNUM* [1 : n] has been carried in the same array. Similarly both numeric and symbolic information is carried in the mass storage array *TERM* (to be declared later); such mixed mode arrays will be declared without specifying their type. A compiler's propensity to transfer types in arithmetic and assignment operations, which might lead to truncation errors, would be circumvented in machine language coding. To stay within the confines of a compiled program, such arrays should be replaced by variables or arrays of the two appropriate types and additional assignment statements added.

Algebraic expressions acceptable as input to this program are similar in generality and form to those familiar in ALGOL or FORTRAN. Arbitrary nesting of paren-

theses is permissible, but division and exponentiation are absent. Provision for integer exponents would be a desirable and minor modification. Implicit multiplication is permitted, allowing one to omit the multiplication sign in contexts where such omission is not ambiguous. Symbolic variables have been extended to include the noncommuting quantities of Dirac matrix algebra and a trace operator to separate intercontracted traces. In this version, only a single character identifies a symbolic variable (ignoring for the moment, the prefix '#' on noncommuting variables); efficient extension to multiple character identifiers enmeshes one in the idiosyncrasies of specific computers and would modify the meaning of implicit multiplication in some expressions.

Input syntax will be specified from two approaches: first by a set of recursive definitions similar to those used in the ALGOL report [9], and then by a "transition" matrix between syntactical entities which will lead naturally to the input routine *SCAN*. Starting with ⟨letter⟩ (uppercase) and ⟨decimal number⟩ (a sequence of digits, possibly containing one decimal point) as primitives, we parallel the ALGOL definition of an (arithmetic) ⟨expression⟩ in the following definitions:

⟨adding operator⟩ : := + | −
⟨multiplying operator⟩ : := × | ⟨empty (except the case ⟨number⟩ × ⟨number⟩)⟩
⟨commuting variable⟩ : := ⟨letter⟩
⟨vector · γ⟩ : := #⟨letter⟩
⟨γ_μ(to be contracted)⟩ : := #⟨digit $\neq$ 5⟩
⟨γ_5⟩ : := #5
⟨trace operator⟩ : := ##
⟨variable⟩ : := ⟨commuting variable⟩ | ⟨vector · γ⟩ | ⟨γ_μ⟩ | ⟨γ_5⟩ | ⟨trace operator⟩
⟨primary⟩ : := ⟨variable⟩ | ⟨decimal number⟩ | (⟨expression⟩)
⟨term⟩ : := ⟨primary⟩ | ⟨term⟩⟨multiplying operator⟩⟨primary⟩
⟨expression⟩ : := ⟨term⟩ | ⟨adding operator⟩⟨term⟩ | ⟨expression⟩
⟨adding operator⟩⟨term⟩

Not all valid expressions make sense as traces, since some care must be exercised in the use of ⟨γ_μ⟩ and ⟨trace operator⟩; restrictions on their use will be discussed in the section on trace reduction. We have arbitrarily decreed that certain digits following '#' represent indices on gamma matrices which will be subsequently contracted; a more elaborate input routine could allow the definition of certain identifiers to stand for such indices. One could also assign integers to represent various external print names of identifiers, rather than simply using the integer assigned to a symbol by *CHAR*; the usefulness of this refinement will become apparent later when sorting and storage schemes are discussed.

The structure implicit in the above definitions can be summarized by a matrix showing which syntactical entities may follow others: the symbols '×', ')', '+', '−', and '(' stand for themselves, while the letters are abbreviations: N = decimal number; V = variable; A = allowed juxta-position; E = error; I = implicit multiplication, '×' is supplied; S = an initial sign on an expression.

preceding entity \ following entity	×	)	+	−	(	N	V
×	E	E	E	E	A	A	A
)	A	A	A	A	I	I	I
+	E	E	E	E	A	A	A
−	E	E	E	E	A	A	A
(	E	E	S	S	A	A	A
N	A	A	A	A	I	E	I
V	A	A	A	A	I	I	I

The logic of the input procedure *SCAN* closely follows this matrix, as it translates an expression into an alternating sequence of operands (variables or numbers) and operators (adding or multiplying) with their associated parenthesis level. The parenthesis level of a variable or operator is defined to be the number of left parentheses minus the number of right parentheses which have preceded it in the expression. An expression breaks naturally into subexpressions separated by adding operators at parenthesis level zero; when one such subexpression has been translated, it is distributed and reduced to invariants before proceeding to the next subexpression. Scanning is terminated by a ';' at the end of an expression or by an error.

```
procedure SCAN;
begin integer LVL, N, S, T, U, VAR; real C, D, R;
  procedure TIMES;
  begin MULT[N] := true; OPL[N] := LVL; N := N+1; end TIMES;
  procedure NUMBER; begin C := 0; D := 1;
  W: if '0' ≤ U and U ≤ '9' then
      begin C := 10×C+U−7; D := 10×D; U := CHAR; go to W end;
      end NUMBER;
LVL := N := T := 0; U := CHAR;
comment break syntax into entities: ×)+−( number variable;
A: S := T; T := U;
if '.' ≤ T and T ≤ '9' then begin NUMBER; R := C; VAR := 0;
  if U = '.' then U := CHAR; NUMBER; R := R+C/D;
  if S = '−' then R := −R; if U = '.' then go to ERR end else
```

```
if T = ';' then begin T := '+'; LVL := 0 end
else begin if '#' ⩽ T then if T = '#' then VAR := −CHAR else VAR := T end;
        U := CHAR end;
comment analyze syntax;
if T = '×' then begin if U ⩾ '(' then TIMES else go to ERR end else–
if T = ')' then begin LVL := LVL−1; if LVL < 0 then go to ERR;
                  if U ⩾ '(' then TIMES end else
if T ⩽ '−' then begin if U ⩾ '(' then begin
  if S ≠ '(' and S ≠ 0 then begin MULT[N] := false; OPL[N] := LVL;
      if LVL = 0 then begin DISTRIBUTE(N); N := 0 end else N := N+1 end;
    if T = '−' and U > '9' then
      begin V[N] := 0; VL[N] := LVL; CO[N] := −1; TIMES end
  end else go to ERR end else
if T = '(' then begin if U ⩾ '+' then LVL := LVL+1 else go to ERR end else
if T ⩾ '.' then begin if T ⩽ '9' then CO[N] := R;
  V[N] := VAR; VL[N] := LVL; if U ⩾ '(' then TIMES end;
if U = ';' then go to B else go to A;
ERR: outstring (2, 'SYNTAX ERROR');
B: end SCAN;
```

There are two anomalies in *SCAN*: a solitary decimal point will be treated as the number zero and any symbol may appear after '#'. Note that there are essentially only two operators ('×' and '+') since a minus sign is absorbed into a following number or treated as the sequence '−1×'.

The distribution algorithm was originally devised by Kaiser [3] and reconstructed by Levine [5, 12]. *DISTRIBUTE* uses the parenthesis level information in *VL* and *OPL* and the identification of operators as multiplicative or additive in *MULT* to produce a list in *INDEX* of the operands in *V* which correspond to a single monomial. Distribution is from left to right (initial factors in monomials change most frequently) rather than the more customary right to left. Loosely speaking, the array *USED* marks the terms within an expression which have been used in a previous monomial; thus each factor in the current monomial is the first unused operand within a set of operands joined by additive operators. At the beginning of the accumulation of factors for a monomial (*MORE* false), the first selected operand followed by an additive operator is marked used and the marks on the operands to the left may be changed to indicate the factors of the next monomial; after this (*MORE* now true), the index I is merely stepped over operands in an additive relation to the current factor. It is difficult to be verbally precise because "term" and "expression" are defined recursively; one must understand the ALGOL:

```
procedure DISTRIBUTE (N); value N; integer N;
begin integer I, J, K, L, LEVEL;
  Boolean MORE, PRODUCT, TERM; Boolean array USED[0:N];
  for K := 0 step 1 until N do USED[K] := false;
NEXT: MORE := false; J := I := -1;
FACTOR: I := I+1; if USED[I] then go to FACTOR;
  J := J+1; INDEX[J] := I;
SKIP: if MULT[I] then go to FACTOR; LEVEL := OPL[I];
  if LEVEL > 0 then begin
    if MORE then begin
      R: I := I+1; if LEVEL ⩽ OPL[I] then go to R end
    else begin L := LEVEL; LEVEL := VL[I] +1;
      USED[I] := PRODUCT := TERM := MORE := true;
      for K := I-1 step -1 until 0 do begin
        if OPL[K] < LEVEL then begin
          LEVEL := OPL[K]; PRODUCT := MULT[K];
          if LEVEL ⩽ L then TERM := false end;
      if PRODUCT then USED[K] := TERM end end;
  go to SKIP end;
PROCESS(J); if MORE then go to NEXT; end DISTRIBUTE;
```

Each monomial is separated into a numerical coefficient and sequences of commuting and noncommuting factors by *PROCESS*. We first give a simpler procedure *SERIES*, which when substituted for *PROCESS* will evaluate the coefficients of a power series in a single variable up to the power *MAX* (nonlocal). The real array $A[0 : MAX]$ must be defined nonlocally and zeroed before evaluating a series.

```
procedure SERIES (N); integer N;
begin integer I, J, Z; real R; Z := 0; R := 1.0;
  for J := 0 step 1 until N do begin I := INDEX[J];
    if V[I] = 0 then R := R×CO[I] else Z := Z+1 end;
  if Z ⩽ MAX then A[Z] := A[Z]+R; end SERIES;
```

The job of *PROCESS* is more complicated because nonnumeric operands of various types must be distinguished and accumulated. In these illustrative algorithms, the nature of the trace remaining is checked at each level of the nested sequence of procedure calls; it would be more efficient to determine the starting level of trace expansion and the presence of γ_5's in *PROCESS*. If extension to complex arithmetic is desired, this is also the place to recognize and count the occurrences of some reserved symbol (perhaps '*I*') for $\sqrt{-1}$. Additional provision must be made here and in *CHISHOLM* if $(\gamma_5)^2 \neq 1$. The coefficient *R* is initialized

to 4 because we assume that every term involves a trace, if only that of the 4×4 unit matrix.

```
procedure PROCESS(N); value N; integer N;
begin integer I, J, K, KO, M, VAR, C; real R; Boolean G5, ODD, EVEN;
  R := 4; KO := K := J := C := 0; CNUM[30] := 0; M := −1;
  G5 := ODD := false; EVEN := true;
for M := M+1 while EVEN and M ⩽ N do begin
  I := INDEX[M]; VAR := V[I];
  if VAR = 0 then R := R×CO[I] else
  if VAR > 0 then
    begin J := J+1; CNUM[J] := 64×64×(64×'␣'+VAR)+'␣␣' end
  else begin VAR := −VAR;
    if VAR = '5' then G5 := not G5 else
    if VAR ≠ '#' then begin K := K+1;
      if G5 then ODD := not ODD;
      if '0' ⩽ VAR and VAR ⩽ '9' then
        QNUM[K] := −VAR else QNUM[K] := VAR;
    end end;
  if VAR = '#' or M = N then begin
    if ODD then R := −R; ODD := false;
    if G5 then begin K := K+1; QNUM[K] := 5; KO := KO+1 end;
    EVEN := (K−KO) mod 2 = 0 and (not G5 or (K−KO) ⩾ 4);
    G5 := false;
    if M ≠ N then begin KO := K; LIM[1,C] := K; C := C+1;
                                LIM[0,C] := K+1 end;
end end;
if EVEN and R ≠ 0 then begin QNUM[0] := R; LIM[1,C] := K; CNUM[0] := J;
  if C < 2 then GMU(K) else CHISHOLM(K,C); end; end PROCESS;
```

Our result at this stage is a single monomial described by a coefficient in *QNUM*[0], a list of commuting factors in *CNUM*[1:*J*], and a sequence of noncommuting factors in *QNUM*[1:*K*]. If there are several (*C*) traces with intercontracted indices, their respective extents in *QNUM* are delimited by LIM[0:1, 1:*C*]. We now proceed to the reduction of the trace or traces in *QNUM* to invariants; the reader who is interested only in ordinary algebra may skip to the section on storage and output.

TRACE REDUCTION

In quantum electrodynamics, pairs of gamma matrices whose indices are contracted arise in the calculation of processes which have photon vertices in their

Feynman diagrams. If there is more than one distinct fermion line in a diagram, at least some of the terms in the square of the matrix element will have the form of products of traces with the individual gamma matrices of the contracted pair occuring in separate traces (e.g., $\mathrm{tr}(\cdots \gamma_\nu \cdots) \, \mathrm{tr}(\cdots \gamma_\mu \cdots \gamma_\nu \cdots) \, \mathrm{tr}(\cdots \gamma_\mu \cdots))$, a situation I call "traces with intercontracted indices." Such products may be reduced to sums of single traces by successive applications of a formula due to Chisholm [13].

Let S and S' denote products of an odd number of gamma matrices, for example, $S = \mathbf{a}_1\mathbf{a}_2 \cdots \mathbf{a}_n$, where n is odd and $\mathbf{a}_i$ is the four-vector inner product $a_i \cdot \gamma$. A product with the same factors as S but written in reversed order will be denoted by $R = \mathbf{a}_n \cdots \mathbf{a}_2\mathbf{a}_1$. The reduction formula for the product of two traces is

$$\mathrm{tr}(\gamma_\mu S) \, \mathrm{tr}(\gamma_\mu S') = 2 \, \mathrm{tr}((S + R) \, S').$$

Successive applications of this formula, with appropriate rotations (cyclic permutations) of individual traces, suffice to reduce the product of several traces with intercontracted indices to a sum of single traces. Such a scheme cannot handle expressions which reduce to products of two or more traces without intercontracted indices; each factor would have to be evaluated separately in that case. Since $\gamma_5 = \gamma_0\gamma_1\gamma_2\gamma_3 = \gamma_3\gamma_2\gamma_1\gamma_0$, a γ_5 may be inserted anywhere in S or S' without altering anything.

In the input, the trace operator "tr" is symbolized by the string '##'; to make sense syntactically, the operator '##' should occur alone as a factor at parenthesis level zero. Its use is obligatory only in the case of products of traces, since the occurrence of noncommuting factors in a monomial implies that a trace is to be taken; in fact, even terms containing only commuting factors are multiplied by $\mathrm{tr}(1) = 4$. Similarly, gamma matrices with explicit indices which will be contracted either by *CHISHOLM* or *GMU* should occur alone in their respective factors, since a factor like $(a + \gamma_\mu)$ would lead to nonsense. The analysis of a monomial from *PROCESS* proceeds in several steps in *CHISHOLM*. First the intercontracted indices are identified and the traces rotated to normal form $(\gamma_\mu S)$ in the array *Y*. Index pairs which occur within the traces in *Y* are marked in *PAIR*, so that an unpaired index will be searched for in the remaining traces in *QNUM*. If the second member of a pair is not found, an error results. The initial trace in *Y* (for $J = 1$) may be entered rather clumsily; duplication of some of the ALGOL would make this step more efficient. Then 2^{C-1} simple traces are generated, with a zero or one bit in the appropriate binary place of *REV* determining whether the sequence *S* or its reverse *R* is used to build up the trace. Finally, multiple γ_5's are consolidated in a step which could be eliminated by a nonlocal Boolean variable set in *PROCESS* when none of the factor traces contains a γ_5.

procedure *CHISHOLM*(*N*,*C*); **value** *N*, *C*; **integer** *N*, *C*;

```
begin array SCRATCH[0:N], Y[0:N]; real MU; Boolean G5, ODD;
   Boolean array PAIR[0:N]; integer array UB[0:C];
   integer Q, I, J, K, L, U, Z, JZ, P, POW, REV;
   for K := 1 step 1 until N do PAIR[K] := false;
   J := 1; I := L := LIM[0,J]; U := LIM[1,J]; UB[0] := JZ := Z := 0;
B: for K := I step 1 until U, L step 1 until I-1 do
      begin Z := Z+1; Y[Z] := QNUM[K] end;
      JZ := JZ+1; UB[JZ] := Z; LIM[0,J] := -LIM[0,J];
   if JZ < C then begin I := 1;
   A: for K := I while Y[K] > 0 or PAIR[K] do I := I+1; MU := Y[I];
      for K := I+1 step 1 until U do if MU = Y[K] then
         begin PAIR[I] := PAIR[K] := true; I := I+1; go to A end;
   if J = 1 and I ≠ 1 then begin JZ := Z := 0;
      for K := 1 step 1 until U do PAIR[K] := false; go to B end;
   for J := 2 step 1 until C do begin L := LIM[0,J]; U := LIM[1,J];
      if 0 < L then for I := L step 1 until U do
        if QNUM[I] = MU then go to B end;
      outstring(2,'error:unpaired index'); C := 0 end;
   P := 2↑(C-1); Y[0] := P×QNUM[0];
for REV := P-1 step -1 until 0 do begin U := UB[1]; Z := -1;
  for I := 0, 2 step 1 until U do begin Z := Z+1; QNUM[Z] := Y[I] end
  for J := 2 step 1 until C do begin L := U+2; U := UB[J]; POW := 2↑(J-2);
    if (REV mod (2×POW))÷POW = 1 then begin
      for I := U step -1 until L do begin Z := Z+1; QNUM[Z] := Y[I] end
      end else for I := L step 1 until U do
               begin Z := Z+1; QNUM[Z] := Y[I] end;
    if 1 < J and J < C then begin MU := Y[U+1];
      I := 1; for K := I while QNUM[K] ≠ MU do I := I+1;
      K := 0; for Q := I+1 step 1 until Z, 1 step 1 until I-1 do
                 begin K := K+1; SCRATCH[K] := QNUM[Q] end;
      Z := K; for K := 1 step 1 until Z do QNUM[K] := SCRATCH[K] end;
  end J;
  G5 := ODD := false; I := 0;
  for K := 1 step 1 until Z do if QNUM[K] = 5 then G5 := not G5
    else begin I := I+1; QNUM[I] := QNUM[K];
      if G5 then ODD := not ODD end;
  if ODD then QNUM[0] := -QNUM[0];
  if G5 then begin I := I+1; QNUM[I] := 5 end;
  GMU(I); end REV;
end CHISHOLM;
```

At this stage we have only single traces. The procedure *GMU* eliminates any contracted indices by reduction formulas due to Caianiello and Fubini [14]. Let S and R be products of an odd number of gamma matrices as defined above; then within a (perhaps larger) product,

$$(\cdots \gamma_\mu S \gamma_\mu \cdots) = -2(\cdots R \cdots).$$

From this result and the basic anticommutation relation, an even string ($S\mathbf{b}$) between contracted gamma matrices reduces to two terms:

$$(\cdots \gamma_\mu S\mathbf{b}\gamma_\mu \cdots) = 2(\cdots \mathbf{b}S \cdots) + 2(\cdots R\mathbf{b} \cdots);$$

a variant of this case is treated separately by *GMU*:

$$\cdots\cdots \gamma_\mu\gamma_\mu \cdots = \cdots (4) \cdots .$$

The reduction proceeds by rewriting the trace in $S[J, 0{:}C]$, eliminating pairs of contracted indices until an even intermediate string forces $J := J + 1$ or until all indices (identifying integer negative) are gone. The simple traces are sent to *GAMMA* 5 which will call *PERMUTE* directly if no γ_5 is present.

```
procedure GMU(M); value M; integer M; begin
  array S[0:M÷2,0:M]; integer array R[0:M], N[0:M÷2];
  integer C, I, J, K, L, U, MU; Boolean LFTOVR;
  for I := 0 step 1 until M do S[0,I] := QNUM[I];
  J := 0; LFTOVR := false; N[0] := M; go to A;
MOREMU: L := I; U := L+1; K := 0;
  for I := U while MU ≠ S[J,I] do begin U := U+1;
    if C < U then begin comment index is unpaired, make it a vector;
                        S[J,L] := −MU; go to A end;
    K := K+1; R[K] := S[J,I]; end;
  for I := U+1 step 1 until C do S[J+1,I−2] := S[J,I−2] := S[J,I];
  LFTOVR := (K mod 2 = 0) and K ≠ 0; N[J] := C := C−2;
  if LFTOVR then begin S[J+1,0] := S[J,0] := 2×S[J,0];
    S[J+1,U−2] := S[J,L] := R[K]; J := J+1; N[J] := C;
    for I := 1 step 1 until K−1 do S[J,U−I−2] := R[I]; end
  else if K = 0 then S[J,0] := 4×S[J,0]
  else begin S[J,0] := −2×S[J,0];
             for I := 1 step 1 until K do S[J,U−I−1] := R[I] end;
A: C := N[J]; for I := 0 step 1 until C do
    begin QNUM[I] := MU := S[J+1,I] := S[J,I];
      if MU < 0 and I ≠ 0 then go to MOREMU end; GAMMA5(C);
```

```
if LFTOVR then begin for I := 0 step 1 until C do QNUM[I] := S[J-1,I];
                              GAMMA5(C); J := J-2 end
                    else J := J-1;
if -1 < J then go to A end GMU;
```

Traces without contracted indices fall into two classes, depending on whether they contain an even or odd number of γ_5's. In *PROCESS* and again in *CHISHOLM* any γ_5 present is anticommuted to one end of the trace; the reader is reminded that minor modifications are needed if his metric is such that $(\gamma_5)^2 \neq 1$. Thus an even number of γ_5's is equivalent to no γ_5 in the trace. Such traces are reduced to vector dot products by the standard recursion relation

$$\begin{aligned}\mathrm{tr}(\mathbf{a}_1\mathbf{a}_2 \cdots \mathbf{a}_n) = a_1 \cdot a_2 \,\mathrm{tr}(\mathbf{a}_3 \cdots \mathbf{a}_n) - a_1 \cdot a_3 \,\mathrm{tr}(\mathbf{a}_2 \cdots \mathbf{a}_n) \\ + \cdots + a_1 \cdot a_n \,\mathrm{tr}(\mathbf{a}_2 \cdots \mathbf{a}_{n-1}),\end{aligned}$$

which is derived from the basic anticommutation relation $\mathbf{ab} + \mathbf{ba} = 2a \cdot b$ (the trace is zero if n is odd). The number of terms generated is $(n-1)!!$ which becomes prohibitive from a temporal standpoint if n is much greater than 12. For long traces, tricks such as scanning for adjacent equal vectors (since $\mathbf{aa} = a \cdot a$) should be used before generating the dot products. The recursive generation of this reduction by actual anti-commutation in LISP[8] is better able to take advantage of identical vectors in a trace, although the process can be profligate in its use of storage. From a practical standpoint it may be more efficient to evaluate traces above a certain size numerically rather than to compute an algebraic result.

The procedure *PERMUTE* reduces an N factor trace in $QNUM[1{:}N]$ to terms containing $N/2$ vector dot products. *STORE* considers pairs of adjacent vector identifiers in *QNUM* to constitute dot products.

```
procedure PERMUTE(N); value N; integer N; begin
   integer array POS[0:N]; integer C, I, L, T;
   for L := 3 step 2 until N do POS[L] := L;
P: STORE(N); QNUM[0] := -QNUM[0];
   for L := 3 step 2 until N do begin C := POS[L]; T := QNUM[L];
     if C = 1 then begin
       for I := L step -1 until 2 do QNUM[I] := QNUM[I-1];
       QNUM[C] := T; POS[L] := L; end
     else begin QNUM[L] := QNUM[C-1];
               QNUM[C-1] := T; POS[L] := C-1;
               go to P end end end PERMUTE;
```

A trace containing an odd number of γ_5's is transformed into one with a single γ_5 (symbolized by the integer 5) as its last factor in $QNUM[M]$. The trace of a

product $\gamma_5 \mathbf{a}_1 \mathbf{a}_2 \cdots \mathbf{a}_n$ can be derived from the reduction formula above by finding four orthogonal vectors such that $\gamma_5 = \mathbf{abcd}$, for example $\mathbf{a} = \gamma_0$, $\mathbf{b} = \gamma_1$, $\mathbf{c} = \gamma_2$, and $\mathbf{d} = \gamma_3$; recall that n must be even and $\geqslant 4$ for a non-zero trace. The result is a sum over terms containing a determinant and a trace of $n-4$ gamma matrices without a γ_5:

$$\operatorname{tr}(\gamma_5 \mathbf{a}_1 \mathbf{a}_2 \cdots \mathbf{a}_n) = \sum_{{}_nC_4} (-1)^p (a_i, a_j, a_k, a_l) \operatorname{tr}\{\mathbf{a}_1 \cdots \mathbf{a}_{i-1} \mathbf{a}_{i+1} \cdots \mathbf{a}_n\}.$$

The sum is over all combinations of 4 elements ($ijkl$) from $a_1 \cdots a_n$ and the sequence of factors in the trace is the original one except that a_i, a_j, a_k and a_l have been deleted. The symbol (a_i, a_j, a_k, a_l) stands for the determinant of the 4×4 matrix

$$\begin{pmatrix} a_i \cdot a & a_i \cdot b & a_i \cdot c & a_i \cdot d \\ a_j \cdot a & a_j \cdot b & a_j \cdot c & a_j \cdot d \\ a_k \cdot a & a_k \cdot b & a_k \cdot c & a_k \cdot d \\ a_l \cdot a & a_l \cdot b & a_l \cdot c & a_l \cdot d \end{pmatrix},$$

and $p = i+j+k+l$, giving us the parity of the permutation $\binom{1\ 2\ 3\ 4\ 5\ \cdots\ n}{i\ j\ k\ l\ 1\ \cdots\ n}$. This algorithm is not optimal since the result includes invariant sums which are identically zero; anyone with interest in this type of trace should investigate alternatives. For example, in the 15 terms of $\operatorname{tr}(\gamma_5 \mathbf{qrstuv})$, the five term sum

$$(s, t, u, v)q \cdot r - (r, t, u, v)q \cdot s + (r, s, u, v)q \cdot t - (r, s, t, v)q \cdot u + (r, s, t, u)q \cdot v$$

may be shown to be zero by anticommuting $\mathbf{q}$ through the trace.

In the procedure *GAMMA5*, the four true elements of the array *DET* denote the current combination chosen from the *N* elements of *G*. The vectors in the determinant are sorted into increasing order on their internal integer representation and packed into the single word *CNUM*[30]. Both the determination of the parity of the permutation and the selection of the next combination are somewhat clumsy from the point of view of machine language programming.

```
procedure GAMMA5(M); value M; integer M; begin
  array G[0:M]; integer array X[0:3], D[1:4];
  Boolean array DET[0:M]; Boolean KEEP, SORT, EVEN;
  integer I, J, K, N, T, U;
if M mod 2 = 0 then PERMUTE(M)
else if QNUM[M] = 5 and M ≥ 5 then begin N := M−1;
  for I := 0 step 1 until N do
    begin DET[I] := I ≤ 4; G[I] := QNUM[I] end;
  for I := 0 step 1 until 3 do X[I] := 4−I;
```

```
CYCLE: K := J := 0; for I := 1 step 1 until N do if DET[I]
                        then begin J := J+1; D[J] := G[I] end
                        else begin K := K+1; QNUM[K] := G[I] end;
KEEP := SORT := EVEN := true;
for I := 4, I − 1 while SORT and KEEP do begin SORT := false;
  for J := 2 step 1 until I do begin T := D[J]; U := D[J−1];
    if T = U then KEEP := false
    else if T < U then
          begin D[J] := U; D[J−1] := T;
                SORT := true; EVEN := not EVEN end;
  end; end;
if KEEP then begin comment: pack the vectors in the determinant
  into a single word; CNUM[30] := D[4]+64×(D[3]+64×(D[2]+64×D[1]));
  if (X[0]+X[1]+X[2]+X[3])mod 2 = 1 then EVEN := not EVEN;
  if EVEN then QNUM[0] := G[0] else QNUM[0] := −G[0];
  PERMUTE(N−4); end;
for I := 0 step 1 until 3 do if X[I] < (N−I) then begin
  for J := I step −1 until 0 do begin DET[X[J]] := false;
    if J = I then X[J] := X[J]+1 else X[J] := X[J+1]+1;
    DET[X[J]] := true end; go to CYCLE end;
end; end GAMMA5;
```

Storage and Output

With all the traces involving the noncommuting quantities of Dirac algebra now reduced to sums of terms, each of which is a product of ordinary commuting factors, we face the problem of accumulating the individual terms and combining the numerical coefficients of terms with equal algebraic factors. The two methods of mass storage which will be discussed in detail are variants of table look-up schemes where the terms are added sequentially to a large array but ordered according to their internal structure by an auxiliary array whose elements point to terms in the large array. It is much easier to rewrite a small array of single word pointers than to shuffle multiple word terms to indicate a new ordering. In the first scheme, the auxiliary array gives a ranked list of the terms in the large array, but must be partly rewritten to insert a new pointer each time a different term is generated. However, the position of any term in the list can be quickly found by a binary search of the list, requiring only $\log_2 N$ term comparisons for N terms. A second method uses a so-called "tree" to provide the ordering information. The tree is a linked list with nodes and branches which can be read in such a manner to indicate

the serial order of the terms in the large array. To add a new term, a tree is expanded rather than rewritten; but it will, on the average, require a few more comparisons per term than the binary search because the details of its structure depend on the specific problem.

We have spoken of ordering algebraic terms as though they were numbers rather than symbolic information. This is possible, nonetheless, because within the computer each algebraic factor is represented by a bit pattern which can be interpreted as an integer. An n-factor term is analogous to an n-letter word, where our "alphabet" is the collection of small integers which represent possible factors. The problem of ordering terms is precisely that of alphabetizing words; but because multiplication is commutative at this level, only the letters occurring within a word are significant, not their sequence. To avoid ambiguity, a canonical order for the factors within a term is prescribed by ordering them according to their numerical representation. Factors with a small internal representation appear to the left in a term and change less frequently in the output listing. Thus to group terms according to specific common factors, it is necessary to be able to specify the internal representation which corresponds to an external print-name for an identifier; such a change is relatively minor.

The procedure *STORE* does two jobs: it transforms the current term into its canonical representation and then searches for an identical term among those previously generated. Explicitly, each term has a numerical coefficient in *QNUM*[0], algebraic constants in *CNUM*[1:*CNUM*[0]], a determinant in *CNUM*[30] (when this word is not zero), and *N*/2 vector dot products in *QNUM*[1:*N*]. Since the integers which represent the algebraic factors are small compared to the computer word size, an optimized program would pack several of these factors into a single word because efficient use of storage is necessary at this stage. However, packing and unpacking operations are awkward and unilluminating when expressed in ALGOL; we have simply used eight words of mass storage for each term, containing a coefficient and up to seven algebraic factors. To facilitate printout, the algebraic factors have been standardized at a four character length, with spaces being added to constants and dot products. To save storage space, such editing would normally be done by the output routine. A dot product containing the vectors *A* and *B* can

[illegible]

internal representation: i.e., $64\times$'*A*'+'*B*'. The use of two characters (12 bits in this case) to represent dot products and constants (when the dummy space is counted) is itself inefficient, since there are rarely more than ten or so vectors in a problem. Instead of manufacturing the dot product identifier from the integers representing the vectors, new constants corresponding to the dot products could be generated by the program (or assigned by the user who wishes to control sorting order); temporal efficiency would be maintained by finding the appropriate

constant in a two dimensional table whose indices are the integers representing the vectors. The final step before comparing the current term to those previously generated is to order the algebraic factors in *JOT* by sorting them on their numerical representation. For terms with many factors, some sorting time might be saved by sorting the algebraic constants in *PROCESS* and merging them with a sorted list of dot products here. We also declare certain nonlocal arrays and variables which will be used both by *STORE* and by the output statements:

```
  integer array TABLE[0:250], TREE[0:1000], NODE[0:50];
  array TERM[0:2000]; integer CENSUS, BIT, L, M, N;
procedure STORE(M); value M; integer M; begin
  integer I, J, K, N, P, Q, S, T, U; Boolean SORT; integer array JOT[0:8];
comment: First transform current term into canonical representation;
  K := CNUM[0]; for I := 1 step 1 until K do JOT[I] := CNUM[I];
  for I := 2 step 2 until M do begin T := QNUM[I]; U := QNUM[I−1];
    if U < T then begin S := T; T := U; U := S end;
    K := K+1; JOT[K] := 64×(64×(64×'␣'+T)+U)+'␣';
  JOT[K+1] := U := CNUM[30]; if U ≠ 0 then K := K+1; P := K;
  for I := K+1 step 1 until 7 do JOT[I] := '␣␣␣␣'; SORT := true;
  for I := K while SORT do begin K := K−1; SORT := false;
    for J := 2 step 1 until I do begin T := JOT[J]; U := JOT[J−1];
      if T < U then begin JOT[J] := U; JOT[J−1] := T; SORT := true;
  end end end;
comment: Now compare term with pre-existing ones and combine or append;
  N := J := BIT;
LOOK: if N = 0 then go to ENTER; N := N÷2;
  if J ⩽ CENSUS then begin P := TABLE[J];
    for K := 1 step 1 until 7 do begin Q := sign(JOT[K]−TERM[P+K]);
      if Q ≠ 0 then begin J := J+Q×N; go to LOOK end end;
    TERM[P] := TERM[P]+QNUM[0]; go to FIN end
  else begin J := J−N; Q := −1; go to LOOK end;
ENTER: for N := CENSUS step −1 until J do TABLE[N+1] := TABLE[N];
  P := 8×CENSUS; if Q > 0 then TABLE[J+1] := P else TABLE[J] := P;
  TERM[P] := QNUM[0]; for K := 1 step 1 until 7 do TERM[P+K] := JOT[K];
  CENSUS := CENSUS+1; if CENSUS ⩾ 2×BIT then BIT := 2×BIT;
FIN: end STORE;
```

The details of this storage algorithm will be discussed below after we give output and initialization statements for it. All the editing of the output occurs in the procedure *OUTERM* which prints out individual terms so that a vector dot

product $A \cdot B$ appears as '␣*AB*␣', a constant *M* as '␣*M*␣␣', and a determinant (*A*, *B*, *C*, *D*) as '*ABCD*', corresponding to the way these quantities were stored in *STORE*, *PROCESS*, and *GAMMA5* respectively. The reference string in *outsymbol*[*10*] is identical to that appearing previously in *insymbol*. Output of terms with zero coefficients has not been suppressed.

```
procedure OUTERM(P); integer P; begin
  integer J, K, S, T; integer array SYMBOL[0:3];
  outreal(2,TERM[P]);
  for J := 1 step 1 until 7 do begin S := TERM[P+J];
    for K := 0, 1, 2, 3 do
      begin T := S÷64; SYMBOL[K] := S−64×T; S := T end;
    for K := 3, 2, 1, 0 do
      outsymbol(2,'×)+−(.0123456789␣#
              ABCDEFGHIJKLMNOPQRSTUVWXYZ;',SYMBOL[K]) end;
end OUTERM;
comment output for statement;
for N := 1 step 1 until CENSUS do OUTERM(TABLE[N]);
comment initialization; CENSUS := 0; BIT := 1;
```

The second part of *STORE* accumulates terms in the mass storage array *TERM* serially as different algebraic products appear. The method given in *STORE* above is a binary search algorithm in which the terms are ordered by pointers kept in the indexing array *TABLE*; the precise function of these pointers can be understood by examining the for statement which causes the terms to be printed out in lexicographical order. Following the label *LOOK*, we construct an index *J* by a succession of comparisons of the current term in *JOT* with those already in *TERM* and ordered by *TABLE*. The index is increased or decreased each time by a successively diminishing power of two until either an identical term is found or *J* and *Q* indicate where the pointer to the new term must be inserted in *TABLE* by the statements following the label *ENTER*. Although a binary search affords the minimum number of term comparisons, the time spent in rewriting *TABLE* can become prohibitive when *CENSUS* is large. To sidestep this problem at the cost of some comparison inefficiency, we consider a "tree" storage scheme.

A tree is an involved structure which indicates the results of successive term comparisons in its generation; it can be interpreted to yield a lexicographical ordering of the terms it describes. The basic component of the tree is what I shall call a node, which consists of four words of information. The zeroth element of a node, *TREE*[4*n*], contains a pointer to the base of the associated term. Its first and third elements are "branches" which may eventually point to nodes one level further into the tree which describe terms that are either smaller or greater;

respectively, than the term corresponding to this node; initially these elements are zero. The second element, *TREE*[4*n*+2], provides a link back to a previous node whose first or third element points to this node. A victim of initialization, the zeroth node, or trunk of the tree, does not conform to the above. The ALGOL for the generation of a tree is somewhat simpler than that for the binary search scheme; the readout is not. To implement this method, replace the statements in *STORE* between the second comment and the label *FIN* by:

```
  N := Q := 0;
LOOK: T := TREE[N+2+Q]; if T = 0 then go to ENTER;
  N := T; S := TREE[N];
  for K := 1 step 1 until 7 do begin Q := sign(JOT[K]−TERM[S+K]);
    if Q ≠ 0 then go to LOOK; end;
  TERM[S] := TERM[S]+QNUM[0]; go to FIN;
ENTER: CENSUS := CENSUS+4; S := 2×CENSUS;
  TREE[CENSUS] := S; TREE[CENSUS+2] := N;
  TREE[N+2+Q] := CENSUS; TERM[S] := QNUM[0];
  for K := 1 step 1 until 7 do TERM[S+K] := JOT[K];
```

Decoding the tree proceeds by advancing from node to node while the current node indicates a smaller term further into the tree. When the smallest term on a branch is found, it is printed out and the corresponding pointer is marked by setting it negative. We then ask whether that node indicates a larger term on the next level. If so, we step to its node and repeat the scanning for smaller terms; otherwise we retreat along the nodes until we find a term which has not been printed. After printing and marking it, we search for a larger term. When the retreat reaches the zeroth node, the output is complete and the tree is initialized.

```
    N := 4;
A:  M := TREE[N+1]; if M ≠ 0 then begin N := M; go to A end;
    M := TREE[N]; OUTERM(M); TREE[N] := −M;
B:  if TREE[N+3] ≠ 0 then begin N := TREE[N+3]; go to A end;
C:  if TREE[N] < 0 then begin N := TREE[N+2]; go to C end;
    if N ≠ 0 then begin OUTERM(2×N); TREE[N] := −2×N; go to B end;
CENSUS := 0; for N := 2, 1 step 2 until 1000 do TREE[N] := 0;
```

Since the structure of the tree depends on the order of generation of and on the nature of its contents, the storage of a sequence of terms already in order could result in a tree with a single branch. To forestall this calamity, one can compute a checksum from the factors of a term and include it in the comparison sequence. Use of a checksum would also reduce the number of steps in a comparison of different terms with identical initial factors. Both the binary search and tree storage

methods above have employed a fixed maximum for the number of factors in a a term and padded a term containing fewer factors with blanks. Because terms are located by a pointer to their initial element, there is no reason why they cannot be of arbitrary length if some indication of the number of factors is given for the purposes of comparison and printing. As a final illustration of methods for handling immediate access storage, we modify the tree storage method to utilize a checksum and permit arbitrary term length. Since there are the same number of nodes and terms and they occupy corresponding positions in *TREE* and *TERM* respectively, we can eliminate *TREE* by incorporating both the algebraic information and its associated node into *TERM*. This obviates the need for the pointer in the zeroth element of the node, which incidentally was superfluous already in the case of a fixed length term. By recording the path taken into the tree during readout, the back link information in the second element can be regenerated when it is needed. The statements required in *STORE* are:

```
   N := Q := 1; for J := 1 step 1 until P do Q := Q+JOT[J];
   Q := Q mod 37; JOT[0] := 64×Q+P; Q := sign(2×Q−37);
LOOK: T := TERM[N+Q]; if T = 0 then go to ENTER; N := T;
   for J := 0 step 1 until P do begin Q := sign(JOT[J]−TERM[N+2+J]);
     if Q ≠ 0 then go to LOOK; end;
   TERM[N] := TERM[N]+QNUM[0]; go to FIN;
ENTER: TERM[CENSUS] := QNUM[0];
   TERM[CENSUS+1] := TERM[CENSUS−1] := 0;
   for J := 0 step 1 until P do TERM[CENSUS+2+J] := JOT[J];
   TERM[N+Q] := CENSUS; CENSUS := CENSUS+P+4;
```

The price paid for pruning the tree to half its former size is a more complicated defoliation routine. The only part of the tree left to mark to indicate the printing of a term is one of the pointers in a previous node corresponding to the first or third element of the full node (one could mark a bit in the term itself, e.g., the sign of the checksum). To be able to climb down out of the tree and to access the pointers correctly, one must record the positions of successive nodes and indicate whether one proceeded to a larger or smaller term in the next node. Output will be lexicographically ordered only within groups of terms with the same checksum; groups must be further merged for a simple ordering. If *OUTERM* is modified to print a variable number of factors for each term, the output and initialization statements become:

```
   N := 1; L := 0;
A: M := TERM[N−1]; if M ≠ 0 then
     begin L := L+1; NODE[L] := −N; N := M; go to A end;
```

```
    OUTERM(N); M := NODE[L]; TERM[abs(M+1)] := -N;
B:  M := TERM[N+1]; if M ≠ 0 then
        begin L := L+1; NODE[L] := N; N := M; go to A end;
C:  M := NODE[L]; if TERM[abs(M+1)] < 0 then
        begin N := abs(M); L := L-1; if L > 0 then go to C end;
    if L > 0 then begin OUTERM(N); TERM[abs(M+1)] := -N; go to B end;
    if TERM[2] > 0 then go to B;
CENSUS := 5; for N := 0 step 1 until 3 do TERM[N] := 0;
```

We have dwelt upon the efficient use of a computer's internal storage on a "first pass" basis without considering the problem of reclaiming space from terms whose numerical coefficients become zero or that of spilling parts of long expressions onto external media such as tape or disc. Such questions can be better dealt with in the context of specific implementations. It might even be that in some systems a more efficient accumulation of terms would result from buffered intermediate output of terms onto disc or tape, perhaps without any preliminary combining, and a final sort and merge operation.

Afterword

A program composed of the procedures defined above will simply evaluate an input expression by expanding it, reducing any traces present, and combining identical terms. One may want to impose additional conditions on the result such as $p \cdot p = m^2$ and $k \cdot k = 0$, or want to substitute algebraic expressions for dot products or constants so that the answer is in terms of a minimal set of invariants. Such substitutions involve internally generating new input from the output of the first evaluation and further expansion and combination of the new expressions. Large parts of the code in *DISTRIBUTE*, *PROCESS*, and *STORE* can be used for this problem which is now one of ordinary algebra. Additional statements including portions of *SCAN* are needed to set up, from the input, a table of equivalents for algebraic constants and vector inner products which includes operator and parenthesis level information for expressions. Nonzero terms from the first expansion are unpacked and the factors with equivalents are replaced; and any products with multiple term factors are distributed and the resulting monomials are accumulated as before.

Once a program does more than a single task or type of manipulation, it is necessary to select from among its several capabilities the function appropriate to the data at hand. Is the current input a trace to reduce, equivalents to tabulate and substitute in an existing expression, information about the desired internal representation or algebraic type of identifiers, or possibly instructions to punch

the output? One approach, of great flexibility, is to preface data relevant to a single task with a control word or symbol which is recognized and causes a transfer to the appropriate segment of the program. Processing continues in this task-segment until an end-of-task delimiter is seen, whereupon a new control word is sought. Another approach, which recognizes the individual nature of any complex problem, involves writing a library of procedures for single tasks or sub-tasks which use consistent conventions for internal representation of algebraic quantities. Those procedures appropriate to the problem are selected and a short control program written which consists mostly of procedure calls and determines the sequence of operations such as input, reduction, substitution, and output. Thus one can omit trace reduction procedures for problems involving ordinary algebra, or include additional procedures to do tensor algebra or to manipulate external storage media (tape, disc) in case of lengthy expressions.

Finally, I should like to pull together and extend some of my scattered remarks on the relation between the illustrative coding given here in ALGOL and bit pushing techniques, the more or less optimized machine language coding which can take advantage of the idiosyncrasies of specific computers to achieve temporal or spacial efficiency. We have been constrained by the conventions of ALGOL implementations which generally make a full computer word the quantum of storage which is most readily accessible within the reference language. Thus in some places we have used full Boolean words (the arrays *MULT, DET*) where only a single bit of information was needed, and in others manipulated the sign bit of related integers (in *V, QNUM, TREE*) since it is virtually the only individually accessible bit in ALGOL expressed code. Character manipulation and the operations involved in packing and unpacking information within a single word have [illegible] Indeed, the ultimate choice of the maximum internal representation of algebraic quantities (and hence the maximum number of different quantities) will probably be related to the character size on a specific computer and to the types of partial word operations available. There are compromises to be made in the construction of a program which bear upon the amount and type of optimization which is attempted. In *STORE*, the time taken in packing and unpacking words must be weighed against the ease of multiple word comparisons but may, on the other hand, be necessitated by space restrictions. When different segments of a program involve similar operations, one must decide whether to write one procedure and call it from several places, or to duplicate statements and save the time required for initialization and linking. In general only the procedures and statements deep within inner loops need to be fully optimized, while some inefficiencies can be tolerated in input analysis and task control. Efficiency at a given job and the versatility to do many different manipulations are usually inversely correlated, as are the overall efficiency or complexity of a program and the time needed to develop

and debug it. For small or infrequent calculations, any program which works may be sufficient. The intricacies of input and output editing have been sidestepped because they are tedious, machine dependent, and undoubtedly good practice for the would-be algebra programmer. Likewise the specification of and recovery from error conditions has been minimally treated. We have stressed the desireability of assignable internal representations for algebraic quantities which may have several characters in their external print-names and mention here the possible usefulness of an output format constructed to be compatible with the input format so that intermediate results can be saved and manipulated further.

Acknowledgments

Some of these algorithms were originally developed in the machine language program FTRACE [7] under the supervision of C. K. Iddings. I am indebted to M. J. Levine for his version of Kaiser's distribution algorithm. Cultural interchange regarding algebraic manipulations on digital computers has occurred at various times with J. Mathews, M. Veltman, J. McCarthy, A. C. Hearn, and M. J. Levine. Support for computer time has come from many sources: the National Science Foundation, the U.S. Atomic Energy Commission, the U.S. Air Force Office of Scientific Research, and the Office of Naval Research.

References

1. Jon Mathews, California Institute of Technology, private communication. (a Balgol program for traces of up to 8 gamma matrices, and a machine language program for spin-one algebra)
2. R. M. Wilcox, A computer method for calculating Dirac traces with applications to quantum electrodynamics. Ph.D. thesis, University of Colorado, 1961. (will not handle γ_5's easily)
3. H. J. Kaiser, Trace calculation on electronic computer. *Nuclear Physics* **43**, 620 (1963). (distribution algorithm for an arbitrary number of nested parentheses)
4. M. J. Levine, Neutrino processes in stars. Ph.D. thesis, California Institute of Technology, 1963. (low order perturbation theory, mostly table look-up)
5. M. J. Levine, Dirac matrix and tensor algebras on a computer. *J. Computational Physics* **1**, 454 (1967).
6. M. Veltman, An IBM 7090 program for symbolic evaluation of Feynman diagrams. Stanford Linear Accelerator Center, unpublished manuscript, *ca.* 1964. (spin-one algebra)
7. S. M. Swanson, "FTRACE: A FAP subroutine for Dirac gamma matrix algebra," Institute of Theoretical Physics, Stanford University, unpublished report ITP-120, June 1964; Addenda and errata to ITP-120, November 1966; cf. also Stanford Linear Accelerator Center program library, program A040.
8. A. C. Hearn, Computation of algebraic properties of elementary particle reactions using a digital computer. *Communications of the ACM* **9**, 573 (1966). (summarizes an ambitious undertaking in general algebra and diagram manipulation written in LISP)
9. "Revised Report on the Algorithmic Language ALGOL 60," P. Naur, editor, *Communications of the ACM* **6**, 1 (1963).
10. *Communications of the ACM* **7**, 628 (1964).

11. "Burroughs B5500 Information Processing System, Extended Algol Reference Manual," Burroughs Corporation, Detroit, Michigan, 1964.
12. M. J. LEVINE, private communication. The present ALGOL version was translated from an assembly language version and flow chart furnished to me by Levine.
13. J. S. R. CHISHOLM, *Nuovo Cimento* **30**, 426 (1963).
14. E. R. CAIANIELLO and S. FUBINI, On the algorithm of Dirac spurs. *Nuovo Cimento* **9**, 1218 (1952). Proof of Chisholm's reduction also follows from formula A3 of this reference.