JOURNAL OF COMPUTATIONAL PHYSICS 5, 280–327 (1970)

Symbolic Analysis of Feynman Diagrams by Computer*

J. A. CAMPBELL

Department of Physics, Center for Particle Theory, and Department of Computer Science, University of Texas at Austin, Austin, Texas 78712

ANTHONY C. HEARN†

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

Received August 28, 1969

We describe a system of programs in the language LISP 1.5 which handles all stages of calculation from the specification of an elementary-particle process in terms of a Hamiltonian of interaction or Feynman diagrams to the derivation of an absolute square of the matrix element for the process. Examples of significant parts of the programs are presented in the text, while a detailed listing of this material is contained in two Appendices which are available on request from the authors.

1. INTRODUCTION

Many of the calculations for scattering or decay processes in elementary-particle physics are based on methods of perturbation theory. Feynman [1], in particular, introduced a classification of parts of perturbative expansions from quantum field theory in terms of diagrams that describe the evolution of a process with time. If there is more than one Feynman diagram for a process, each diagram shows the contribution of one possible intermediate state to the interaction. The order of perturbation theory with which any diagram is associated is equal to the number of vertices in that diagram, and each line refers to one real or virtual particle in the process. The matrix element (in the quantum-mechanical sense)

* Research sponsored in part by the Air Force Office of Scientific Research Office of Aerospace Research, U.S. Air Force, under AFOSR Contract No. F44620–68–C–0075, and in part by the National Science Foundation. Computer time supported in part by the Stanford Artificial Intelligence Project under the sponsorship of the Advanced Research Project Agency of the Office of the Secretary of Defense (SD–183).

† Alfred P. Sloan Foundation Fellow. Present address: Department of Physics, University of Utah, Salt Lake City, Utah 84112.

which corresponds to any diagram can be derived unambiguously from the diagram by the application of a well-defined set of rules [2]. The total matrix element for a process with given initial and final states is equal to the sum of matrix elements for all the diagrams with those initial and final states.

In quantum electrodynamics, where perturbative expansions in the fine-structure constant have so far proved to be sufficient for the interpretation of any experiment, simple application of Feynman rules for diagrams can lead very quickly to large numbers of terms in the matrix elements. Further, the number of terms or diagrams that must be considered for accurate interpretation of an experiment rises rapidly with either the characteristic energy or momentum transfer used, and probably even more rapidly when particles of high spin can enter into a process. Thus, manual calculations of interest for present high-energy experiments, particularly at the 20-GeV Stanford linear accelerator, can easily get out of hand, and some form of mechanized computation is essential if the results are to be obtained accurately within a reasonable time.

The need for automated computation was first recognized for traces of products of Dirac matrices and four-momenta in quantum electrodynamics. Several authors have programmed this self-contained problem in various combinations of machine code and FORTRAN ([3], [4], [5], [6], [7], [8], [9]) or LISP ([10], [11]). Two of the programs ([6], [10]) have developed into general symbol-manipulating schemes whose scope extends beyond theoretical physics. However, the evaluation of traces is first carried out, in general, *after* a matrix element has been obtained and squared, and is used for reduction of the absolute square to a simplified quantity, such as a cross-section or polarization for comparison with experiment. In many cases, it is also difficult to proceed from basic quantum field theory through Feynman diagrams to a matrix element. That procedure cannot be defined as succinctly as the trace calculation, so that it has received less attention in the past. Because FORTRAN is not flexible enough for the task, and machine code in large volumes is tedious to write, programs ([10], [12]) that have analyzed Feynman diagrams have been written in the symbol-manipulating language LISP 1.5 [13]. While some of the results that they have produced have been discussed in papers which have emphasized questions of theoretical physics, there has been no previous detailed account of the programming techniques involved. This paper is intended to fill that gap and report the most settled conclusions of our work on the subject since the first announcement of work in this area in 1964 [14].

We begin in Section 2 with a description of basic LISP and the syntax we shall use in later sections. In Section 3 there is a discussion of how Feynman diagrams are represented in the computer, and how properties are assigned to lines in diagrams. Section 4 covers the synthesis of Feynman diagrams from quantum field theory with the help of Wick's theorem. The derivation of unsimplified matrix elements from diagrams is analyzed in Section 5. The two principal problems of

simplification of matrix elements are where to make judicious use of Dirac matrix algebra to avoid excessively long results, and how to recognize the integrals that arise if a diagram contains sets of lines that form closed loops. We present the programming methods that we have applied to these problems in Sections 6 and 7, respectively. In Section 8 we discuss the relatively short operation of squaring of matrix elements, which is the last step in the analysis based on Feynman diagrams, before the computed expressions are turned over to other programs which perform traces and a final general simplification. We adopt the view that the expressions produced by our programs should be in a form acceptable to any general symbolic simplifying program, which need not be written in LISP. The paper concludes in Section 9 with a survey of heuristic methods that we either have used for specific problems or believe will be needed for the treatment of future problems.

Although we have referred above to quantum electrodynamics, our programs can deal equally well with other interactions that are represented graphically by Feynman diagrams. In those cases, one is generally not interested in calculation because of a naive view of perturbation theory, which may not even lead to a convergent answer, but because of assumptions that only a few possible intermediate states dominate a process, or that the structure of singularities for an exact matrix element can be deduced or guessed from the singularities of low-order terms in a divergent perturbative expansion [15].

We have included examples of parts of programs in the text where they can illustrate a point well and are only a few lines long. Not all examples are exact renditions of extracts from the programs; occasionally, we write a simplified example to save space and make the discussion more readable. Exact LISP definitions of the algorithms that form the backbone of the programs are given, with explanatory headings, in Appendix 1. Appendix 2 contains some heuristic procedures to which we have referred in Sections 4 and 9. Because the detailed programs are long and probably of interest to fewer people than the total number of readers of this paper, we do not include the Appendices with this paper. They may, however, be obtained on request from the authors.

If we use the name of a LISP function without quoting its definition, it is a function that is a regular part of a LISP system. Descriptions of some of the simplest basic functions occur in Section 2, but other sources of description are the LISP 1.5 Programmer's Manual [13], Weissman's introductory LISP textbook [16] and a recent article [17] presenting a standardization of the various LISP dialects now available.

2. A Language for the Description of Programs

Apart from a few short functions written directly in machine code, all the programs discussed in this paper are written in Standard LISP [17], a uniform subset

Similarly, a number cannot begin with a period; 0.5 is a legal LISP atom, but .5 is not.

In general, an S expression is either an atom or any structure which consists of a left parenthesis, an S expression, a period, an S expression, and a right parenthesis, in that order. Briefly, we can write

$$\langle \text{S expression} \rangle : : = \langle \text{atom} \rangle | (\langle \text{S expression} \rangle \cdot \langle \text{S expression} \rangle) \tag{2.1}$$

Equation (2.1) is an example of syntactic definition in Backus normal form [19]. In this notation, we introduce the special symbols ': : =', '⟨', '⟩', and ' | '. Equation (2.1) means that an S expression is either an atom or is a left parenthesis followed by an S expression, a period, another S expression and right parenthesis, in that order. The vertical bar is the logical connective 'or', and the angular brackets always enclose elements of the syntax that are being defined. Other combinations of symbols stand for themselves.

Using Eq. (2.1), we can build up S expression structures such as

$$((\text{APPLE} \,.\, 1.2) \,.\, ((\text{X} \,.\, 2) \,.\, \text{F*})),$$

but in such examples the problem of placing all periods correctly, rapidly gets out of control. Periods are not a natural feature of *list structure*, which ideally should have a form like (A (B C) D (E F) G), for example. In LISP, this is made possible by introduction of the special atom NIL which acts as a terminator of lists, and which can be suppressed in input and output if it occurs as the second member of any dotted pair. We then have the formal equivalence

$$(\text{X} \,.\, \text{NIL}) \equiv (\text{X}) \tag{2.2}$$

If we add a new element to an existing list by the prescription Eq. (2.2), the resulting S expression can be simplified by a second important equivalence

$$(\text{X} \,.\, (\text{Y})) \equiv (\text{X Y}) \tag{2.3}$$

There is of course no reason why every dotted pair should have NIL or a list as its second member, and it is, therefore, possible to mix both dot and list notation in the same S expression. The legality of the S expression can always be tested by conversion to dot notation throughout using Eqs. (2.2) and (2.3), and then comparison of the resulting structure with Eq. (2.1).

Equations (2.2) and (2.3) deal with the LISP atoms X, Y, and NIL. However, in discussing conversion of dot to list notation, we can interpret X and Y to stand for arbitrary S expressions. Therefore, there is an ambiguity in the meaning of the equivalences. To resolve the ambiguity, we need in addition to the programming language which contains S expressions a language in which we can make statements

about the manipulation of S expressions. This is the LISP meta-language. In contrast to S expression atoms, which stand for themselves and include uppercase alphabetic characters, atoms ('names') in the meta-language are written in lowercase form. In the meta-language, Eqs. (2.2) and (2.3) become the unambiguous forms

$$[y\ .\ NIL] = [y] \tag{2.4}$$

and

$$[x\ .\ [y]] = [x;y] \tag{2.5}$$

NIL always stands for itself, so it remains in uppercase letters.

The basic parts of the meta-language which we shall use are:

$$\begin{aligned} &\langle \text{letter} \rangle : : = a \mid b \mid c \ldots\ldots \mid z \\ &\langle \text{decimal digit} \rangle : : = 1 \mid 2 \mid \ldots\ldots \mid 9 \mid 0 \\ &\langle \text{blank space} \rangle : : = \end{aligned} \tag{2.6}$$

The "name", the meta-language equivalent of the atom, can be defined in two stages:

$$\begin{aligned} \langle \text{partial name} \rangle : : = &\langle \text{blank space} \rangle \mid \\ &\langle \text{letter} \rangle \langle \text{partial name} \rangle \mid \\ &\langle \text{decimal digit} \rangle \langle \text{partial name} \rangle \\ \langle \text{name} \rangle : : = &\langle \text{letter} \rangle \langle \text{partial name} \rangle \end{aligned} \tag{2.7}$$

NIL in Eq. (2.4) is effectively a constant. We can include such constants in a definition of the meta-language by writing

$$\langle \text{constant} \rangle : : = \langle \text{S-expression} \rangle \tag{2.8}$$

Variables, by contrast with constants, may be represented by *any* name in the meta-language. Hence, we have

$$\langle \text{variable} \rangle : : = \langle \text{name} \rangle \tag{2.9}$$

Bearing in mind the equivalence between dot and list notation in Eqs. (2.4) and (2.5), we can define the meta-language analogue of the S expression from Eq. (2.1):

$$\begin{aligned} \langle \text{M-expression} \rangle : : = &\langle \text{name} \rangle \mid \\ &[\langle \text{M-expression} \rangle\ .\ \langle \text{M-expression} \rangle] \end{aligned} \tag{2.10}$$

The syntax defined by Eqs. (2.4) to (2.10) is sufficient for examination of items of data in LISP. For any M expression, like [a;b;c], manipulation requires

three basic operations—one which recovers the first element from the list, one which produces the remainder of the list, and one which adds an element to the existing list. The names of these three operations are *car*, *cdr*, and *cons*, whose effects are as follows:

$$\begin{aligned} &\text{car[[a;b;c]]} = \text{a} \\ &\text{cdr[[a;b;c]]} = \text{[b;c]} \\ &\text{cons[a;[b;c]]} = \text{[a;b;c]} \end{aligned} \tag{2.11}$$

Operators such as *car*, *cdr* and *cons* are called *functions*. In addition to these, the LISP language contains several other basic functions whose definitions are in machine code, outside the scope of the meta-language. Two such functions are the predicate functions *atom* and *eq*. These are predicate functions in the sense that their values are always truth values, 'true' being represented by the LISP atom T, and 'false' by the atom NIL. For example,

$$\text{atom[u]}$$

has the value T if u is an atom, and NIL, otherwise. Similarly, if u and v are atoms, then the function

$$\text{eq[u;v]}$$

is true if u is the same atom as v and false, otherwise. To test for equality of general S expressions requires a more complicated function *equal*, whose definition can be given in the meta-language and may be found in the cited LISP references.

We are now in a position to define more complicated functions in terms of the basic functions *car*, *cdr*, *cons*, *eq* and *atom*. A general LISP function has a name, a list of arguments, and a definition. To specify a function definition and complete our description of the LISP meta-language we write

$$\begin{aligned} &\langle\text{form}\rangle ::= \langle\text{constant}\rangle \mid \langle\text{variable}\rangle \mid \\ &\qquad \langle\text{function}\rangle[\langle\text{argument}\rangle;...;\langle\text{argument}\rangle] \mid \\ &\qquad \textit{if}\ \langle\text{form}\rangle\ \textit{then}\ \langle\text{form}\rangle\ \textit{else}\ \langle\text{form}\rangle \\ &\langle\text{list of variables}\rangle ::= [\langle\text{variable}\rangle\ ;\ \langle\text{variable}\rangle;...;\langle\text{variable}\rangle] \\ &\langle\text{argument}\rangle ::= \langle\text{form}\rangle \\ &\text{and}\ \langle\text{function}\rangle ::= \langle\text{name}\rangle \mid \lambda\ [\langle\text{list of variables}\rangle;\ \langle\text{form}\rangle] \end{aligned} \tag{2.12}$$

The syntax presented here, which is adequate for the purposes of this paper, is a subset of those definitions found in Ref. [13] and [20]. The λ in the definition of ⟨function⟩ is a convention first proposed by Church [21] as an identifier for a list of arguments in the specification of a function.

The use of Eq. (2.12) in defining new functions is best seen by several examples. First, it is often necessary to make multiple applications of the functions *car* and *cdr* to a list to obtain a certain part of that list. Thus, we can introduce the function

$$\text{cadar [u]} = \text{car [cdr [car[u]]]} \tag{2.13}$$

Compound calls of *car* and *cdr* up to four applications are allowed in Standard LISP, and abbreviated by sandwiching the relevant a's and d's between a single c and r as in Eq. (2.13).

Another simple function tests for the atom NIL, and is, therefore, useful in determining the end of lists. This is a predicate function defined by

$$\text{null[u]} = \text{eq[u ; NIL]}. \tag{2.14}$$

The alternative *if, then else* construction in Eq. (2.12) permits the expression of a finite number of possible values for the function. Its general form, known as a *conditional expression*, is

if p_1 *then* e_1 *else if* p_2 *then* e_2 $\cdots$ *else* e_n .

Each p_i stands for a predicate function with an argument or arguments. The form is to be scanned from left to right until one p_i is found whose value is true. The value of the entire expression is then the value of the corresponding e_i . If no p_i is true, the value of the expression is e_n . As an example, consider the one-dimensional step function, whose value is zero if its one numerical argument is negative, one if the argument is positive, and 0.5 if the argument is zero. In the meta-language, its definition is:

$$\begin{aligned} \text{step [n]} = {} & \textit{if}\ \text{minusp[n]}\ \textit{then}\ 0.0 \\ & \textit{else if}\ \text{zerop[n]}\ \textit{then}\ 0.5 \\ & \textit{else}\ 1.0 \end{aligned} \tag{2.15}$$

A specification in terms of a conditional expression may be used to display the very powerful LISP feature of recursion, i.e., the definition of a function in terms of itself. Suppose that we wish to write a function which reverses the elements in a list. We can define it in terms of an auxiliary function *revl* as follows:

$$\text{reverse [u]} = \text{revl [u; NIL]}$$

$$\text{revl [u;v]} = \textit{if}\ \text{null [u]}\ \textit{then}\ \text{v}\ \textit{else}\ \text{revl [cdr [u];cons [car [u];v]]} \tag{2.16}$$

It is an instructive exercise to let u = (A B C) and carry through the operation of Eq. (2.16) by hand. The function finally terminates, because (A B C) is equivalent to (A . (B . (C . NIL))).

Most of the standard numerical operators occur as machine-coded functions in the basic LISP language. However, in LISP, numerical and general algebraic expressions are usually written in *prefix* form rather than the standard mathematical infix notation. Thus, the expression

$$1.2 + 3.4 * 5$$

is written in S expression form as

(PLUS 1.2 (TIMES 3.4 5))

or in M expression form as

plus [1.2 ; times [3.4 ; 5]].

The alphabetic notation for the operators + and * is used since the standard mathematical operators are not legal LISP atoms.

We now consider two extensions of the language which are particularly useful in many applications. First, we are permitted to use functional arguments in defining functions. Thus, if we want to apply a function f to every member of a list [u;v;..] to obtain the result [f[u];f[v];..], we can use the function defined by maplist [u;f] = *if* null [u] *then* NIL *else* cons [f[u];maplist[cdr[u];f]].

Secondly as an alternative to the recursive definition of functions as in Eq. (2.16), it is sometimes more convenient to define a function by an iterative sequential program using the LISP program feature. An iterative version of Eq. (2.16) is

reverse [u] = prog [[x]; A; *if* null [u] *then* return [x];
x: = cons[car[u]; x]; u:= cdr[u]; go [A]] (2.17)

This definition displays all the special parts of the program feature. *prog* has an indefinite number of arguments, the first of which is a list of *program variables* used inside the program feature. These variables are initially set to NIL. The use of *go*, *return* and labels such as A is evident by inspection of Eq. (2.17). The infix operator := sets its first argument to the value of the second argument. The particular property that conditional expressions do not require a terminating *else* clause within a program feature is also shown.

Three other simple functions which we shall often use in the definitions of more complicated functions are *member*, *append* and *delete*. *member* [a;b] is a predicate whose value is T if a is a member of the list b and NIL otherwise, *delete*[a;b] has a value equal to b from which the first occurrence of a has been deleted and *append*[a;b] combines two list structures a and b into a single list. These functions are easy to define in meta-language, and the reader is recommended to do this for himself before proceeding further.

Another important part of the language which we shall make much use of is the concept of the *property list* of an atom. Each LISP atom has its own private list of properties, on which we can record attributes. For example, if we use P1 to represent the four-momentum of an antiparticle in a calculation in particle physics, we can put a label (a 'flag') ANTIPTL on the property list of this LISP atom, using the function *flag*. In practice,

$$\text{flag[x;ANTIPTL]} \tag{2.18}$$

places the flag ANTIPTL on the property list of each atom in the list x. We can later test for the presence of this flag by using the predicate function *flagp*. More generally, if the attribute which we wish to assign to the atom has a range of values, we can place on its property list an indicator followed by the relevant value. For example, if we wish to assign a value EM to the attribute MASS of the atom P1, we can use the function *put* as in

$$\text{put [P1;MASS;EM].} \tag{2.19}$$

A function *deflist* is also available for putting various masses, say, on different atoms, as in

$$\text{deflist [((P1 EM) (Q1 WMU) (K 0)) ; MASS].} \tag{2.20}$$

As we see in this example, the first argument of *deflist* is a list of pairs of the respective atoms and properties. A function of two arguments which is an inverse of *put* in the sense that it can be used here to recover the mass value of any four-momentum is *get*. In this case, for example,

$$\text{get[P1;MASS]} = \text{EM.} \tag{2.21}$$

Finally, we note in passing that all variables which have occurred in function definitions so far are *local* to the relevant function. It is also often necessary to carry information between functions, and for this purpose *global* variables are used. Such variables will be indicated as they occur in the remaining sections.

This concludes our summary of the LISP language. It is by no means complete, and the interested reader is referred to the cited references for more details. In particular, we have not discussed the translation between the meta-language definition of a function and its S-expression equivalent, which is the form in which the program is given to the computer. This translation process is unique and may, therefore, in fact be performed by the machine itself if suitable means for representing the necessary characters are available. In particular, our M-expression function definitions in the Appendices were produced by machine from their S-expression counterparts, and we are grateful to Mr. W. Weiher of the Stanford Artificial Intelligence Project for providing the program for doing this.

In subsequent sections, we shall adopt the following conventions. Specimens of data and material illustrating actual input and output from a program will be written in an ALGOL-like form or as S-expressions. Examples of definitions of functions and parts of the programs will be given in the meta-language.

3. Representation of Diagrams

(a) *General*

An elementary particle process is represented in Fig. 1. It shows a set of particles entering an interaction region and another set emerging. Those particles are characterized by various properties such as momenta, mass, spin, charge and so on, and there exist various laws and selection rules applying to these particles and their interactions which specify whether such a process is physically possible. For example, conservation of momentum requires that the sum of incoming momenta equal the sum of outgoing momenta.

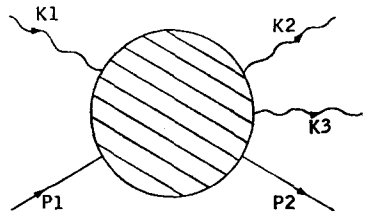

FIG. 1. An elementary particle interaction. p_1, p_2 etc. represent the four-momenta of electrons and k_1, k_2 etc. the four-momenta of photons.

It is often possible to describe such a process in terms of a set of diagrams or graphs which formally represent the development of the process with time. Various methods have been devised, principally those of Feynman [1], to compute the contribution of such diagrams to any physically measurable quantity. Feynman's technique may be summarized as a set of rules for computing physical cross-sections and related quantities directly from a set of diagrams. A typical set of diagrams for the process in Fig. 1 is shown in Fig. 2.

It is our aim in this section to show how such diagrams may be represented in a form easily manipulated by a computer and, hence, how the various structural properties of the diagram may be found. We emphasize that this description involves general principles which are applicable in many fields outside of elementary particle physics; in particular, precise analogies between the analysis of the structure of Feynman diagrams and electrical circuits have been made [22].

vertex and the second half of another. Thus, the following set of functions will compute the internal vectors of a diagram.

```
invectors[dia] = if null[dia] then NIL
    else append[caar[dia]; invectors[cdr[dia]]]
outvectors[dia] = if null[dia] then NIL
    else append[cdar[dia]; outvectors[cdr[dia]]]
allvectors[dia] = if null[dia] then NIL
    else append[caar[dia]; append[cdar[dia]; allvectors[cdr[dia]]]]
repeats[x] = if null[x] then NIL
    else if member[car[x]; cdr[x]] then cons[car[x]; repeats[cdr[x]]]
    else repeats[cdr[x]]
intvectors[dia] = repeats[allvectors[dia]].
```

Similarly, the inputs to a diagram are those vectors which occur only in the first half of a vertex and not in the second half. Thus, one definition would be

```
inputs[diagram] = setdiff[invectors[diagram]; intvectors[diagram]]
```

where

```
setdiff[u;v] = if null [v] then u
               else if member[car[u];v]then
                   setdiff[delete[car[v];u];cdr [v]]
               else setdiff [u; cdr[v]]
```

This is not the most efficient definition of *inputs*, as *invectors* is computed twice during the evaluation of *inputs*. However, as the evaluation of such properties takes negligible time, we shall not discuss improvements here.

A complete list of functions for finding such properties of a diagram is given in Appendix 1.

(b) *Particle Properties*

The various physical properties of particles in a diagram or process may be conveniently included by using the property lists of the relevant atoms as described in Section 2. For example, an ALGOL-like declaration of those atoms which represent antiparticles is made by the function *antiptl*, as in

```
ANTIPTL P4, P5;
```

The S-expression form of this statement would be

ANTIPTL ((P4 P5))

and so a suitable definition for ANTIPTL is

antiptl [u] = flag [u; ANTIPTL].

Masses of particles are included as properties of the relevant atoms with the flag MASS. For example,

MASS P1 = EM, K = 0; (3.2)

assigns a mass EM to P1 and mass zero to K. The S-expression form of this statement is

MASS (((EQUAL P1 EM) (EQUAL K 0)))

and so the function *mass* can either check for EQUAL explicitly, or assume its presence as in

```
mass [u] = if null [u] then STAR else prog2 [put [cadar[u];MASS;
                                    caddar [u]]; mass [cdr [u]]],
```

where the occurrence of the global atom STAR causes an asterisk to appear in the output as a signal that the operation performed by *mass* is complete. We note with this definition that any binary operator may be used in (3.2) in place of the equality sign. In addition, it is preferable to use symbolic representations of masses rather than numerical values (except for the obvious case of zero), as our aim at this stage is to keep all expressions exact and, thus, be aware of explicit cancellations when they occur.

The procedure employed in specifying spins of particles is a little more complicated. Particle spin may be an integer or half-integer, depending on whether the particle is a fermion or boson. A call of *spin* in infix form may, therefore, have the structure:

SPIN P1 = 1/2, Q1 = 0;

which, translated, reads

SPIN (((EQUAL P1 (QUOTIENT 1 2)) (EQUAL Q1 0)))

The function *spin* is designed to scan such a list, add the integer part of the spin to the property list of the respective atom with the flag SPIN, and also add the flag FMN to any atoms which have half-integer spin. The code for this function and other functions which add such properties to particle atoms is given in Appendix 1.

(c) *Specification of Fermion Paths*

In our particular application to physical problems, it is necessary to trace out more detailed structure of a diagram than simply a specification of inputs, internal vectors and so on. In particular, as a preliminary to computing Feynman matrix elements for a process, we need to know all fermion paths in a diagram.

Because of our orientation convention for fermions, there are only two cases which need be considered:

(i) fermion paths which terminate in the outputs of a diagram

(ii) fermion paths which are completely internal to a diagram

We begin by determining paths of the first kind. Rather than give the complete code as we go, however, we shall outline the method used and refer the reader to Appendix 1 for a complete specification.

We begin by finding all fermions in the outputs of a diagram. (These are recognized, of course, by the flag FMN on their property lists.) We then follow each of these fermions back through the diagram to its initial point, determining first the vertex the particle came from, and then the appropriate continuation of the fermion line. This procedure is then repeated until the line terminates.

Next, to compute those fermion paths completely internal to the diagram, we take all those fermions encountered in the first pass and subtract them from all fermions in the diagram. The residual fermions must constitute closed loops (unless an error has been made in specifying the diagram), and we can, therefore, start with one of these fermions and trace a closed path back from that. This procedure can then be repeated until all fermions in the residual lists are exhausted. The relevant function which determines all these paths is *fpaths*, whose definition, together with the associated functions used in its calculation, is given in Appendix 1.

Finally, as preparation for the computation of matrix elements, we need to recognize the remaining lines and vertices in the diagram. This is computed by the function *bospart* using the information found by *fpaths*. Alternatively, the complete specification of the physical structure is found using *allpaths*, which computes a dotted pair of the fermion paths in the diagram and the remainder.

(d) *Conservation of Momentum*

Another physical property of Feynman diagrams which we need to consider is the law of conservation of momentum. This law requires that the sum of all incoming momenta be equal to the sum of all outgoing momenta, for each vertex, and also for the overall process. Thus, the law specifies that one external vector may be replaced in terms of all other external vectors and, furthermore, that if there are no closed loops in a diagram, then all internal vectors may be written in terms of external vectors.

If closed loops are present in a diagram, then the replacement process is not so straightforward. The standard method, which we follow here, is to associate a *loop momentum* with each independent closed loop. This loop momentum will be in fact one of the momenta bounding the relevant loop.

The procedure we use in applying the law of conservation of momentum is to develop an association list expressing replacements for all internal vectors of a diagram, and for one external vector. The function which performs this procedure is *momrep*, and again we specify the procedure followed in words, giving the full code for the calculation in Appendix 1.

We divide all vectors in a diagram into two classes; 'solved' and 'unsolved'. Initially, the solved vectors are the independent external vectors plus one external vector expressed in terms of the others, and the unsolved vectors are the internal vectors. Each internal vector is then examined to see if it can be expressed completely in terms of solved vectors at either its incoming or outgoing vertex. If it can, it is transferred to the solved vector list. After all unsolved vectors are examined, a check is made to see if any vectors were 'solved' in that pass. If so, a new search of the unsolved vectors begins. If not, the system is recognized insoluble, and a message

*** LOOP VARIABLE ⟨variable⟩ NECESSARY

is printed, ⟨variable⟩ being the first member of the unsolved vector list, which is then transferred to the solved list. This procedure is continued until all vectors are solved.

In some cases it is desirable to specify the loop vectors in advance. The declaration *loopvar* defined in Appendix 1 adds its argument, a four-momentum to a global variable LPVAR, and then transfers it to the solved vector list at the beginning of the application of *momrep*. For example,

LOOPVAR QA;

adds QA to the solved vector list.

4. Generation of Feynman Diagrams

It is convenient to regard the final quantum-mechanical state of a system of interacting elementary particles as obtainable from the initial state by the application of an operator, the S matrix, which describes the interaction of the particles. In terms of a Hamiltonian H(x) of interaction which is a function of the space-time variable x and its time component x^0, and in natural units, the S matrix is

$$S = \sum_{n=0}^{\infty} \frac{(-i)^n}{n!} \int_{-\infty}^{\infty} d^4x_1 \int_{-\infty}^{\infty} d^4x_2 \cdots \int_{-\infty}^{\infty} d^4x_n \, P[H(x_1)\, H(x_2) \cdots H(x_n)], \tag{4.1}$$

where P is the chronological ordering operator which rearranges the n terms in its argument into a form

$$H(x_l)H(x_j)H(x_k) \ldots \text{ such that } x_l^0 > x_j^0 > x_k^0 \ldots\ldots$$

Since H(x) is proportional to the coupling constant for the interaction, each term in the sum in (4.1) describes the transition between initial and final states by all possible Feynman diagrams with n vertices. H(x) is a product which contains as many interacting fields as there are particles that are permitted to interact at any vertex. Moreover, because the interpretation of a field quantity as an operator identifies parts which are used as creation and annihilation operators for individual particles, it is possible to trace out the history of any particle within the set of vertices of any Feynman diagram. Details of the formalism can be found in standard texts ([2], [23]).

The practical problem is the reduction of the product of operators in (4.1) to a form which makes identification of all of the distinct Feynman diagrams (for the purposes of calculation of matrix elements in momentum space) of a given order as easy as possible. Here, the reduction is done in three stages. First, we write Wick's chronological ordering operator T, which is equal to $P(\pi)$ except for a multiplying factor that is 1 if the permutation of fermion fields that is necessary to turn the product π into $P(\pi)$ is even and -1 if the permutation is odd. Hence, P in (4.1) can ultimately be expressed in terms of T. Second, we introduce the contraction $A^{\cdot}B^{\cdot}$ of two factors and the normal product N according to

$$T(AB) = A^{\cdot}B^{\cdot} + N(AB)$$

The particular distinction of N is that it causes all creation operators in its argument to fall to the left of all annihilation operators. Finally, we use Wick's theorem [24], which tells us that a T product can be decomposed uniquely into a sum of normal products as

$$T[N(A_1B_1C_1 \ldots X_1Y_1Z_1)N(A_2B_2C_2 \ldots X_2Y_2Z_2)] = f_1f_2 ,$$

where

$$\begin{aligned} f_1 = {} & N(A_1B_1C_1 \ldots X_1Y_1Z_1) + N(A_1^{\cdot}B_1^{\cdot}C_1 \ldots X_1Y_1Z_1) \\ & + N(A_1^{\cdot}B_1^{\cdot\cdot}C_1 \ldots X_1Y_1^{\cdot\cdot}Z_1^{\cdot}) \ldots\ldots\ldots + N(A_1^{\cdot}B_1^{\cdot\cdot}C_1^{\cdot\cdot\cdot} \ldots X_1^{\cdot}Y_1^{\cdot\cdot}Z_1^{\cdot\cdot\cdot}) \end{aligned} \tag{4.2}$$

and each possible combination of pairwise contracted factors occurs once in the sum of normal products above.

If a contraction in (4.2) is made between a creation operator function of x_l and an annihilation operator function of x_j, where $x_j^0 > x_l^0$, it may be understood to represent a line for the appropriate particle beginning at vertex i and ending at the

later vertex j of a Feynman diagram. Uncontracted creation operators in a normal product refer to particle lines that leave the corresponding diagram, and lines that enter the diagram can be associated with uncontracted annihilation operators. Equation (4.2), which is discussed in some detail by Bogoliubov and Shirkov [25], is the basis of a short subprogram to derive Feynman diagrams in the representation (3.1). The association just described between field operators in (4.2) and lines in Feynman diagrams relies implicitly on the substitution of (4.2) into (4.1) and the relation of each term in the resulting expanded form of S to a single Feynman diagram.

The program is designed to generate only diagrams for one order of perturbation theory, or value of n, at a time. The programmer declares this order by the command

VERTICES *n*;

where n is a positive integer. The equivalent LISP function is:

vertices[n] = prog2[ord := n; skn := skp[n;1]],

where

skp[n;m] = *if* greaterp[m;n]*then* NIL
else cons[list[m;NIL;NIL];skp[n;addl[m]]]

The global variable ORD is carried through the calculation for reference, and SKN represents the skeleton of a Feynman diagram with vertices numbered. Initially, as set up by *skp*, it has the structure ((1 NIL NIL) (2 NIL NIL).....). Lines entering any vertex are recorded on a list which is the second member of the relevant sublist, and lines leaving the vertex are listed in the third position of the sublist. Later, when the calculation of a diagram is complete, the vertex numbers are discarded and the rest of each sublist is changed into a form equivalent to that of (3.1).

Free use of (4.2) generates three classes of Feynman diagrams, only one of which is usually needed. The unused diagrams represent either vacuum fluctuations (and therefore have no external lines) or two or more disjoint processes with a total of n vertices. Output of these diagrams is suppressed by the initial commands

VACUUM NO;

and

DISJOINT NO;

which effectively close off the printing branch of the program to vacuum fluctuations and disjoint processes. The functions *vacuum* and *disjoint* are listed in Appendix 2. On the rare occasions when either of these types of diagram is wanted, the

appropriate command is used with any other argument that signifies approval (e.g., YES).

In many situations, the user of the program may wish to have particular labels for the four-momenta of the ingoing and outgoing particles that are present in the diagrams which are generated. Two declarations are provided for that purpose: INITIAL for the inputs and FINAL for the outputs. Use of these declarations is optional, but it is compulsory in all applications of the diagram-generating program to make the initializing statement

INOUT ;

which sets up certain internal lists. At present the programmer may choose labels for fermion lines from the atoms P1, P2 ... and labels for boson lines from the atoms K1, K2 .... Then, after the INOUT declaration above, suppose that we select P1 and K1 as labels for two inputs to our diagrams, and P2 and K2 as outputs. The appropriate statements to establish this selection are:

INITIAL P1,K1;

and

FINAL P2,K2;

The atoms P1, P2, K1 and K2 are thus reserved for external lines of the generated diagrams.

The final command, which starts the actual calculation, makes use of the function *interaction*. Its argument is a list of the factors that occur in the chosen expression for H(x), with a few arbitrary conventions. The interacting fields are the sole quantities of interest in the construction of Feynman diagrams, so that their important attributes must be represented in some detail. Other quantities, like coupling constants and Dirac matrices, are unimportant here. We make the convention that the unimportant quantities should be given as atoms in the argument of *interaction*, and the interacting fields by lists of their important attributes. For the simple cases so far considered, these attributes identify creation operators and spins or statistics. We, therefore, represent a field by [a;b] where a is 1 if the field operator creates a fermion and 0, otherwise, and b is 1 for a fermion field and 0 for a boson field. Hence, for the often-used example of a local vector interaction between an electron field Ψ and a photon field A_μ according to a relation like $H(x)=e\overline{\Psi}(x)\gamma_\mu\Psi(x)A_\mu(x)$, a LISP representation of the argument for *interaction* is

(E (1 1) GAMMA (0 1) (0 0)). (4.3)

At present the atomic members of (4.3) have mnemonic value only, because the first step in *interaction* is a call to a function *nonatom* which simply removes all

atoms from the argument. However, the convention can be altered in the future if the program is expanded to perform more sophisticated operations than just the generation of the structure of a Feynman diagram.

Next, a label from the list (L1 L2 .....) is associated with each field or particle in the argument of *interaction*, and the function *dinit* is called to produce a list of dotted pairs of particle labels and vertex numbers. For example, if the argument is as given in (4.3) and the order n is 3, the list is

$$((L1 . 1)(L2 . 1)(L3 . 1)(L1 . 2)(L2 . 2)(L3 . 2)(L1 . 3)\\ (L2 . 3)(L3 . 3)). \qquad (4.4)$$

The process of contraction in (4.2) collects all possible pairings of elements of (4.4). This operation is controlled by the important function *contract*, of which a simplified definition is

```
contract[y]=prog[[a;b;c;d];c:= cpairs[y];G;a:=comb[a];
                *if* null[a]*then* return[NIL];d:=items[a;c];
                b:=setdiff[y;roll[d]];generate[cons[d;b]];go[G]],
```

where

```
roll[u]=if null[u]then NIL else cons[caar[u];cons[cadar[u];
                                                roll[cdr[u]]]],
items[u;v]=if null[u]then NIL
        else cons[item[car[u];v];items[cdr[u];v]]
```

and item[n;u]=*if* onep[n]*then* car[u]*else* item[sub1[n];cdr[u]].

The function *cpairs* produces the physically possible pairings from (4.4). Each pairing that is combinatorially possible is examined by the subsidiary function *crule*, which discards the large number of pairs that have the interpretation that a particle changes its spin, starts and terminates at the same vertex, or is not created at one vertex and annihilated at a later vertex (i.e., on one with a higher vertex number). Each of the remaining pairings is then selected in turn by the function *comb*. The set-theoretic difference of a structure like (4.4) and the elements of each pairing is regarded as a list from which the external lines of one Feynman diagram are derived. Finally, *contract* calls *generate* with this information, and the latter function uses it to build the Feynman diagram upon the structure of SKN. Line labels are selected from the lists (P1 P2 P3.....) for fermions and (K1 K2 K3.....) for bosons. The actual addition of new lines to the diagram is performed by *add*, which is called by *generate*. The last step of the latter function is to print each diagram when it is formed.

The discussion of the basic algorithm is now complete. Other functions which occur in Appendix 2 but are not described here are not essential to the discussion, and merely carry out various types of bookkeeping.

The program for the generation of diagrams is not listed in Appendix 1 along with the regular programs because it does not fit into them smoothly. In particular, while it produces diagrams in the style of (3.1), it does not assign attributes like mass and polarization to momentum atoms, and it does nothing about the assignment of rules for the treatment of vertices by later programs. There is no present reason why the programs are not joined smoothly except that physicists who have used them have tended to think in terms of Feynman diagrams rather than Hamiltonians of interaction for the initial statement of a problem. When the order n of a problem is high enough, though, this reason must disappear, because of the difficulty of writing down all the diagrams by intuition and without error. In that case, the result of development of the diagram-generating program will probably depend on the most popular problems which require solution.

5. Computation of Matrix Elements

(a) *Preliminary*

The application of the rules developed by Feynman [1] to determine physical matrix elements from a given diagram is a good example of unique mapping of one data space onto another. The rules which determine this mapping are easy to specify, and the result of the mapping depends only on the properties of the particles and vertices in a diagram. There are two problems to be considered here; first, the specification of the rules themselves and their resulting storage and, secondly, the application of the rules to a given diagram.

(b) *Specification of the Feynman Rules*

A convenient summary of the Feynman rules is given by Bjorken and Drell [2]. In specifying these rules, and in the subsequent development, we shall use the conventions and metric of Bjorken and Drell throughout. Thus, the four-momentum of a particle is defined by

$$p^{\mu} = (E, p_x, p_y, p_z)$$

and the scalar product of two four-momenta is

$$p_1 \cdot p_2 = E_1 E_2 - p_1 \cdot p_2$$

The Dirac gamma matrices which we employ are defined in general by the equation

$$\gamma_\mu \gamma_\nu + \gamma_\nu \gamma_\mu = 2g_{\mu\nu} \tag{5.1}$$

which defines a set of hypercomplex numbers γ in a space with metric tensor $g_{\mu\nu}$. In practice, we limit ourselves to a four-dimensional matrix representation of these quantities defined in a space with metric

$$g_{00} = -g_{11} = -g_{22} = -g_{33} = 1 \text{ and } g_{\mu\nu} = 0 \text{ for } \mu \neq \nu$$

We also need to specify symbols to represent the various special operators which are encountered in these rules. The operators we introduce are as follows:

	scalar product of two four-vectors
G	gamma matrix operator
EPS	completely antisymmetric tensor of order 4
PROPR	denominator of propagator
U	fermion spinor
UB	conjugate of fermion spinor
V	fermion spinor for an antiparticle
VB	conjugate of fermion spinor for an antiparticle.

A description of these operators is as follows:

(i)

The operator is a binary operator used to denote the scalar product of two Lorentz four-vectors. Its arguments are vector expressions, e.g.,

$$\text{P.Q , (P— P.K*K/K.K).(R + S).}$$

As we mentioned earlier, components of vectors in diagrams are not explicitly displayed but, if ever needed, they may be indicated with the inclusion of representations of unit vectors in the system. Thus, if E0 represents the unit vector (1,0,0,0), P·E0 represents P^0, the zero-th component of the four-vector, P. Similarly, an arbitrary component P^μ may be represented by P·MU. When contraction over an index is required, the system has to be informed that MU, for example, represents an index rather than a four-vector. To achieve this, indices are added to a global variable INDICES, either explicitly by the system, or indirectly by the user with the function

index[u] = indices := append[u;INDICES]

Thus

INDEX MU, NU;

would inform the system that MU and NU are indices to be contracted in expressions involving them. The index status of atoms may be removed by the user with the function

remind[u] = indices := setdiff[INDICES;u]

(ii) *G*

The operator G is an n-ary operator used to denote a product of gamma matrices contracted with Lorentz four-vectors. To allow for the occurrence of more than one fermion line in a diagram, the first argument of G is a nonnumerical line-identification variable used to distinguish different lines. Thus, G(L1,P)*G(L2,Q) denotes the product of $\gamma\cdot P$ or $\not{P}$ associated with a fermion line identified as L1, and $\not{Q}$ associated with another line identified as L2. A product of gamma matrices associated with the same line may be written in a condensed form, thus:

$$G(L1,P1,P2+P3,P4) \equiv G(L,P1)*G(L1,P2+P3)*G(L1,P4)$$

The vector A is reserved in arguments of G to denote the special gamma matrix γ_5. Thus,

$$G(L,A) = \gamma_5 \text{ associated with the line L}$$

and $G(L,P,A) = \not{P} * \gamma_5$ associated with the line L. The general matrix γ_μ associated with a line L may be written as G(L,MU) with MU declared as an index if contraction over μ is required.

(iii) *EPS*

The operator EPS has four arguments, and is used only to represent the completely antisymmetric tensor of order 4 and its contraction with Lorentz four-vectors. Thus,

$\epsilon_{\mu\nu\rho\sigma} = +1$ if μ,ν,ρ,σ is an even permutation of 0,1,2,3.

-1 in the case of an odd permutation

0 otherwise

A contraction of the form $\epsilon_{\mu\nu\rho\sigma}P_\rho q_\sigma$ may be written as EPS(MU, NU, P,Q) with MU and NU declared as indices, and so on.

(iv) *PROPR*

Whenever an internal particle of momentum p and mass m appears in a diagram, there is a propagator denominator $(p\cdot p - m^2)$ associated with it in the corresponding matrix element. To avoid proliferation of rational functions, and for easier recog-

nition of matrix element structure, we represent such terms by the binary operator PROPR. In infix notation, this is

$$\text{PROPR(P,M)} = 1/(\text{P·P}-\text{M**2}) \tag{5.2}$$

(v) *U,UB,V and VB*

These operators are used to represent spinors and their conjugates for particles and antiparticles, respectively. They have two arguments, the first of which is a line-identification variable, and the second of which is either a vector variable or minus a vector variable. e.g., U(L,P) or VB(L2,−Q1).

The rules which are necessary for the complete specification of matrix elements may be divided into three classes; those which apply to lines in a diagram, those which apply to vertices, and those which apply to the general structure of the diagram.

To specify a Feynman rule for an arbitrary line requires information on the type of particle, and also whether it is internal or external to the diagram. Except for the ambiguity between the photon and the neutrino, it is sufficient to identify a particle by its status (external or internal) and mass to specify a rule uniquely. To include the photon and neutrino in this scheme, we given them the symbolic masses WLA and WNU, respectively, and make adjustments in the matrix routines so that these symbols are replaced by zero when they occur in expressions, unless a nonzero value is needed temporarily [12] as an infrared cutoff.

A general rule also contains information on the momentum, polarization vectors and indices of the particle, which are not known at the time the rule is specified. To allow for this, we use the atom MOM to represent the (dummy) momentum of the particle, the index atoms IN1, IN2...to represent indices, and POL1, POL2...to represent polarization vectors.

In specifying rules for lines involving gamma matrices and spinors, we also use the dummy line identification symbol LN as the first argument of the relevant operators.

We must also establish standard names for at least the common particles, whose Feynman rules will be defined initially in the system (but can be changed by the user if he so desires). The following conventions will therefore be used in the remainder of the paper and the system itself.

Particle	*Mass*
electron	EM
proton	PM
neutron	WNU
pion	PIM
muon	WMU

The artifices used above to avoid the apparently more natural labelling of mass with M as the first character are to provide compatibility with FORTRAN programs, where numerical values for masses are floating-point quantities.

A rule for a line in a diagram is, thus, specified in the form:

L(⟨mass symbol⟩, ⟨status⟩) = ⟨expression⟩,

where

⟨status⟩ : := EXT | INT

These rules are added to the system by the function *frule*, which we shall discuss later. The rules for the common particles which are already in the system are introduced as follows:

```
FRULE L(EM, INT) = I*(G(LN,MOM)+EM)*PROPR(MOM,EM),
      L(EM, EXT) = U(LN,MOM),
      L(PM, INT) = I*(G(LN,MOM)+PM)*PROPR(MOM,PM),
      L(PM, EXT) = U(LN,MOM),
      L(WLA, INT) = −IN1 . IN2 * PROPR(MOM,WLA),
      L(WLA, EXT) = IN1·POL1,
      L(WMU, INT) = I*(G(LN,MOM)+WMU)*PROPR(MOM,WMU),
      L(WMU, EXT) = U(LN, MOM),
      L(PIM, INT) = PROPR(MOM, PIM),
      L(PIM, EXT) = 1,
      L(WNU, INT) = I * G(LN,MOM) * PROPR(MOM, WNU),
      L(WNU, EXT) = U(LN, MOM);
```

A specification of a rule for a vertex can be made in a similar way. Again, knowledge of the masses of the particles at the vertex is sufficient for unique specification of the rule. However, we now need to associate a momentum with each particle and give it a direction. We, therefore, use the dummy labels MOM1, MOM2, MOM3,.... for this purpose, associating MOM1 with the first mass in the vertex list, and so on. The association is made unique by the requirement that the direction of each momentum be *into* the vertex. In a like manner, variables which identify fermion lines will have names LN1, LN2,... and so on. In both these cases, it is only necessary to associate at most one symbol with each particle. However, in the case of spin indices, a given particle may have more than one index, and, therefore, indices IN11, IN12, IN13, ... are associated with the first mass on the vertex list, IN21, IN22, ... with the second mass, and so on. Vertex

The third type of Feynman rule to be considered defines contributions from explicit structural properties of a diagram. Such rules include:

(i) A factor of -1 for each closed fermion loop;

(ii) A factor of $\frac{1}{2}$ for each closed loop containing only two photon lines;

(iii) A factor of $\frac{1}{2}$ for each closed loop containing only two meson lines.

These rules require no *a priori* format specification, and are therefore included during the explicit calculation of a matrix element which we discuss in the next section.

(c) *Computation of a Matrix Element from a Diagram*

Having now defined a format for storing arbitrary rules for Feynman diagrams in our system, it is a relatively straightforward matter to compute the Feynman matrix element for any given diagram. This quantity is computed by the function *matrix*, which takes a single diagram as argument and is defined with its associated functions in Appendix 1. We shall assume in this discussion that all necessary properties of the particles involved (masses, spins, etc.) are defined. If they are not, then this will be detected during the calculation of the matrix element, and an error message generated. In this case, *matrix* will return NIL.

The function *matrix* first computes the fermion paths of the diagram (using *inpaths* and *rempaths*) which are then passed to *mat1*. The function *mat1* gives each fermion path to *mat2* which scans each term in the path, and looks up the relevant line or vertex rule for each element encountered in the list. If no rule is found, an error message is printed, and *mat1* returns NIL. Having found the rule, it is then necessary to rename the momenta, indices, polarization vectors and fermion line variables in the rule by the current values associated with the relevant particles.

A standard nomenclature is also used here. Indices for use in explicit matrix elements are given names I0, I1, I2... etc., polarization vectors E0, E1...and fermion line variables L0, L1.. Indices associated with a given momentum variable are stored on the association list INDLIS, and polarization vectors on PLIS for later reference. In addition, all indices and polarization vectors used in an expression are added to the global variables INDICES and POLVTRS, respectively. The function *mat2* then returns the matrix element for the whole fermion line to *mat1*, which renames the fermion line variables in this expression, and then repeats the procedure for each subsequent fermion path. Finally, *mat1* adds a factor of -1 for each internal fermion loop found to the list of matrix elements from all paths and returns the expression to *matrix*. The latter function now computes the boson part of the diagram, whose matrix element is then found by *mat2* and added to the term produced by *mat1*. The complete matrix element is next scanned by the function

ascan which removes all numbers and variables which are simple multiplicative factors in the expression (such as coupling constants and factors of I) and places them at the front of the expression. A final check for possible factors of $\frac{1}{2}$ from internal photon or meson loops completes the computation of the matrix element.

6. Dirac Algebra

In the previous section, the Dirac matrices were introduced through the defining relation (5.1). From (5.1), there follows the property that any product of Dirac matrices may be reduced to a sum of 16 terms. One of these terms contains the 4×4 unit matrix, four contain single Dirac matrices (i.e., γ_0, γ_1, γ_2, and γ_3), six contain factors of $\gamma_\mu\gamma_\nu$ where $\mu \neq \nu$, one depends only on $\gamma_5 = \gamma_0\gamma_1\gamma_2\gamma_3$, and four contain $\gamma_5\gamma_\mu$.

Programs which perform large-scale simplifications of expressions do not, in general, gain in speed or efficiency if arbitrary products of Dirac matrices are reduced to sums of 16 terms in advance of any other operations. In the present situation, however, there may be occasions when this facility is of value. For example, if the complete result of a calculation is to be the sum of several products, and substantial cancellations are expected between products, the expression of each product as a sum of 16 terms allows rapid inspection for these cancellations. Alternatively, our programs automatically derive Dirac-matrix products sandwiched between spinors and adjoint spinors, and useful simplification of these expressions may be available if one anticommutes factors like $\gamma \cdot \mathrm{p}$ through the product structure by means of the relation

$$\gamma_\mu\gamma\cdot\mathrm{p} + \gamma\cdot\mathrm{p}\gamma_\mu = 2\mathrm{p}_\mu \tag{6.1}$$

until they meet a spinor function of p. When that happens, simplification is made with the help of the Dirac equation

$$\gamma\cdot\mathrm{pu(p)} = \mathrm{mu(p)}\,,\ \bar{\mathrm{u}}(\mathrm{p})\gamma\cdot\mathrm{p} = \mathrm{m}\bar{\mathrm{u}}(\mathrm{p}) \tag{6.2}$$

Both of these types of simplification rely on the ability of the program to rearrange factors in products by anticommutation via (6.1). We have implemented two distinct methods of anticommutation, which we describe separately below.

First, in a large arbitrary product, we may identify all the occurrences of special four-momenta like p_2 for which we wish to anticommute factors of $\gamma\cdot\mathrm{p}_2$ in either direction. The functions

lcom[x]=prog[NIL;redlis := union[REDLIS;x];flag[x;LEFT];mshell[x]]
and rcom[x]=prog2[redlis := union[redlis;x];mshell[x]], (6.3)

for which x is a list of four-momenta, specify anticommutation to the left and right, respectively. If P2 occurs in the argument of *lcom*, it is placed on the list named REDLIS as an atom to be anticommuted (to the left, because of the flag LEFT on its property list) to the end of any structure in which it occurs. Where $\gamma\cdot p_2$ appears, the part of the program governed by the function *sred* effectively makes repeated applications of (6.1). This section is called if needed inside *matrix* at a stage after which all the parts of an unsimplified representation of a matrix element have been derived from a Feynman diagram by the procedures described in Section 5. The function *mshell* within (6.3) ensures that juxtapositions like $\gamma\cdot p_2\gamma\cdot p_2$ in a product are replaced by m_2^2. REDLIS is inspected by the program before any terms of a product are simplified; if REDLIS has no members, regular simplification proceeds but, otherwise, anticommutation of the type specified by (6.1) and (6.3) is performed first.

The method discussed above is effective in computations which in principle can have an arbitrary number of Dirac matrices in a product and, therefore, an arbitrary number of different ordered patterns of Dirac matrices and four-momenta. On the other hand, most calculations in perturbation theory which use Feynman diagrams do not go beyond diagrams with about eight vertices and two fermion lines (not including closed fermion loops). The maximum number of Dirac matrices obeying (5.1) and (6.1) and leading to a nontrivial result when placed between a pair of spinors in these cases is six. Therefore, it is reasonable to store representations of each distinct pattern of Dirac matrices and numerators of rationalized propagators together with the simplified general expression which results when one or more factors of the form $\gamma\cdot p$ are removed by the use of (6.2).

In the simplest case, consider the fermion line $p_2p_6p_4$ shown in Fig. 3. The

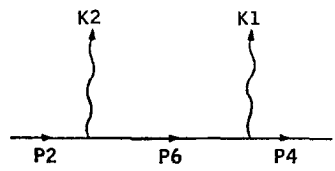

FIG. 3. Diagram illustrating the simplification under commutation of the part of an amplitude expressed by Eqs. (6.4) and (6.5).

numerator of the amplitude for the process of which Fig. 3 illustrates a part contains a factor

$$\bar{u}(p_4)\gamma_\mu(\gamma\cdot p_6 + m_6)\gamma_\nu\, u(p_2) \tag{6.4}$$

Suppose further that $k_2 = 0$, so that $p_6 = p_4$. Applying (6.1) for $p=p_6$ once, we find that (6.4) reduces to

$$2p_{4\mu}\bar{u}(p_4)\,\gamma_\nu u(p_2) \tag{6.5}$$

since $m_6 = m_4$. Where connections such as that between (6.4) and (6.5) exist, the fastest method of programmed anticommutation avoids the direct use of (6.1) and replaces unsimplified by simplified products by recognizing the patterns that products like (6.4) follow.

The LISP S-expression equivalent to (6.4) is

((UB L2 P4) (G L2 MU) (PLUS (G L2 P4) M4) (G L2 NU) (U L2 P2)), (6.6)

where L2 labels the fermion line and the operator TIMES is implicit. The form of factors between the spinors in (6.6) is adequately given by (G T G), where T is chosen as the symbol to represent the numerator of a rationalized propagator. In order to reduce (6.6) to an expression corresponding to (6.5), we need to select the four-momenta from each factor where they occur, and Greek indices from the remaining factors. Hence, (6.6) displays a structure (P4 MU P4 NU P2). The function which acts on (6.6) to produce this pattern is *pattern*[x]. Given that the various types of factor which can occur are well represented in (6.6), and that their syntax is constant, *pattern* contains in essence only chains of operations of the *car-cdr* type following tests with the predicate *equal.*

In fact, the value of *pattern* when x stands for (6.6) is

(P4 MU P4 NU P2 NIL NIL) (6.7)

The extra entries of NIL are to facilitate the use of the function *apair*, which in this case detects the factor $p_{4\mu}$ for inclusion in (6.5) and prepares for cancellation from (6.6) the factors which do not appear in an S-expression representation of (6.5). This function, which is applied next in the overall algorithm, has as its result the dotted pair whose *car* is a list of identifiers for the terms to be cancelled from (6.6) and whose *cdr* is a list of factors like $p_{4\mu}$. As given in Appendix 1, *apair* has five arguments; the first stands for the expression like (6.7) that is to be processed, and the remaining arguments are merely for bookkeeping of partial results (since the function is defined recursively). If the first argument of *apair* is actually the expression (6.7), the value of the function is ((3 2) (DOT P4 MU)). DOT is the LISP

question of how one programs recognition of the correct function of $p_{4\mu}$ (in this case just $2p_{4\mu}$) which reproduces (6.5) exactly. To do this, we return to the pattern (G T G) of types of factors which occur between the spinors in (6.6). During the analysis of (6.6) by *pattern*, this structure is labelled by the atom ACOMLIST. We can now make use of ACOMLIST by setting up as part of the program a table with the form $[a_1;b_1;a_2;b_2....]$, where each a_i represents a pattern and the corresponding b_i gives the general multiplying factor which accompanies that pattern.

ACOMLIST is compared with each a_1 in turn until a correspondence is found. For example, if

a_1 = (G T G), then b_1 = (TIMES 2 (DOT MOM IN1)) or 2*(MOM1 · IN1)

in infix notation. The table is a list called ARULE and in the program it is searched by a function *asearch*. ARULE can be built up by the programmer with a function *atypes* which works on the same principle as the rule-assigning function *frule* described in Section 5.

It is now possible to describe in simplified form the function *acommute* which implements this method of anticommutation by recognition of patterns and calls the other functions discussed above. A definition is:

```
acommute[x]=prog[[a;b;c]; a:=pattern[x];b:=apair[a;NIL;1;length[x];NIL];
                          c:=car[b];b:=cdr[b];a:=dotsubs[b;asearch[acomlist;arule]];
                          return[append[a;acancel[x;c;1]]]]                    (6.8)
```

In (6.8), the purpose of the function *dotsubs* is to convert the dummy expression (DOT MOM IN1) to (DOT P4 MU) in the case of the example given. Also, since we have observed that the value of *apair* during the processing of the example is ((3 2) (DOT P4 MU)), then the program variable c in (6.8) in this particular situation takes the value (3 2). Hence, its presence as an argument of the final function *acancel* may be understood; *acancel* removes from x the second and third elements of that list, so that the output of *acommute* corresponds with (6.5). The purpose of *acancel* in general is just this cancellation of unwanted factors.

The programming methods discussed in this section are adequate to handle problems involving Dirac matrices as the basic non-commutative items of data. Where it is desirable to use other basic entities (e.g., in computation of the magnetic moment of the electron [25], where $F_{\mu\nu}$ is the electromagnetic stress tensor and $F = \frac{1}{2} i\gamma_\mu\gamma_\nu F_{\mu\nu}$ is needed frequently) having anticommutation properties other than (5.1) or (6.1), the programs can be expanded fairly readily to accept the new data, although we have not systematized that procedure and do not present it in Appendix 1. It is only necessary to enter information about new patterns of factors on ARULE, and extend the definition of *pattern* by one new test using *equal* to take account of the behaviour of each new type of data under anticommutation.

The two methods of Dirac algebraic manipulation carried out by *sred* and *acommute* are not exact alternatives; one can choose to use either method, both, or neither. For the last choice, no special declaration is needed. Otherwise, the function or functions required are declared in the initial stages of a program. For example, if both are required, the programmer makes the declaration

ALGEBRA SRED, ACOMMUTE;

The declarative function *algebra* is included in Appendix 1.

7. Recognition of Integrals

If it is possible to trace out at least one continuous closed path purely on internal lines of a given Feynman diagram, the matrix element derived from that diagram will contain at least one nontrivial four-dimensional integral over an internal four-momentum. A second type of integral that occurs in calculation of measurable quantities is defined over phase space or momentum space for particles produced in a process but not detected. This latter type, however, is encountered during the simplification and reduction of the absolute square of a matrix element. Since the aim of the programs discussed here is the production of an unsimplified absolute square, phase-space integrals fall outside the scope of this paper. We remark only that the problems associated with each type of integral are the same, and that the result of a calculation involving extensive use of phase-space integration by table look-up within the general simplifying program REDUCE [26] is reported elsewhere [27]. In the rest of this section, we review methods for the recognition of integrals that occur in the derivation of matrix elements from Feynman diagrams.

A typical integral arises from the diagram in Fig. 4. Suppose that it illustrates the

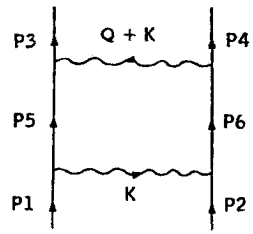

Fig. 4. Example of a Feynman diagram which generates integrals including that in Eq. (7.1).

scattering of an electron ($p_1p_5p_3$) from a proton (p_2 and p_4) which is temporarily excited to a nucleon resonance p_6 of mass r or RESM unequal to the proton mass. To avoid complications that are unnecessary here, we assume that the spin of the resonance is $\frac{1}{2}$. Figure 4 raises some highly characteristic problems even though it is the simplest Feynman diagram with a closed internal loop that does not merely signify some kind of renormalization. First, for electron-proton scattering, we can make a reasonable estimate of vertex rules at vertices on the proton line only if the resonant object is not too far off its mass shell. This situation generally persists for quite arbitrary scattering processes represented by Fig. 4. Secondly, the electromagnetic form factors F1 and F2 introduced in Section 5b are not known for all values of their argument, but it appears that F1 falls off rapidly from a maximum at an argument of zero. So it is usually assumed ([12], [28]) that the integral from

Fig. 4 is a sum of two contributions, one when the total four-momentum transferred in the scattering is q as shown, and one when the assignments of $q + k$ and k to the photon lines are reversed. For the first case, $k \approx 0$ and $q+k \approx q$. The corresponding unsimplified integral, apart from factors at vertices, is

$$\frac{1}{q^2}\int \frac{d^4k}{k^2-\lambda^2}\,\frac{\gamma\cdot p_5+m}{p_5{}^2-m^2}\,\frac{\gamma\cdot p_6+r}{p_6{}^2-r^2}. \tag{7.1}$$

To enable the factoring of $1/q^2$ in (7.1) to be carried out, we use the declarative function *xfervar* to show that q is both fixed and the measure of momentum transfer. Similarly, we distinguish k as the loop variable or variable of integration with *loopvar*. The programmer in this instance makes the declarations

LOOPVAR K;

and

XFERVAR Q;

in advance of computation based on Fig. 4.

The S-expression form of the integrand in (7.1) is

((PLUS (G L1 P5) EM)(PLUS (G L2 P6) RESM)(PROPR K WLA)
(PROPR P5 EM)(PROPR P6 RESM)) (7.2)

with the overall operation TIMES implicit. From (7.2) it is not clear how P5 and P6 depend on the loop variable K, so that substitutions are required as the first stage of the calculation. The function *momrep* which we have met already in Section 3d is used to arrive at the correct substitutions. In fact, it is sufficient to let $p_5 \approx p_1$ and $p_6 \approx p_2$ in the present example, but no approximations can be made for substitutions into the PROPR factors in (7.2) without drastically affecting the result. For example, $p_5=p_1-k$ implies that $p_5{}^2-m^2 = k^2-2p_1\cdot k$, which is totally different from the value of zero which would follow from the approximation. Also, $p_6=p_2+k$ implies that $p_6{}^2-r^2=k^2+2p_2\cdot k+M^2-r^2$, which is different in form from the denominator term for the electron. Both of these terms now carry more information than can be accommodated in the simple PROPR representation (5.2). Hence, the program uses a new and purely internal representation which is a list [DENOM;n;b;c], where b is the four-vector quantity whose scalar or dot product with $-2k$ occurs in the term under consideration, c stands for the constant part of the term that is built only out of scalars, and n is the highest degree of k in any part of the term. The latter item of information is not used directly in the programs in Appendix 1, but has been helpful in heuristic programs (discussed in Section 9) that analyze asymptotic behaviour of matrix elements.

corresponding to an integral result b_I. The table is prepared or updated by a function *itypes*:

$$\text{itypes[x]}=\text{ityp:}=\textit{if}\ \text{null[ityp]}\textit{then}\ \text{x}\ \textit{else}\ \text{append[x;ityp]}. \tag{7.4}$$

Therefore, although the table contains initial entries, the programmer can add new entries of his own choosing. For instance, if the result of the integration whose coded type according to (7.3) is (2 2 2) is a function $h(\lambda)$ of the photon mass, this result is added to ITYP by the declaration

$$\text{ITYPES (2 2 2), H(WLA);} \tag{7.5}$$

The familiar infix notation may be used to express results b_I, but a_I must be in the list form shown in (7.5). Arbitrary dummy arguments may be used for mnemonic convenience in b_I, but the program actually replaces them by ID1, ID2.... from a list IDLIST for ease in later computation.

The next stage in the procedure is the examination of a representation like (7.2) for contributions to an integral. This task is performed by a function *idenom*, which not only calculates the identifier for each factor in the denominator but also picks up the quantities like λ^2, p_1 and M^2-r^2 in (7.2) in terms of which the result of integration is expressed. As we know how many of these quantities there will be for a given integral, the representation b_I of the integral in ITYP will contain that number of dummy arguments. The value of *idenom* is not just a list of identifiers for terms, but a list in which each identifier is followed by any new potential arguments for eventual substitution in b_I that have been found. When *idenom* is given an integral of type (2 3 4) as an argument, the general result is:

$$\text{(2 (ID1 MINUS (EXPT WLA 2)) 3 (ID2 . P) 4 (MINUS B))} \tag{7.6}$$

For substitutions to be made in the correct places in b_I, forms like (7.6) are expressed by *idenom* so that the numerical identifiers always occur in ascending order. As may be seen from (7.6), each argument is *cdr* of a dotted pair, where *car* is a dummy label extracted from the list IDLIST which the program keeps for that purpose.

Last, the sequence of identifiers, like (2 3 4) above, is recovered from (7.6) by a function *isearch* that also searches for an entry a_I on ITYP which is equal to that sequence. The final stage of calculation is the substitution in b_I of arguments like those in (7.6) by means of the LISP function *sublis*. The entire procedure described in this section is administered by a function *intop*, whose short definition is:

$$\begin{aligned}\text{intop[x]}=\text{prog[[u;v];u:=idenom[iconvert[x]];v:=isearch[u;ityp];}\\ \text{return[sublis[nonatom[u];v]]].}\end{aligned} \tag{7.7}$$

The definition of *nonatom*, which removes all atoms from its argument, is found in Appendix 2.

The methods described so far are adequate for the integral (7.1), where no explicit k dependence remains in the numerator because of the physical approximations in the problem. Nevertheless, other problems can generate integrals of the type

$$\int \frac{k_\mu d^4k}{(k^2 - 2q \cdot k - \Delta)^n} \tag{7.8}$$

or

$$\int \frac{k_\mu k_\nu d^4k}{(k^2 - 2q \cdot k - \Delta)^n} \tag{7.9}$$

However, the parameters q and Δ in an exact calculation are functions of further variables of one-dimensional integration such as x in the Feynman integral identity

$$(AB)^{-1} = \int_0^1 [Ax + B(1 - x)]^{-2}\, dx$$

and its generalizations to n factors. Thus, even though (7.8) reduces to

$$q_\mu \int \frac{d^4k}{(k^2 - q^2 - \Delta)^n} \tag{7.10}$$

and the integral over k may be recorded on ITYP, (7.10) then requires further one-dimensional integrations over Feynman parameters. This is actually the way the analytic expression for the integral over k in (7.10) is calculated, but since q_μ in the numerator is also a function of the Feynman parameters, the final result may not be a single analytic form. It has been found that the most practical method here is to regard the result as a new four-vector, and to assign it a name PINT1, PINT2, ... from a list PINTLIST. This name may then appear in the output. The function *idenom* collects potential arguments of the new object as before, but in this case they are stored on a list PRESLIST as *cdr* of a dotted pair whose *car* is the name of the four-vector. The equivalences calculated by the program are printed out at the end of the calculation.

Because of the identical denominators in (7.8) and (7.9), it may seem that a similar treatment is required for (7.9). This is not so, because in most cases (7.9) can be expressed as a sum of integrals like (7.1) or (7.8) through a projective method of Brown and Feynman [29] which is discussed further in Section 9. As a clearly optimal approach to this analysis in an algebraic program has not yet been decided, the program in Appendix 1 is designed to halt calculation, in general, when faced with an integral such as (7.9). There is an exception to this course when $\mu = \nu$, for then it may be possible to cancel the resulting factor of k^2 with a factor of either k^2 or $k^2 - \lambda^2$ (since the λ^2 is then superfluous) in the denominator and reduce the problem to the detection of an integral with a solution on the list ITYP. The program follows that procedure.

The controlling function in Appendix 1 for the integral branch of the program is still *intop*, but its definition is rather more complicated than in (7.7) to allow it to

handle the separate courses of action as described above for the three general cases illustrated by (7.1), (7.9), and (7.10).

8. Squaring of Matrix Elements

(a) *Preliminary*

Although we have seen in previous sections that a certain amount of simplification of matrix elements is possible by exploiting the general properties of gamma matrices and the recognition of integrals, most applications of these techniques to physical problems require the eventual calculation of a physically measurable quantity such as a cross-section or a polarization of a final-state particle. Such calculations require summations over spins of particles in a product of the conjugate of one matrix element by another. The procedure for performing this is quite straightforward, and is given in many books, such as that of Bjorken and Drell [2]. The steps involved in such a calculation are three in number, viz,

(i) Taking the conjugate of a matrix element and renaming summed indices

(ii) Performing spin sums by replacing products of matched spinors and polarization vectors by projection operators,

(iii) Computing the resultant trace of the gamma matrix product and sum over indices in terms of invariants or scalar products of the momenta given in the problem.

We shall only consider the first two problems here; once a trace expression is obtained we have the necessary input for any of the simplification programs which calculate such traces, although our internal LISP representation forms a natural input to a program such as REDUCE ([10], [26]).

(b) *Computation of Conjugates of Matrix Elements*

The computation of the conjugate of a matrix element is another example of a straightforward mapping operation. Because of our choice of metric and gamma matrices, seven rules are involved in the calculation, namely:

$$\begin{aligned}
&\text{(i)}\quad I \rightarrow -I \\
&\text{(ii)}\quad U(P) \rightarrow UB(P) \\
&\text{(iii)}\quad UB(P) \rightarrow U(P) \\
&\text{(iv)}\quad V(P) \rightarrow VB(P) \\
&\text{(v)}\quad VB(P) \rightarrow V(P) \\
&\text{(vi)}\quad \gamma_5 \rightarrow -\gamma_5
\end{aligned} \tag{8.1}$$

(vii) the order of terms in a product of gamma matrices and spinors is reversed.

The validity of these rules follows immediately from the definition of the conjugate of a matrix element as given for example by Bjorken and Drell.

The function *cnj* and its associated function *gcnj*, which perform this transformation and rename summed indices, are given in Appendix 1 without further comment.

(c) *Calculation of Sums over Spins and Polarizations*

The computation of the sum over spins in a product of two matrix elements is a less trivial operation. Again the rules for doing this are straightforward but more searching of the expression is involved in their application. The rules for summing over fermion spins are as follows:

$$\sum_{\text{spins}} u_\alpha(p)\,\bar{u}_\beta(p) = (\gamma \cdot p + m)_{\alpha\beta}$$
$$\sum_{\text{spins}} v_\alpha(p)\,\bar{v}_\beta(p) = (\gamma \cdot p - m)_{\alpha\beta} \tag{8.2}$$

where m is the mass of the particle with momentum p.

The difficulty which arises in applying these rules is that the matching right spinor does not always occur adjacent to the left, and so the expression must be searched for this spinor. Moreover, when it is found, the gamma matrix line identification variables in the line associated with the right spinor must be renamed to conform with the left. After the whole expression has been scanned in this manner, the sum over the matrix components in (8.2) for a given line converts the expression to a trace. The functions *spinsum* and *findline* which perform these operations are also given in Appendix 1.

A sum over particle polarizations if required follows in an analogous manner.

and

$$\sum_{\gamma} \epsilon_\mu \epsilon_\nu = -g_{\mu\nu} \frac{+k_\mu k_\nu}{m_V^2} \qquad \text{for vector mesons.} \tag{8.3}$$

The application of these formulas, however, is more straightforward than that of the analogous fermion formulas. In the case of photons, each polarization vector encountered for the first time is declared an index, and a corresponding minus sign included with the expression. With vector mesons, the first occurrence of a polarization vector is renamed, both its new and old names are declared indices, and the whole expression multiplied by the appropriately labelled projection operator in (8.3). These processes are performed by the function *polsum* defined in Appendix 1.

Finally, a function *mpair*, whose two arguments are matrix elements, computes the product of the conjugate of its first argument by its second using the procedures described in this section.

9. Heuristic Methods

(a) *Preliminary*

In this paper so far, we have considered the solution by computer of many of the well-formed problems which arise in calculations in quantum electrodynamics and other fields where processes are presented in terms of diagrams. For the cases which we have discussed in detail, either there exist single algorithms or else individual procedures which have appeared to us to be clearly superior to various available alternatives. The question of heuristics involves choices between alternatives where preferences are not clear-cut, description of operations which are not particularly well-defined, and overall control of the programs which we have introduced in the previous sections of this paper. These three types of questions are listed in increasing order of difficulty. In Section 9b we consider asymptotic operations, an example of the first type. Section 9c contains a more general commentary on integration than is found in Section 7. At present integration is probably best classified as a problem of the second type, but recent mathematical work ([30], [31]) on the subject of integration in finite terms suggests that this status is rather uncertain. Where there is a possibility that a procedure may become better defined in the light of evidence gained during the operation of a program, ideally that program should be able to improve itself by self-modification. The ability of LISP programs to do so is explained further in Section 9d. We discuss the problem of heuristic control of our computations in Section 9e, and conclude in Section 9f with some remarks about future work.

(b) *Asymptotic Operations*

In general, when an algebraic result calculated by the programs is too long to be useful in an understanding of the principal features of a problem, one tries to discard all but the leading terms in some combination of kinematical variables. Hence, the terms to be retained carry a certain minimum asymptotic weight, computed, for example, from the power of the initial energy for a process. Since power-counting in LISP is simply a matter of scanning arbitrary list structures for occurrences of a given base (say X) in S-expressions such as X, (TIMES X X) or (EXPT X N), there is no need to give detailed examples of appropriate functions. Also, excessive length of results has not often constituted a difficulty of calculation with the programs described in this paper. Of course, that situation is quite different [10] for programs which simplify the products of traces of Dirac matrices that occur in absolute squares of matrix elements. We mention the power-counting technique here because it is essentially common to the heuristics governing both types of program, and because it is, therefore, required by the procedures for heuristic control of programs which are outlined in Section 9e.

A somewhat different justification for the use of asymptotic operations occurs

For integrals over four-momenta, we discuss by way of illustration a projective method of Brown and Feynman [29]. The method reduces the calculation of integrals like (7.9) which are second-rank tensors to the solution of a set of simultaneous equations, followed by detection of simpler integrals such as (7.8) whose values should already be recorded on tables by the programs of Appendix 1. Suppose that we have an integral over k in which the denominator of the integrand is a product of the factors

$$k^2-\lambda^2,\ k^2-2p_1\cdot k,\ k^2-2p_2\cdot k \text{ and } k^2-2p_3\cdot k-b.$$

With the code of (7.3), we can then write (2 3 3 4) as a suitable abbreviation for the denominator. The most general form for the integral

$$I_{\mu\nu} = \int \frac{k_\mu k_\nu d^4k}{(2\,3\,3\,4)} \tag{9.3}$$

must be

$$I_{\mu\nu} = c_{1\mu}p_{1\nu} + c_{2\mu}p_{2\nu} + c_{3\mu}p_{3\nu} + c_4\delta_{\mu\nu} \tag{9.4}$$

in terms of the "external" four-vectors in the problem. Using (9.3) and (9.4), and taking the dot or scalar products of each equation with $2p_1$, $2p_2$, and $2p_3$ in turn, we obtain three equations, of which

$$2(c_{1\mu}m_1^2 + c_{2\mu}p_1\cdot p_2 + c_{3\mu}p_1\cdot p_3 + c_4p_{1\mu}) = \int \frac{k_\mu d^4k}{(3\,3\,4)} - \int \frac{k_\mu d^4k}{(2\,3\,4)}, \tag{9.5}$$

where the second term on the right-hand side is no longer a function of p_1, is a representative example. A fourth equation can be obtained from the relation between (9.3) and (9.4) for $I_{\mu\mu}$. These four equations provide solutions for the c coefficients, because values of all integrals on the right-hand sides of equations like (9.5) should already be recorded on either of the lists ITYP or PRESLIST quoted in Section 7. Once again, functions for algebraic manipulation are all that are needed to make the solution of this set of simultaneous equations possible. For systematic use, we expect the method to be quite fast when we have implemented it with the help of existing functions [26] to expand determinants algebraically. In principle, it is also valid for tensor quantities of higher rank than (9.3).

A second method of extension of tables that we have tested involves differentiation. For example, if

$$\int \frac{d^4k}{(k^2-\lambda^2)^2(k^2-2p\cdot k)} = \int \frac{d^4k}{(2\,2\,3)} = f(\lambda^2, p),$$

then

$$\int \frac{d^4k}{(2\,2\,2\,3)} = \frac{1}{2}\frac{\partial}{\partial(\lambda^2)} f(\lambda^2, p).$$

is almost always correct that compiled functions operate faster than interpreted functions, so that in fact we do compile functions whose definitions are unlikely to be changed, and which will be called many times in long production runs of a program.

When the question of speed is irrelevant or unimportant, there is one strong advantage of interpreted functions. As a comparison between Appendix 1 (consisting of programs) and the various specimens of data given as examples throughout this paper shows the, S expression syntax which somewhat resembles (2.12) is the same for programs and for data. This is a feature not often found in other programming languages, where usually the rules for writing programs and data are completely distinct. Thus, in LISP, programs can treat other programs, or themselves, as data and, hence, modify themselves or each other. Therefore, LISP is an ideal medium for the writing of "learning" programs, programs for artificial intelligence, or heuristics of the type necessary for the control of our own programs.

So far, our main application of the method to a problem of physics has concerned the calculation of a matrix element M_2 from a Feynman diagram with four vertices, in such a way that it should be proportional to a matrix element M_1 from a diagram with two vertices [20]. Because the matrix elements are for electron-proton scattering, and the form of the proton vertex function

$$\gamma_\mu F_1(q) + (2m)^{-1}\kappa F_2(q)\gamma_\mu$$

that occurs in M_1 is to be preserved in M_2, only one out of about four different sequences of algebraic substitutions for four-momenta is "correct". The appropriate function produced this correct answer by trial and error, changed its own definition to remove the incorrect possibilities, and then went on to take part in the calculation of three other matrix elements from diagrams with four vertices, where the form of the proton vertex function was also to be preserved. Historically, this success was quite important because it obviated the need for the programmer to test each alternative with a separate run of the program in a batch-processing situation where turn-around time for jobs was of the order of three days.

(e) *Heuristic Control of Calculations*

The common feature of each of the problems discussed in preceding sections of this paper and in [10] is that a compact algorithm for their solution exists in LISP. It is now almost trivial to write a program which links all of these routines together and computes a differential cross-section, for example, from a given set of diagrams or a Hamiltonian of interaction which generates these diagrams. For instance, given a function, *reduce*, to carry through the calculation of traces and the sums over indices, we can define a function *cxn* to compute the differential cross-section from any single diagram as follows:

$$\text{cxn[dia]} = \lambda[[x]; \text{mathprint[reduce[mpair[x;x]]]]}\ [\text{matrix[dia]}].$$

The reason why such a naive approach to the complete calculation is not very helpful for current research problems in quantum eletrodynamics is that the amount of computation necessary, and the size of expressions produced, can grow catastrophically large unless very careful control is exercised over the calculation at each step. This is a lesson that a theoretical physicist leans quickly if he is to hope for success in this field. It is equally important in the computer program that such control is exercised. The point is to be stressed even more in symbolic than in numerical calculations, because here the computation time and storage requirements tend to grow at exponential rates rather than the power rates encountered in numerical analysis. In each of the following paragraphs, we mention some of the heuristic decisions which a theoretical physicist makes at various places in a complete calculation.

At the diagram stage, we must select only those diagrams which are relevant to the given calculation. For example, one usually rejects disconnected graphs, or those whose contribution is known on heuristic grounds to be negligible for the problem in hand. The decisions which influence this choice are not always easy to formulate in detail. Thus, one habitually works with the Bethe-Salpeter equation in the approximation that considers only ladder diagrams, because other approaches are considerably more difficult. We can hope for better methods of selecting subsets of diagrams when the asymptotic operations described in Section 9b are more fully developed, so that they can add information to physicists' present stock of intuition.

At the matrix-element stage, much reduction is possible, particularly if closed loops in diagrams are involved. Several suitable tricks have been proposed, and their best review is provided by Nakanishi [33]. If such methods are not employed, then, even if it may not be unduly difficult to keep track of the large numbers of terms generated, eventual numerical calculation based on the symbolic result may lose accuracy on computation if many terms contribute to the result. A simple example [34] raises the question of how best to represent the value of $\int_0^1 x^n e^{-kx}\, dx$ for calculation, when it occurs somewhere in a physical computation based on LISP.

At the stage where matrix elements are squared, it may be best from the point of view of symmetry and compactness to represent long traces which are associated with the spin sums pictorially in terms of diagrams containing closed loops between pairs of Feynman diagrams, as discussed by Bjorken and Drell [2]. An example of such diagrams, which represent the distinct traces arising in the cross-section computed from Fig. 2, is given in Fig. 5. As in this case, it often turns out that because of the symmetries arising from identical fermions or bosons in a process, many of the diagrams have the same topological structure, so that the cross-section may be written in terms of a smaller set of traces than would normally be necessary. This recognition can have an important bearing on the amount of computation

necessary, especially if integrations over loop or final-state momenta are necessary. Additionally, the symmetries in the final cross-section may be more apparent if such symmetries are recognized at this stage.

Following the principle of Section 1, we do not discuss trace algorithms and their implementation in this paper. However, we note that the preparation of a trace expression for evaluation, and the sequence in which trace operations are

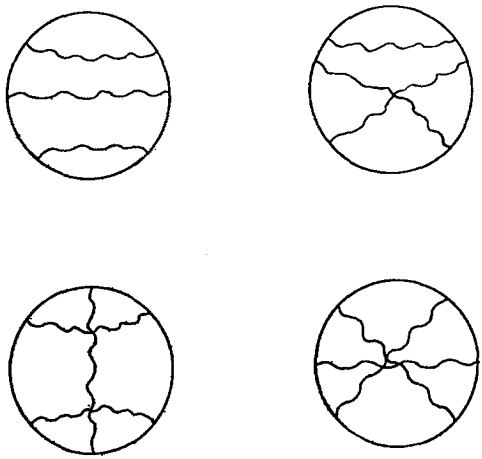

FIG. 5. Graphical representation of the spin sums arising in the cross-section computed from the Feynman diagrams in Fig. 2.

carried out, can have important consequences. To that extent, it may not always be best to give the calculation to an arbitrary algebraic simplifying program. For example, we have found by analyses of computing time that if we have a product of two traces which share common indices, it is faster to use Chisholm's algorithm [35], which reduces the product to a sum of single traces, only if three or more indices are shared. Otherwise, it is faster to compute each trace separately and then contract indices. However, Chisholm's algorithm may not be the optimal way even for three or more indices, as one discovers that the same expression arises over and over again in the computation of seemingly different traces. This was recognized by Kahane [36], who found that it was possible in a single trace to reduce the number of redundant calculations by a rearrangement of a pictorial representation of the trace. This is an algorithm which is highly suited for computer implementation, and we are in the process of implementing it.

Finally, at all stages of calculation one should attempt to use asymptotic methods to make reductions wherever possible.

A successful theoretical physicist, then, incorporates many considerations of the above type into his calculation, and a completely successful computer program for performing these calculations must do the same.

Ultimately, we hope to include many of these ideas in a heuristic supervisory program which will automatically control the calculation from start to finish. The problems are fairly well-defined, even though no essential algorithms exist as such for their solution. At the present time, the heuristic supervisor remains the physicist, who sits in front of a console, and directs the calculation now as we hope that a program will do later. The most successful interface between man and machine has been a keyboard and CRT display at the Stanford Artificial Intelligence Project, backed by a high-speed line printer for hard copy.

(f) *Future Activities*

The main area of present and future work, on heuristic control and integration, has been described above. In this development, we intend to make the maximum use of man-machine interaction through such devices as teleprinters, so that future