tables to their own segment, in the same way we did with generic arrays in Chapter 5.

§3 Computer algebra

One of the most revolutionary recent developments in scientific programming is the ability to do algebra on the computer. Programs like REDUCE, SCHOONSCIP, MACSYMA, DERIVE and MATHEMATICA can automate tedious algebraic manipulations that might take hours or years by hand, in the process reducing the likelihood of error[14]. The study of symbolic manipulation has led to rich new areas of pure mathematics[15]. Here we illustrate our new tool (for compiling finite state automata) with a rule-based recursive program to solve a problem that does not need much formal mathematical background. The resulting program executes far more rapidly on a PC than REDUCE, *e.g.*, on a large mainframe.

§§1 Stating the problem

Dirac γ-matrices are 4×4 traceless, complex matrices defined by a set of (anti)commutation relations[16]. These are

$$\gamma^\mu \gamma^\nu + \gamma^\nu \gamma^\mu = 2\eta^{\mu\nu}, \quad \mu, \nu = 0, \dots, 3 \qquad (1)$$

where $\eta^{\mu\nu}$ is a matrix-valued tensor,

$$\eta^{\mu\nu} = \begin{array}{c|cccc} \mu\backslash\nu & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & -1 & 0 \\ 3 & 0 & 0 & 0 & -1 \end{array} \qquad (2)$$

14. R. Pavelle, M. Rothstein and J. Fitch, "Computer Algebra", *Scientific American* 245, #6 (Dec. 1981) 136.
15. See, *e.g.*, M. Mignotte, *Mathematics for Computer Algebra* (Springer-Verlag, Berlin, 1992).
16. They are said to satisfy a Clifford algebra. See, *e.g.*, J.D. Bjorken and S.D. Drell, *Relativistic Quantum Mechanics* (McGraw-Hill, Inc., New York, 1964); also V.B. Berestetskiĭ, E.M. Lifshitz and L.P. Pitaevskiĭ, *Relativistic Quantum Theory, Part 1* (Pergamon Press, Oxford, 1971) p. 68ff.

The **trace** of a matrix is defined to be the sum of its diagonal elements,

$$\mathrm{Tr}(\not{A}\,) \mathrel{\hat{=}} \sum_{k=1}^{N} \not{A}_{kk} \,. \qquad (3)$$

Clearly, traces obey the **distributive law**

$$\mathrm{Tr}(\not{A} + \not{B}\,) \equiv \mathrm{Tr}(\not{A}\,) + \mathrm{Tr}(\not{B}\,) \qquad (4a)$$

as well as a kind of **commutative law**,

$$\mathrm{Tr}(\not{A}\,\not{B}\,) \equiv \mathrm{Tr}(\not{B}\,\not{A}\,)\,. \qquad (4b)$$

From Eq. 1, 2, and 4a,b we find

$$\begin{aligned}\mathrm{Tr}(\not{A}\,\not{B}\,) &\equiv \frac{1}{2}\left[\mathrm{Tr}(\not{A}\,\not{B}\,) + \mathrm{Tr}(\not{B}\,\not{A}\,)\right] = \frac{1}{2}\mathrm{Tr}(\not{A}\,\not{B} + \not{B}\,\not{A}\,) \\ &= \frac{1}{2} 2A\cdot B\,\mathrm{Tr}(\,\mathbf{1}\,) \equiv 4A\cdot B\end{aligned} \qquad (5)$$

where the ordinary vectors A^{μ} and B^{ν} form the (Lorentz-invariant) "dot product"

$$A\cdot B \equiv A^0B^0 - A^1B^1 - A^2B^2 - A^3B^3 \qquad (6)$$

The trace of 4 gamma matrices can be obtained, analogous to Eq. 5, by repeated application of the algebraic laws 1-4:

$$\begin{aligned}\mathrm{Tr}(\not{A}\,\not{B}\,\not{C}\,\not{D}\,) &= 2A\cdot B\,\mathrm{Tr}(\not{C}\,\not{D}\,) - \mathrm{Tr}(\not{B}\,\not{A}\,\not{C}\,\not{D}\,) \\ &\equiv 2A\cdot B\,\mathrm{Tr}(\not{C}\,\not{D}\,) - \mathrm{Tr}(\not{A}\,\not{C}\,\not{D}\,\not{B}\,) \\ &\equiv 4\left(A\cdot B\,C\cdot D - A\cdot C\,B\cdot D + A\cdot D\,B\cdot C\right)\end{aligned} \qquad (7)$$

We include Eq. 7 for testing purposes. The factors of 4 in Eq. 5 and 7 do nothing useful, so we might as well suppress them in the interest of a simpler program.

§§2 The rules

We apply formal rules analogous to those used in parsing a language[17]. The rules for traces are:

```
\ Gamma Matrix Algebra Rules:

\ a                  ->    string of length < =3
\ /                  ->    delineator for factors
\ <factor>           ->    a/
\ product/           ->    a/b/c/d/ ...
\ Tr( a/b/prod/)     ->    a.b ( tr( prod/) )  - tr( a/prod/b'/)
\  b'                ->    mark b as permuted
\ Tr( a/)            ->    0  (a single factor is traceless)
```

Repeated (adjacent) factors can be combined to produce a multiplicative constant:

$$\not{B}\,\not{B} \equiv B\cdot B\,; \tag{8}$$

recognizing such factors can shorten the final expressions significantly, hence the rule

```
\ Tr( a/a/prod/)     ->    a.a ( Tr( prod/) ).
```

In the same category, when two vectors are orthogonal, $A\cdot B = 0$, another simplification occurs,

```
\ PERP  A B          ->    A.B = 0.
\ Tr( A/B/prod/)     ->    - Tr( A/prod/B'/)
```

The ability to recognize orthogonal vectors lets us (correctly) include the trace of a product times the special matrix[18]

17. It is not difficult to introduce one further level of indirection and produce Forth code for generating a "compiler". See T.A. Ivanco and G. Hunter, *J. Forth Appl. and Res.* **6** (1990) 15.
18. Note: Berestetskiĭ, *et al.* (*op. cit.*) define γ^5 with an overall "–" sign relative to Eq. 9.

$$\gamma^5 = i\gamma^0\gamma^1\gamma^2\gamma^3 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{9}$$

that **anticommutes** with all four of the γ^μ :

$$\gamma^5\gamma^\mu + \gamma^\mu\gamma^5 \equiv 0\,, \quad \mu = 0,\dots,3\,.$$

The fully antisymmetric tensor $\varepsilon_{\mu\nu\kappa\lambda} \equiv -\,\varepsilon_{\nu\mu\kappa\lambda}$, *etc.*, lets us write

$$\gamma^5 \not{A} \not{B} \not{C} \equiv i\,\varepsilon_{\mu\nu\kappa\lambda} A^\mu B^\nu C^\kappa \gamma^\lambda\,, \quad i = \sqrt{-1} \quad . \tag{10}$$

A complete gamma matrix package includes rules for traces containing γ^5 :

```
\ Trg5( a/b/c/d/x/)  ->    i Tr( ^/d/x/)
\  ^.d               ->    [a,b,c,d] (antisymmetric product)
\ Note:   ^.a = ^.b = ^.c = 0
```

Since γ-matrices often appear in expressions like $(\not{A} + m_A \mathbf{I})$, it will also be convenient to include the additional rules

```
\  *A/               ->    [A/ + mA]
\  Tr( *A/ x/ )      ->    m[A] ( Tr( x/)  ) + Tr( x/ A/)
```

§§3 The program

We program from the bottom up, testing, adding and modifying as we go. Begin with the user interface; we would like to say

```
TR( A/B/C/D/)
```

to obtain the output:

```
= A.BC.D-A.CB.D+A.DB.C ok
```

Evidently our program will input a string terminated by a right parenthesis ")", *i.e.*, the right parenthesis tells it to stop inputting. This can be done with the word[19]

```
: get$  ASCII )  TEXT  PAD X$  $!  ;
```

Since the rules are inherently recursive, we push the input string onto a **stack** before operations commence. What stack? Clearly we need a stack that can hold strings of "arbitrary" length. The strings cannot be *too* long because the number of terms of output, hence the operating time, grows with the number of factors N, in fact, like $\left(\frac{1}{2}N\right)!$.

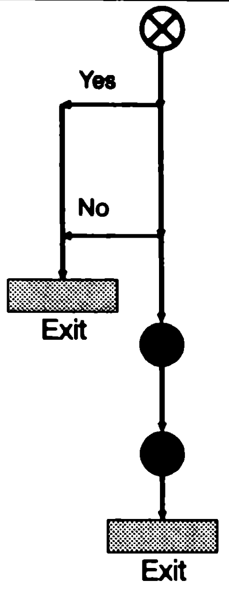
The pseudocode for the last word of the definition is clearly something like

```
: TR(      )get$          \ get a $ -terminated with ")"
           setup          \ push $ on $stack
           parse  ;
```

The real work is done by **parse**, whose pseudocode is shown below in Fig. 11-4; note how recursion simplifies the problem of matching left and right parentheses in the output.

Next we define the underlying data structures. Recursion demands a **stack** to hold strings in various stages of decomposition and permutation. Since the number of terms grows very rapidly with the number of factors, it will turn out that taking the trace of as many as 20 distinct factors is a matter of some weeks on –say– a 25 Mhz 80386 PC; that is, 14 or 16 factors are the largest practicable number. So if we make provision for expressions 20 factors long, that should be large enough for practical purposes[20].

19. The word **TEXT** can be defined as `: TEXT   WORD  HERE  PAD  $!  ;`
20. Actually, 19 factors, since we want to **ALLOT** space in multiples of 4 to maintain even paragraph boundaries. That is, the strings will use 20 bytes including the count.



```
: parse    $pop              \ a/b/x/
    2factors   ( - - a b)    \ x/
    b=b'?                    \ done?

    A.B                      \ output

    length( x/) 0>           \ more?

    rearrange                \ ~a/x/b'/
                             \ x/
    ." ( " recurse ." )"     \ Tr( x/)

    recurse                  \ Tr( ~a/x/b'/)
;
```

Fig. 11-4 *Pseudocode/flow diagram for "parse"*

How deep can the $stack get? The algorithm expressed by the rule

\ Tr(a/b/prod/)   ->   a.b (Tr(prod/)) - Tr(a/prod/b'/)

suggests that for an expression of length n = 2k, the maximum depth will be k + 1. Thus we should plan for a stack depth of at least 11, perhaps 12 for safety. Assuming factor-names up to three characters long, and strings of up to 20 names, we need ≈60 characters per string. My first impulse was to create a dynamic $stack that could accomodate strings of variable length, meaning that some 330 bytes of storage would be needed, at the cost of some complexity in keeping track of the addresses. This is a big improvement on 720 bytes needed for a 60-wide fixed-width stack, of course. However, the convenience of a fixed-width $stack led me ultimately to set up a table (array) of 20 names, whose indices would act as tokens; that is, the strings that actually go on the $stack would be tokenized. The memory cost of the table together with a 12-deep, fixed-width $stack is thus only 80 + 242 = 322 bytes.

```
TASK GAMMA
\ FINITE STATE MACHINE
: WIDE  0 ;
: FSM:  ( width 0 – )  CREATE  , , ]
    DOES>   ( col# – )
        UNDER D@          ( – adr col# width state )
        * +  1+  4*       ( – adr offset )
        OVER +            ( – adr adr' )
        DUP@ SWAP 2+      ( – adr [adr'] adr'+2 )
        EXECUTE@          ( – adr [adr'] state')
        ROT !  EXECUTE   ;
0 CONSTANT >0  3 CONSTANT >3
1 CONSTANT >1  4 CONSTANT >4
2 CONSTANT >2  5 CONSTANT >5
\ END FINITE STATE MACHINE

\ Automatic conversion tables
: TAB:  ( #bytes – )
    CREATE  HERE OVER ALLOT
              SWAP 0 FILL        \ init table
    DOES>  +  C@ ;

: install    ( col# adr char.n char.1 – )     \ fast fill
    SWAP 1+ SWAP                              \ addresses
    DO DDUP I +  C! LOOP DDROP ;
\ end automatic conversion tables

\ STRING HANDLING
HEX
\ Note: ?((( ...... ))) conditionally compiles "......"
FIND $! 0=
  ?(((: $!  ( sadr dadr – )  OVER C@ 1+    CMOVE ; )))
FIND $ + 0=
  ?((( : $ +     ( adr$1 adr$2  – pad )
    DUPC@   ( – $adr a)
    >R 1+  OVER C@ PAD + 1+
    R@  <CMOVE   PAD $!
    R> PAD C@  +   0 MAX FF MIN   PAD C! ; )))
DECIMAL
: 1+C!  ( adr – )  DUPC@  1+  SWAP  C! ;
: -BL  ( $adr – )      \ delete all blanks from $
    DUP>R   0 PAD C!      COUNT OVER + SWAP
    DO  I C@ DUP   32 < >
        IF   PAD COUNT  +  C! PAD 1+C!
        ELSE  DROP  THEN
    LOOP  PAD R>  $! ;
\ END STRING HANDLING

\ PARSE WORDS
\ Note: The factor *A/ in the input string
\ means (A/ + M sub A)
0 VAR star
: star!  128 IS star  DROP ;      \ set 7th bit if 1st char = *

0 VAR #/                          \ counts # of factors
: +#/           AT #/  1+!      DROP ;   \ inc #/
```

```
: err  CR -1 ABORT" Name -> letter ( letter | numeral )* " ;
: +PAD  ( char – )  star OR       \ set bit 7
    PAD COUNT  +  C! PAD C@ 1+  PAD C!  0 IS star ;

5 WIDE FSM: (factor)      ( char col# – )
\ input | other  |   letter   |  digit  |    /     |   *   |
\ state ------------------------
( 0 )    err  >0  +PAD  >2  err  >0   +#/   >5  star! >1
( 1 )    err  >0  +PAD  >2   err  >1   +#/   >5  err  >1
( 2 )    err  >0  +PAD  >3  +PAD  >3  +#/  >5  err  >2
( 3 )    err  >0   +PAD  >4 +PAD  >4 +#/   >5  err  >3
( 4 )    err  >0   DROP  >4 DROP >4  +#/  >5  err >4
     ;     \ terminate definition

128  TAB:  [factor]                 \ convert char to col#
1  ' [factor]  ASCII Z ASCII A  install
1  ' [factor]  ASCII z ASCII a  install
2  ' [factor]  ASCII 9 ASCII 0  install
3  ' [factor]  ASCII /    +  C!
4  ' [factor]  ASCII *    +  C!

: <factor>  ( adr – adr')
    PAD 0!  ' (factor) 0!  0 IS star          \ initialize
    BEGIN DUPC@  DUP  [factor] (factor) 1+
        ' (factor) @ 5 =
    UNTIL ;

CREATE factor{  20 4* ALLOT  OKLW    \ up to 20 factors
: }  ( adr n – adr + 4*n) 4*  +  ;   \ compute address

0 VAR N   0 VAR N1
CREATE BUF$  20 ALLOT  OKLW     \ data structures

: unstar   127 AND ;        \ token[*A] = token[A] 128 OR
: $= ( $adr1 $adr2 – f )
    -1 -ROT  COUNT
    ROT COUNT   ( – $adr2 + 1 n2 $adr1 + 1 n1 )
    ROT DDUP  =
    IF    DROP
        0 DO DUPC@ unstar     >R 1+
            OVER C@ unstar R>  < >
            IF  ROT NOT -ROT LEAVE THEN
           SWAP 1+
        LOOP  DDROP
    ELSE DDROP DDROP NOT THEN ;
: check.table  ( - - )             \ prevent duplicate tokens
    -1 IS N1
    N 0  DO  PAD factor{ I } $=
        IF  I IS N1  LEAVE THEN
    LOOP  ;

: +buf$  N N1 -1 = AND N1 -1 > N1 AND  +
                                       \ token N or N1
    PAD 1 + C@ 128 AND  OR             \ if star, set bit 7
    BUF$ DUPC@ + 1+  C! BUF$ 1 + C! ;
                                       \ append token
```

```
: tokenize  ( $adr -- )               \ decompose into factors
    factor{ 80 0 FILL                 \ initialize table
    COUNT  OVER +   ( -- $adr+1 $adr')  \ addresses
    >R                        \ $adr' = $adr + LEN($) + 1
    0  BUF$  C!               \ init. buffer
    0 IS N                    \ init. N
    BEGIN  <factor>           \ begin loop; get factor
        check.table           \ prevent multiple entries
        +buf$                 \ add factor to tokenized $
        N1 -1 =               \ not in table?
        IF  PAD  factor{ N } $!    \ put in table
            AT  N  1+!        \ inc. N
        THEN
    DUP R@  =  UNTIL          \ end loop
    DROP  RDROP ;             \ clean up
\  END PARSE WORDS

\ $stack
CREATE  $stack  20 20 * 2 +  ALLOT OKLW     \ 2 for ptr
: $push   ($adr --)
    DUPC@  19 >  ABORT" STRING TOO LONG!"
    $stack  DUP@  DUP  19 >  ABORT" $stack too deep!"
    20 *  2+   +    $!       $stack 1+! ;
: $pop   ($adr --)  $stack  DUP@   1-  0  MAX
    DDUP SWAP  !
    20 *  2+   +  SWAP  $! ;
\  end $stack

CREATE  X$ 80 ALLOT OKLW          \ buffer for input $, tail$
: get$  ASCII )  TEXT        \ input a $ terminated by )
    PAD  -BL                 \ delete blanks
    PAD X$  $! ;

:  1factor   (-- a)            \ get 1 factor, tail -> x$
    BUF$  1+  C@                 (-- a)
    BUF$  COUNT  1-  >R
    1+  X$ 1+  R@  <CMOVE            \ tail -> x$
    R>  X$  C! ;                 \ adjust count
: 2factors  (-- a b)             \ get 1st 2 factors
    1factor  X$ BUF$  $!  1factor ;

0  VAR sign                              \ emit correct sign
2 TAB:  [sign]                           \ make table
ASCII  +   ' [sign]       C!             \ fill table
ASCII -    ' [sign]  1+  C!
: .sign    (a --)  64 AND  0>  ABS  [sign]  sign AND  EMIT  ;

\ Note: we mark prime ( ' )  by 64 OR   (set bit 6 in token)
\      since 1st factor is never ', we set bit 6 for leading "–" sign
: prime!    64   OR   ;
: unprime   63   AND  ;
: A.B ( a b --)                        \ emit "dot product"
    unprime                            \ b' -> b
    OVER   .sign                       \ emit sign
    SWAP  unprime                      \ drop leading "–"
    DDUP  MAX  -ROT   MIN              \ sort factors
    factor{ SWAP }  $.
    ASCII . EMIT     factor{ SWAP }  $.      -1 IS sign  ;

: clean     ($adr --)                \ unprime all factors
    COUNT OVER  +  SWAP
    DO  I C@   unprime   I C!   LOOP ;


: rearrange               (a b -- )
    1 BUF$ !   \ init buf$
    BUF$  X$  $+   PAD  BUF$ $!   \ buf$ -> _/x/
    prime!                        (-- a b')
    BUF$ COUNT  +   C!   BUF$ 1 + C!    \ buf$ -> _/x/b'/
    64 XOR           (-- a xor 64)    \ toggle sign of a
    BUF$ 1 +   C!  ;  (--)          \ buf$ -> ~a/x/b'/

: setup     X$  tokenize                  \ form token$
            $stack  0!               \ init $stack
            0  IS sign               \ init sign flag
            BUF$  $push   ;          \ input token$ on $stack

\                               debugging code
: $$.    ($adr --)  COUNT  OVER  +  SWAP\ translate token$
    DUPC@  .sign   factor{ OVER C@  unprime } $. ." /"
    1 +  DO    factor{ I C@ unprime }   $.
               I C@  64 AND 0>            \ primed?
               IF ." ' "  THEN     ." /"
    LOOP ;
: dump$stack? DEBUG  NOT  IF   EXIT   THEN
    $stack 2 +               (-- $adr[0] )
    $stack @  DUP  0=   \ $stack empty?
    IF  DDROP   CR ." $stack empty"  EXIT  THEN
    0  DO  I  20 *  OVER  +   CR $$.   LOOP  DROP  ;
0  VAR  DEBUG
: DEBUG-ON  -1 IS DEBUG  ;
: DEBUG-OFF  0 IS DEBUG  ;
\                               end debugging code

: (.    ." ("   0 IS sign  ;
: ).    ." )"  -1 IS sign  ;
: parse   dump$stack?
    BUF$   $pop           2factors    (-- a b)
    DUP  64 AND  0>                  \ b = b' ?
    IF   DDROP  EXIT  THEN
    DDUP  A.B       (-- a b)         \ output
    X$ C@ 0>                         \ more?
    IF     rearrange              \ buf$ = ~a/x/b'/
           BUF$ $push                \ ~a/x/b'/
           X$  clean   X$  $push            \ x/
           (.   RECURSE  ).
                RECURSE
    ELSE      DDROP    THEN   ;

: TR( get$                \ input $
    setup
    ."  = "    CR             \ for beauty
    parse  ;
```

Since the tokens are 1-byte integers smaller than 32, their 5th, 6th and 7th bits can serve as flags to indicate their properties. For example, we need to indicate whether a factor was "starred", *i.e.* whether it represents $(\mathcal{A} + m_A \mathbf{I})$ or $\mathcal{A}$, according to the rule

```
\ *A/                   ->     (A/ + m[A] ) .
```

Again, we need to be able to indicate a "prime", showing that a factor has been permuted following the rule

```
\ Tr( a/b/prod/)      ->     a.b ( tr( prod/) )  - tr( a/prod/b'/)
```

Thus we set bit 7 (**128 OR**) to indicate "star", and bit 6 (**64 OR**) to indicate "prime".

We still need to indicate the leading sign. My first impulse was to use bit 5, but I realized the first factor is never permuted, hence its 6th bit is available to signify the sign. It is toggled by the phrase **64 XOR**. (In the $stack pictures appearing in the Figures we indicate toggling by a leading " ~ ".)

Programming these aspects is fairly trivial so we need not dwell on it. The entire program appears on pages 279 and 280 above.

Now we test the program:

```
TR( A/B/C/D/E/F/) =
  A.B(C.D(E.F)–C.E(D.F) + C.F(D.E))
–A.C(D.E(B.F)–D.F(B.E) + B.D(E.F))
 + A.D(E.F(B.C)–B.E(C.F) + C.E(B.F))
-A.E(B.F(C.D)–C.F(B.D) + D.F(B.C))
 + A.F(B.C(D.E)–B.D(C.E) + B.E(C.D))  ok
```

Clearly the concept works. Our next task is to incorporate branches to take care of "starred", as well as identical and/or orthogonal adjacent factors. The possible responses to the different cases are presented in decision-table form in Table 11-2:

To avoid excessively convoluted logic we eschew nested branching constructs. A finite state machine would be ideal for clarity; however, as Table 11-2 makes clear, the logic is not really that of a FSM, besides which, the FSM compiler described above would

put:	a/b/x/	*a/b/x/	a/*b/x/	a/a/x/	a/b/x/, a.b=0
esulting	~a/x/b'/	a/b/x/	a/b/x/	...	...
stack:	x/	b/x/	a/x/	x/	~a/x/b'/
ction(s)†:	a.b	m[a]	m[b]	a.a	**RECURSE**
	(**RECURSE**)	(**RECURSE**)	(**RECURSE**)	(**RECURSE**)	
	RECURSE	**RECURSE**	**RECURSE**		

Note: characters shown in light typeface are **EMIT**ted.

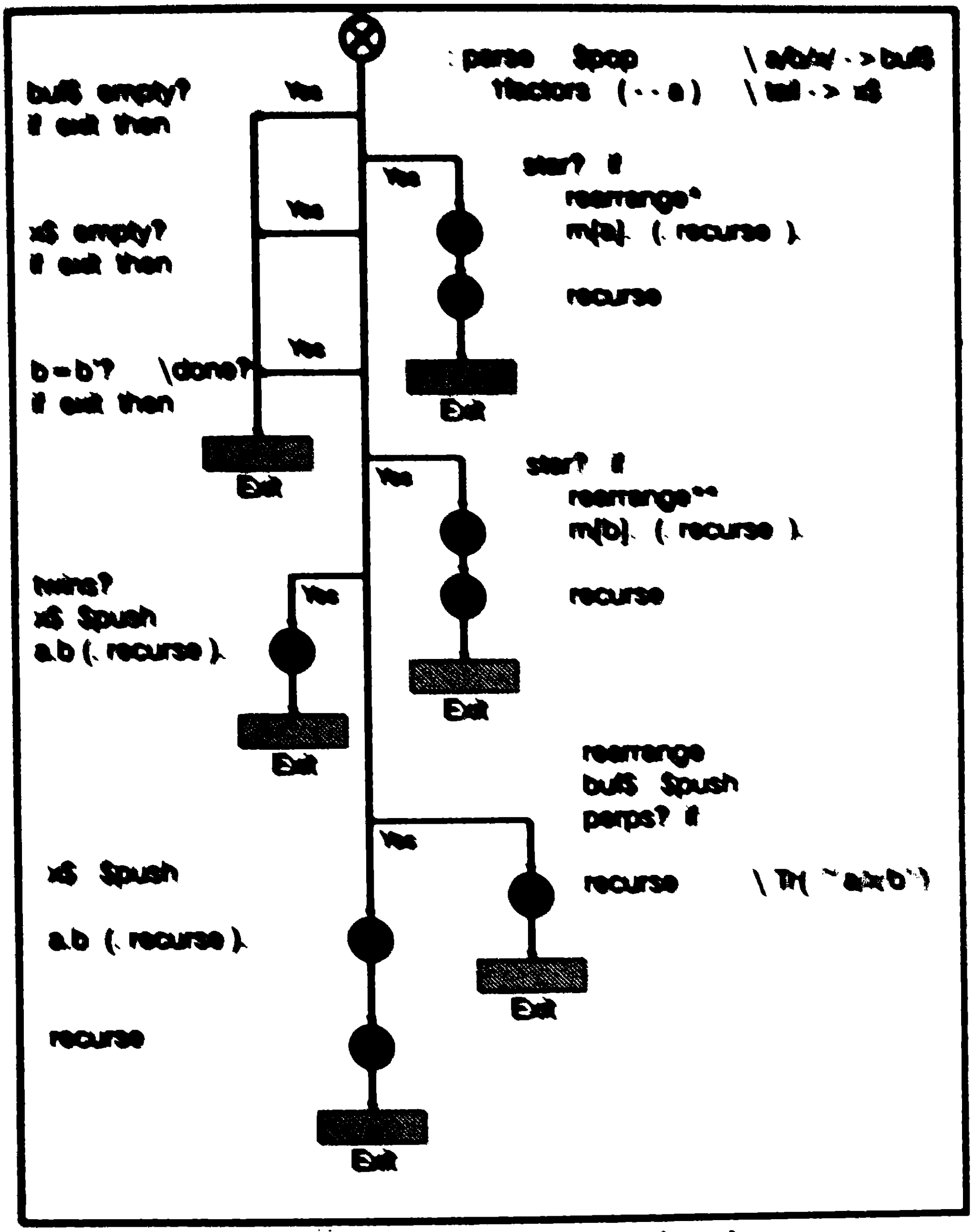
have to be modified to keep its state variable on the stack, since otherwise it could not support recursion. The resulting pseudo-code program is shown in Fig. 11-5. Implementing the code is now straightforward, so we omit the details, such as how to define **PERP** to appropriately mark the symbols. The simplest method is a linked list or table of some sort, that is filled by **PERP** and consulted by the test word **perps?**.

How might we implement a leading factor of γ^5? While there is no difficulty in taking traces of the form

$$\mathrm{Tr}(\gamma^5 \not{A}\, \not{B}\, \not{C}\, \not{D} \ldots) \equiv i\,\mathrm{Tr}(\varepsilon_{\mu\nu\kappa\lambda} A^{\mu} B^{\nu} C^{\kappa} \gamma^{\lambda} \not{D} \ldots)\,, \qquad (11)$$

expressions with γ^5 between "starred" factors are more difficult. However, the permutation properties of traces let us write, *e.g.*,

$$\begin{aligned} &\mathrm{Tr}\Big((\not{A} + m_{\mathrm{A}})\, (\not{B} + m_{\mathrm{B}})\, \gamma^5\, \not{C}\, (\not{D} + m_{\mathrm{D}}) \not{E} \ldots \Big) \\ &\equiv \mathrm{Tr}\Big(\gamma^5\, \not{C}\, (\not{D} + m_{\mathrm{D}}) \not{E} \ldots (\not{A} + m_{\mathrm{A}})\, (\not{B} + m_{\mathrm{B}}) \Big)\,, \end{aligned} \qquad (12)$$



```
: parse   $pop          \ a/b/a/ -> buf$
    1factors  ( -- a )  \ tail -> x$
buf$ empty?
if exit then
                        star? if
                            rearrange*
                            m[a]. ( recurse ).
                            recurse
x$ empty?
if exit then
b=b?   \done?
if exit then
                        star? if
                            rearrange**
                            m[b]. ( recurse ).
                            recurse
twins?
x$ $push
a.b ( recurse ).
                            rearrange
                            buf$ $push
                            perps? if
x$ $push                    recurse     \ Tr( ~a/x'b')
a.b ( recurse ).
recurse
```

Fig. 11-5 Pseudocode/flow diagram for extended 'parse'

so the key issue becomes decomposing leading starred factors, saving the pieces on the $stack, until at least three distinct unstarred factors are adjacent on the left. Replace these by a special token which stands for " ^ " as shown on page 276. This token is marked orthogonal to all three of the vectors it represents, at the time it is inserted.

To avoid further extending **parse**, probably the best scheme is to define a distinct word, **Trg5(**, that uses the components of **parse** to perform the above preliminary steps. Then **Trg5(** will invoke **parse** to do the rest of its work.

The only other significant task is to extend the output routine to

a) recognize the special " ^ " token; and

b) replace dot products like " **^.d** " by **[a,b,c,d]**.

A final remark: one or another form of vectoring can simplify **parse** (relative to Fig. 11-5) by hiding the recursion within words that execute the branches. We have avoided this method here because it conceals the algorithm, a distinct pedagogical disadvantage.

FORmula TRANslator

That prehistoric language FORTRAN —despite its manifold deficiencies relative to FORTH— contains a useful and widely imitated invention that helps maintain its popularity despite competition from more modern languages: This is the FORmula TRANslator from which the name FORTRAN derives.

FORTH's lack of FORmula TRANslator is keenly felt. Years of scientific FORTH programming have not entirely eliminated my habit of first writing a pseudo-FORTRAN version of a new algorithm before reexpressing it in FORTH. Sometimes I will even write a test program in QuickBASIC® before re-coding it in FORTH for speed and power, just to avoid worry about getting the arithmetic expressions correct.

§§1 Rules of FORTRAN

A FORmula TRANslator provides a nice illustration of rule-based programming. To maintain portability, we employ the standard FORTH kernel, omitting special HS/FORTH words as well as CODE words.

In principle we could provide a true compiler that translates formulae to machine code (or anyway, to assembler). But unless we use p-code or some such artifice[21] we would lose all hope of portability. Thus, we take instead the simpler course of translating FORTRAN formulae to FORTH according to the rules

```
\ NUMBERS:
\ <int>           -> {-| Q} {digit  digit^8}
\ <exp't>         -> {dDeE}{& | Q} {digit  digit^2}| Q}
\ <fp#>           -> {-|Q}{ dig | Q } . dig^  <exp't>

\ FORMULAS:
\ <assignment>    -> <subject> = <expression>
\ <id>            -> letter {letter|digit}^6
\ <subject>       -> <id> {<idlist> | Q}
\ <idlist>        -> ( <id> { , <id>}^)
\ <arglist>       -> ( <expr'n>  { , <expr'n> }^)
\ <function>      -> <id> <arglist>
\ <expression>    -> <term>  | < term>  &  <expr'n>
\ <term>          -> <factor> |  <factor> % <term>
\ <factor>        -> <id> |  <fp#> | ( <expr'n> ) |
                  -> <factor> **  <factor>
```

Clearly, the FORTH FORmula TRANslator could become the kernel of a more complete FORTRAN->FORTH filter by adding to the above rules for formulae the following rules for loops and conditionals:

```
\ DO LOOPS:
\ <label>  -> <integer>
\ <lim>    -> {<integer>|<id>}
\ <step>   -> {,<lim>|Q}
\ <do>     -> DO<label> <id> = <lim>,<lim> <step>
```

21. That is, code for an artificial, generic machine whose code can be mapped easily onto the instruction set of a real computer.

```
\ BRANCHING STRUCTURES:
\ <logical expr>   -> <factor> .op. <factor>
\ <if0>            -> IF(<logical expr>)<assignment>
\ <if1>            -> IF(<logical expr>)THEN
\                         {<statement>} ^
\                      END IF
\
\ <if2>            -> IF(<logical expr>)THEN
\                         {<statement>} ^
\                      ELSE
\                         {<statement>} ^
\                      END IF
\
\ <if3>            -> IF(<logical expr>)THEN
\                         {<statement>} ^
\                      ELSEIF(<logical expr>)
\                         {<statement>} ^
\                      END IF
```

§§2 Details of the Problem

The general principles of compiler writing are of course well understood and have been described extensively elsewhere. Several computer science texts expound programs for formula evaluators[22]. Once we have our translator, we can easily make it an evaluator by compiling the FORTH as a single word, then invoking it.

Let us proceed by translating a FORTRAN formula into FORTH code by hand. For simplicity, ignore integer arithmetic and assume all literals will be placed on the intelligent floating point stack (**ifstack**). Similarly assume all variable names in the program refer to **SCALAR**s (see Ch. 5). A word that has become fairly standard is **%**, which interprets a following number as floating point, and places it on the fstack. With these conventions, we see that we shall want to translate an expression like

22. See, *e.g.*, R.L. Kruse, *Data Structures and Program Design*, 2nd Ed. (Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987).

```
A = -15.3E7*EXP(7/X) + Z/(W-SIN(THETA*PI/180)/4)
```

into (generic) FORTH something like this:

```
% 4 REAL*8  >FS
% 180 REAL*8  >FS
PI G\
THETA   >FS G*
GSIN    G\
W  >FS GR-
Z  >FS G\
X  >FS
% 7 REAL*8    G\
GEXP
% -15.3E7 REAL*8  >FS
G*
G+
A  FS>
```

Begin with the user interface. We will define a word, F" , that will accept a terminated string and attempt to translate it to FORTH. That is, we might say

```
F" A=-15.3E7*EXP(7/X)+Z/(W-SIN(THETA*PI/180)/4)"
```

and obtain the output (actual output from the working program!)

```
% -15.3E7 REAL*4  F>FS  % 7 REAL*8  F>FS
X >FS   G/   GEXP  G*   Z >FS  W >FS
THETA >FS  PI >FS  G*  % 180 REAL*8  F>FS
G/  GSIN  % 4 REAL*8  F>FS  G/  GNEGATE
G+ G/   G+  A FS>  ok
```

Although the second version differs somewhat from the hand translation, the two are functionally equivalent.

We would also like to have the possibility of compiling the emitted FORTH words, if F" appears within a colon definition, as in

```
: do.B     F"  B=39.37/ATAN(X**W)+7*Z/X"  ;
```

A FORTRAN expression obeys the rules of algebra in a generally obvious fashion. Parentheses can be used to eliminate all ambiguity and force a definite order on the evaluation of terms