Symbolic Programming

Contents

All symbolic programming is based on **rules** – a set of generalized instructions that tells the computer how to transform one set of **tokens** into another. An assembler, *e.g.*, inputs a series of machine instructions in mnemonic form and outputs a series of numbers that represent the actual machine instructions in executable form. A FORTH compiler translates a definition into a series of addresses of previously defined objects. Even higher on the scale of complexity, a FORTRAN compiler inputs high-level language constructs formed according to a certain **grammar** and outputs an executable program in another language such as assembler, machine code or C.

What do rules have to do with scientific problem-solving? The crucial element in the rule-based style of programming is the ability to specify general **patterns** or even classes of patterns so

© Julian V. Noble 1992 – All rights reserved.

the computer can recognize them in the input and take appropriate action.

For example, in a modern high-energy physics experiment the rate at which events (data) impinge on detectors might be 10^7 discrete events per second. Since each event might be represented by 5-10 numbers, the storage requirements for recording the results of a search for some rare process, lasting 3-6 months of running time, might be 10^{16} bytes, or 10^7 high-capacity disk drives! Clearly, so much storage is out of the question and most of the incoming data must be discarded. That is, such experiments demand extremely fast filtering methods that can determine – in 10-20μsec – whether a given event is interesting. The criteria for "interesting" may be quite general and may need to be changed during the running of the experiment. In a word, they must be specified by some form of pattern recognition program rather than hard-wired.

Another area where pattern recognition helps the scientist is computer algebra. Closely related is the ability to translate mathematical formulae into machine code. So far we have stressed a FORTH programming style natural to that language, namely postfix notation, augmenting it primarily for readability or abstracting power. It cannot be denied, however, that sometimes it is useful simply to be able to write down a mathematical formula and have it translated automatically into executable form. This chapter develops the tools for symbolic programming and illustrates their use with a typical algebra program and a simple FORmula TRANslator.

§1 Rules

Before we can specify rules we need a language to express them in. We need to be able to describe the **grammar** of the rules in some way. The standard notation states rules as **regular expressions**[1]. The following rules describing some parts of FORTRAN illustrate how this works:

---

1. See A.V. Aho, R. Sethi and J.D. Ullman, *Compilers:...* (Addison-Wesley, Reading, 1988).

```
\ Rules for FORTRAN

\ NOTATION:
\ |          -> "or",
\ ^          -> "unlimited repetitions"
\ ^n         -> "0-n repetitions"
\ Q          -> "empty set"
\ &          -> + | -
\ %          -> * | /
\ <d>        -> "digit"

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

\ FORMULAS:
\ <assign>  -> <subj> = <expression>
\ <id>      -> <letter> {<letter> | <d>} ^6
\ <subject> -> <id> {<idlist> | Q}
\ <idlist>  -> ( <id> { , <id>} ^)
\ <arglist> -> ( <expr'n> { , <expr'n>} ^)
\ <func>    -> <id> <arglist>
\ <expr'n>  -> <term> | <term> & <expr'n>
\ <term>    -> <fctr> | <fctr> % <trm> | <fctr> ** <fctr>
\ <factor>  -> <id> | <fp#> | ( <expr'n> ) | <func>
```

We use angular brackets "<", ">" to set off "parts of speech" being defined, and arrows "->" to denote "is defined by". Other notational conventions, such as "|" to stand for "or", are listed in the "NOTATION" section of the rules list, mainly for mnemonic reasons. A statement such as

```
\ <int>      -> {-|Q}<d><d>^8
```

therefore means "an integer is defined by an optional leading minus sign, followed by 1 digit which is in turn followed by as many as 8 more digits". Similarly, the phrase

```
\ <assign> -> <subj> = <expression>
```

means "an assignment statement consists of a subject – a symbol that can be translated into an address in memory – followed by an equals sign, followed by an expression". Literal symbols – parentheses, decimal points, commas – are shown in **bold** type.

© Julian V. Noble 1992 – All rights reserved.

Note that some of these definitions are **recursive**. A statement such as

```
\ <expr'n> -> <term> | <term> & <expr'n>
```

seems to be defined in terms of itself. So it is a good bet the program that recognizes and translates a FORTRAN expression will be recursive, even if not explicitly so.

§2 Tools

In order to apply a rule stated as a regular expression, we need to be able to recognize a given pattern. That is, given a string, we need –say– to be able to state whether it is a floating point number or something else. We want to step through the string, one character at a time, following the rule

```
\ <fp#>     -> {-|Q}{{ d . | . d | d } d ^ exp't
```

This pattern begins with a minus or nothing, followed by a digit and a decimal point or a decimal point and a digit or a digit with no decimal point, followed by zero or more digits, then an exponent.

§§1 Pattern recognizers

One often sees pattern recognizers expressed as complex logic trees, *i.e.* as sequences of nested conditionals, as in Fig. 11-1 on page 263 below. As we see, the tree is already five levels deep, even though we have concealed the decisions pertaining to the exponent part of the number in a word **exponent?**. When programmed in the standard procedural fashion with **IF...ELSE...THEN** statements, the program becomes too long

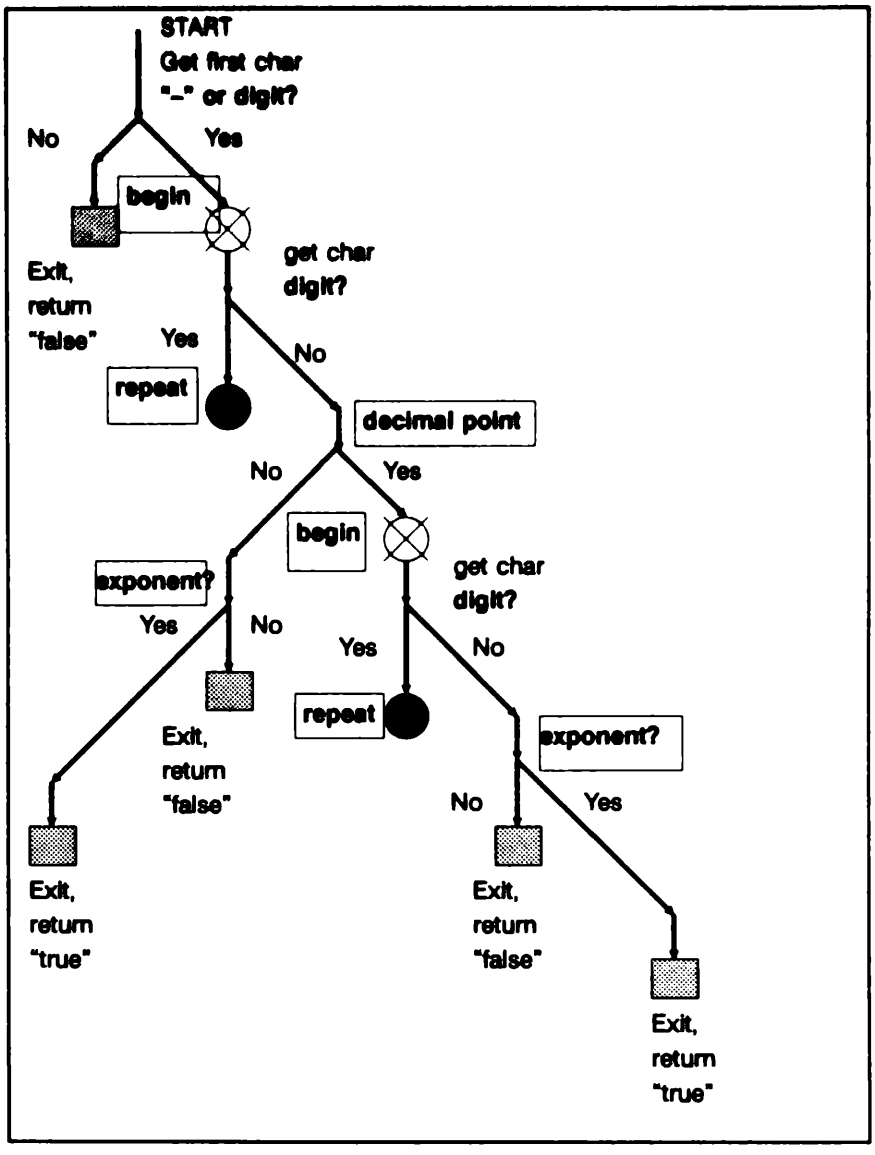

Fig. 11-1 *Logic tree for <floating point #>*

and too complex either for easy comprehension or for easy maintainance[2].

It has been known for many years that a better way to apply general rules —*e.g.*, to determine whether a given string conforms to the rules for "floating point number"— uses **finite state machines** (FSMs—we define them in §2 §§2 below). Here is an example, written in standard FORTH:

```
\ determine whether the string at $adr is a fp#
: skip-    ( adr - - adr')  DUP C@  ASCII - =  - ;
: skip_dp  ( adr - - adr')  DUP C@  ASCII . =  - ;
\ NOTE: these "hacks" assume "true" = -1.
: digit?   ( char - - f)    ASCII 9 ASCII 0  WITHIN ;
: skip_dig ( adr2 adr1 - - adr2 adr1')
   BEGIN   DDUP >  OVER C@  digit? AND
   WHILE       1+  REPEAT  ;  \ ... cont'd below
: dDeE?  ( char - - f)   95 AND     \ -> uppercase
   DUP ASCII D  =  SWAP  ASCII E  =  OR  ;

: skip_exponent ... ;     \ this definition shown below

: fp#?     ( $adr - - f)
   DUP  0 OVER COUNT + C!    \ add terminator
   DUP C@ 1+ OVER C!         \ count = count + 1
   COUNT OVER + 1- SWAP      ( - - $end $beg )
   skip- skip_dig  skip_dp  skip_dig
   skip_exponent
   UNDER =                   \ $beg' = $end?
   SWAP C@ 0=  AND ;         \ char[$beg'] = terminal?
```

The program works like this:

- Append a unique terminal character to the string.
- If the first character is "–" advance the pointer 1 byte, otherwise advance 0 bytes.

---

2. These defects of nested conditionals are generally recognized. Commercially available CASE tools such as Stirling Castle's *Logic Gem* (that translates logical rules to conditionals); and Matrix Software's *Matrix Layout* and AYECO, Inc.'s *COMPEDITOR* (that translate tabular representations of FSMs to conditionals in any of several languages) were originally developed as in-house aids.

- Skip over any digits until a non-digit is found.
- If that character is a decimal point skip over it.
- Skip any digits following the decimal point.
- A floating point number terminates with an exponent formed according to the appropriate rule (p. 261). **skip_exponent** advances the pointer through this (sub)string, or else halts at the first character that fails to fit the rule.
- Does the initial pointer ($beg') now point to the calculated end of the string ($end)? And is the last character (char[$beg']) the unique terminal? If so, report "true", else report "false".

We deferred the definition of **skip_exponent**. Using conditionals it could look like

```
: skip_exponent ( adr - - adr')
   DUP C@ dDeE? IF 1+ ELSE EXIT THEN
   skip- skip+
   DUP C@ digit? IF 1+ ELSE EXIT THEN
   DUP C@ digit? IF 1+ ELSE EXIT THEN
   DUP C@ digit? IF 1+ ELSE EXIT THEN ;
```

which,as we see in Fig. 11-2 below, has nearly as cor.voluted a logic tree as Fig. 11-1 on page 263 above.

§§2 Finite state machines

Just as we needed a FSM to achieve a graceful definition of **fp#?**, we might try to define **skip_exponent** as a state machine also. This means it is time to define what we mean by finite state machines. (We restrict attention to **deterministic** FSMs.) A **finite state machine**[3,4] is a program (originally it was a hard-wired switching circuit[5]) that takes a set of discrete, mutually exclusive

---

3. R. Sedgewick, Algorithms (Addison-Wesley Publishing Co., Reading, MA 1983) p. 257ff.
4. A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Tools and Techniques* (Addison Wesley Publishing Company, Reading, MA, 1988).
5. Zvi Kohavi, *Switching and Finite Automata Theory*, 2nd ed. (McGraw-Hill Publishing Co., New York, 1978).

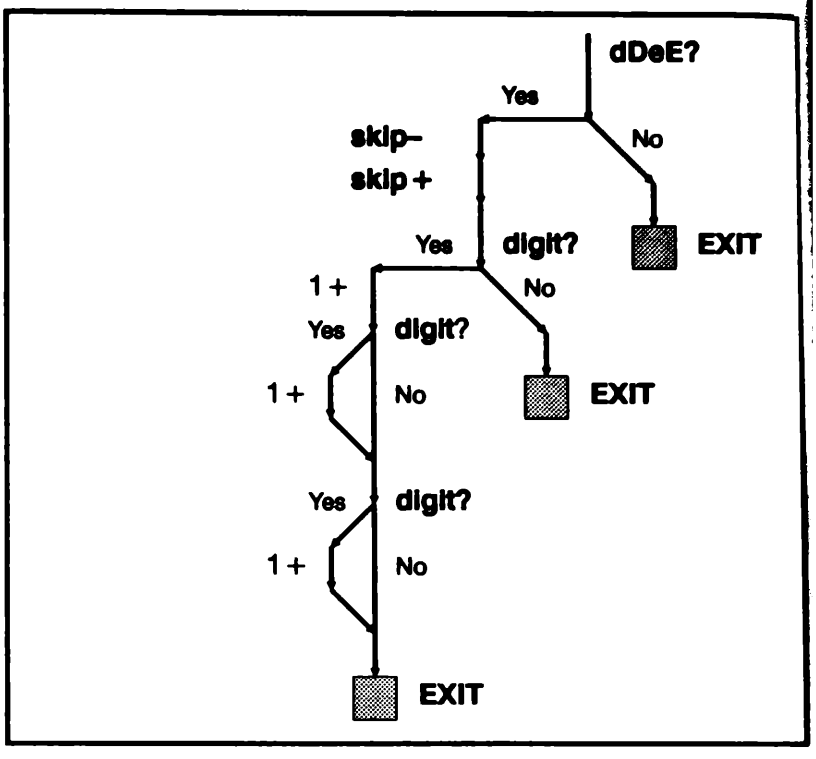

Fig. 11-2 *Logic tree for <exponent>*

inputs and also maintains a **state variable** that tracks the history of the machine's inputs. According to which state the machine is in, a given input will produce different results. The FSM program is most easily expressed in tabular form, as in Table 11-1, which we interpret as follows:

- each major column heading is an input.
- the inputs must be **mutually exclusive** and **exhaustive**; to exhaust all possibilities we include "other".
- each row represents the current state of the machine.
- each cell contains an action, followed by a state-transition.

Table 11-1 Example of finite state machine arrow (→) means "next state"

Input: / State ↓	other	→	dDeE	→	+/-	→	digit	→
0	Next	5	1+	1	Error	5	Next	5
1	Next	5	Next	5	1+	2	1+	3
2	Next	5	Next	5	Next	5	1+	3
3	Next	5	Next	5	Next	5	1+	4
4	Next	5	Next	5	Next	5	1+	5

The tabular representation of a FSM is much clearer than the logic diagram, Fig. 11-2. Since the inputs must be **mutually exclusive**[6] and **exhaustive**[7], there are *never* conditions that cannot be fulfilled —that is, leading to "dead" code— as frequently happens with logic trees (owing to human frailty). This means the chance of introducing bugs is reduced by FSMs in tabular form.

FORTRAN, BASIC or Assembler can implement FSMs with computed GOTOs. In BASIC, *e.g.*,

```
DEF SUB FSM (c$, adress%)
k% =0              ' convert input to column #
C$ = UCASE (c$)
IF C$ = "D" OR  C$ = "E" THEN k% = 1
IF C$ = "+" OR C$ = "-"  THEN k% = 2
IF ASC(C$) > = 48 AND ASC(C$) < = 57 THEN k% = 3
' cont'd
```

6. "Mutually exclusive" means only one input at a time can be true.
7. "Exhaustive" means every possible input must be represented.

© Julian V. Noble 1992 – All rights reserved.

```
                ' begin FSM proper
ON state% *3 + k% GOTO
   (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19)
0: state% = 5
1: adress% = adress% + 1 : state% = 1
2: CALL Error : state% = 5                    ' row 0
3: state% = 5
4: state% = 5
5: state% = 5
6: adress% = adress% + 1 : state% = 2         ' row 1
7: adress% = adress% + 1 : state% = 3
   ..... etc. .....

16: state% = 5
   ..... etc. .....                           ' row 4
19: adress% = adress% + 1 : state% = 5
END SUB
```

The advantage of FSM construction using computed GOTOs is simplicity; its disadvantage is the linear format of the program that hides the structure represented by the state table, 11-1. CASE statements –as in C, Pascal or QuickBASIC– are no clearer. We can use a state table for documentation, but the subroutine takes more-or-less the above form.

§§3 FSMs in FORTH

In the preceding FORTH example we *synthesized* the FSM from **BEGIN...WHILE...REPEAT** loops. FORTH's lack of line-labels and GOTOs (jumps) imposed this method, producing code as untransparent as the BASIC version. The Eaker CASE statement can streamline the program,

```
CASE: ADVANCE? NEXT 1+ ;CASE
: 1digit DUP C@ digit? ABS ADVANCE? ;
: skip_exponent ( adr - - adr')
   DUP C@ dDeE? IF 1+ ELSE EXIT THEN
   skip– skip+
   1digit 1digit 1digit ;
```

(three of four **IF...ELSE...THEN**s have been factored out and disguised as **CASE: ... ;CASE**), but this does not much improve clarity. We still need the state transition table to understand the program.

© Julian V. Noble 1992 – All rights reserved.

Various authors have tried to improve FSMs in FORTH using what amount to line-labels and GOTOs[8,9,10]. The resulting code is *less* elegant than the BASIC version shown above.

Whenever we reach a dead end, it is helpful to return to the starting point, restate the problem and re-examine our basic assumptions. One fact our preceding false starts make abundantly clear is that nowhere have we used the power of FORTH. Rather, our attempts merely imitated traditional languages in FORTH.

But FORTH is an endlessly protean language that lends itself to *any* programming style. Ideally FORTH relies on names so cunningly chosen that programs become self-documenting – readable at a glance.

Since state tables clearly document FSMs, it eventually occurred to me to let FORTH compile the state table – representing an FSM – directly to that FSM!

Compilation implies a **compiling word**; after some experimentation[11] I settled on the following usage[12]:

```
4 WIDE FSM:  (exponent)
\ input: | other      | dDeE       | +/-        | digit      |
\ state: - - - - - - - - - - - - - - - - - - - - - - - - - - -

  (0)  NEXT  >5    1+  >1    Error  >5    NEXT  >5
  (1)  NEXT  >5    NEXT  >5    1+  >2      1+  >3
  (2)  NEXT  >5    NEXT  >5  NEXT  >5      1+  >3
  (3)  NEXT  >5    NEXT  >5  NEXT  >5      1+  >4
  (4)  NEXT  >5    NEXT  >5  NEXT  >5      1+  >5 ;
```

Fig. 11-3 *Form of a FORTH finite state machine*

8. J. Basile, *J. FORTH Appl. and Res.* 1,2 (1982) 76-78.
9. E. Rawson, *J. FORTH Appl. and Res.* 3,4 (1986) 45-64.
10. D.W. Berrian, *Proc. 1989 Rochester FORTH Conf.* (Inst. for Applied FORTH Res., Inc., Rochester, NY 1989) p. 1-5.
11. The development process is described in detail in my article "Avoid Decisions", *Computers in Physics* 5,#4 (1991) 386.
12. The FORTH word **NEXT** is the equivalent of NOP in assembler.

© Julian V. Noble 1992 – All rights reserved.

The new defining word **FSM:** has a colon " : " in its name to remind us of its function. Its children clearly must "know" how many columns they have. The word **WIDE** reminds us the newly created FSMs incorporate their own widths.

The column labels and table headers are merely comments following " \ "; the state labels " **(0)** ", " **(1)** ", *etc.* arealso comments, delineated with parentheses. Their only purpose is readability.

The actions —**NEXT**, **Error**, **1+** —in Fig. 11-3 are obvious: they are simply previously-defined words. But what about the state transitions " **>1** ", " **>2** ", ... ? The easiest, most mnemonic and natural way to handle state transitions defines them as **CONSTANT**s

```
0 CONSTANT >0
1 CONSTANT >1
2 CONSTANT >2
   ... etc. ...
```

which are also actions to be compiled into the FSM. This follows the general FORTH principle that words should execute themselves[13].

In Chapter 5§2§§6 we used components of the compiler, particularly the **IMMEDIATE** word] ("switch to compile mode"), to create self-acting jump tables. We apply the same method here: The defining word **FSM:** will **CREATE** a new dictionary entry, build in its width (number of columns) using " , ", and then compile in the actions and state transitions as cfa's of the appropriate words.

The runtime code installed by **DOES>** provides a mechanism for finding the addresses of action and state transition corresponding to the appropriate input and current **state** (that is, in the cell of interest). Then the runtime code updates the **state**.

13. That is, we should not continually reinvent interpreters equivalent to the FORTH interpreter.

To alow nesting of FSMs (*i.e.*, compiling one into another), we incorporate the state variable for each child FSM within its data structure. This technique, using one extra memory cell per FSM, protects the state from accidental interactions, since if state has no name it cannot be invoked inadvertantly.

The FORTH code that does all this is

```
: 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
1  CONSTANT  >1
2  CONSTANT  >2
   ... etc. ...
```

We are now in a position to use the code for **(exponent)** (defined in Fig. 11-3 on page 269 above) to define the key word **skip_exponent** appearing in **fp#?**. The result is

```
: skip_exponent                 ( adr - - adr')
   ' (exponent)  0!             \ initialize state
   BEGIN   DUP  C@  DUP         ( - - adr char)
         dDeE?  ABS  OVER       \ input -> col#
         +/-?   2 AND    +  SWAP
         digit?   3 AND  +  ( - - adr col#)
   ' (exponent)  @              \ get state
   5  <                         \ not done?
   WHILE    (exponent)   REPEAT   ;
```

§§4 Automatic conversion tables

Our preceding example used logic to **compute** (*not* decide!) the conversion of input condition to a column number, *via*

```
( - - char)  dDeE?  ABS  OVER
         +/-?   2 AND    +  SWAP
         digit?   3 AND  +        ( - - col#)
```

© Julian V. Noble 1992 – All rights reserved.

When the input condition is a character, it is usually both faster and clearer to translate to a column number using a lookup table rather than tests and logic. That is, we can trade increased memory usage for speed. If a program needs many different pattern recognizers, it is worth generating their lookup tables *via* a defining word rather than crafting each by hand.

```
: TABLE:                        (- - #bytes )
   CREATE  HERE                 ( - - #bytes tab[0] )
   OVER  ALLOT                  \ allot #bytes in dictionary
   SWAP  0 FILL                 \ initialize to all 0's
   DOES>   +  C@  ;             ( n tab[0] - - [tab[n]] )

: install    ( col# adr char.n char.1 - - )      \ fast fill
   SWAP 1 + SWAP
   DO  DDUP  I  +  C!  LOOP  DDROP ;
```

Here is how we define a new lookup table:

```
128  TABLE:  [exp]              \ define 128-byte table
\ modify certain chars
\ Note: all unmodified chars return col# 0

1 ASCII d  ' [exp]   +   C!              \ col# 1
1 ASCII D  ' [exp]   +   C!
1 ASCII  e  ' [exp]  +   C!
1 ASCII E  ' [exp]   +   C!

2 ASCII  +   ' [exp]  +   C!             \ col# 2
2 ASCII  -   ' [exp]  +   C!

3   ' [exp]   ASCII 9   ASCII 0   install  \ col# 3
```

With the lookup table **[exp]**, **skip_exponent** becomes faster and more graceful,

```
: skip_exponent  ( adr - - adr')
    ' (exponent)  0!                    \ state = 0
   BEGIN   DUP    C@                    ( - - adr char)
           [exp]                        ( - - adr col#)
           ' (exponent) @               ( - - adr col# state)
           5  <                         \ not done?
   WHILE    (exponent)   REPEAT   ;
```

at a cost of 128 bytes of dictionary space. If dictionary space becomes tight, it would be perfectly simple to export the lookup

© Julian V. Noble 1992 – All rights reserved.

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

$$\gamma^\mu \gamma^\nu + \gamma^\nu \gamma^\mu = 2\eta^{\mu\nu}, \quad \mu, \nu = 0, \ldots, 3 \tag{1}$$

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

$$\eta^{\mu\nu} = \begin{array}{c} \mu\backslash\nu \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \end{array} \tag{2}$$

---

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. Berestetskii, E.M. Lifshitz and L.P. Pitaevskii, *Relativistic Quantum Theory, Part 1* (Pergamon Press, Oxford, 1971) p. 68ff.

© Julian V. Noble 1992 - All rights reserved.

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

$$\mathrm{Tr}(\not{A}) \hat{=} \sum_{k=1}^{N} A_{kk} \ . \tag{3}$$

Clearly, traces obey the **distributive law**

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

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

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

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

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

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

$$A\cdot B \equiv A^0 B^0 - A^1 B^1 - A^2 B^2 - A^3 B^3 \tag{6}$$

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

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

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.

© Julian V. Noble 1992 – All rights reserved.

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

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

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

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

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

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.

© Julian V. Noble 1992 – All rights reserved.

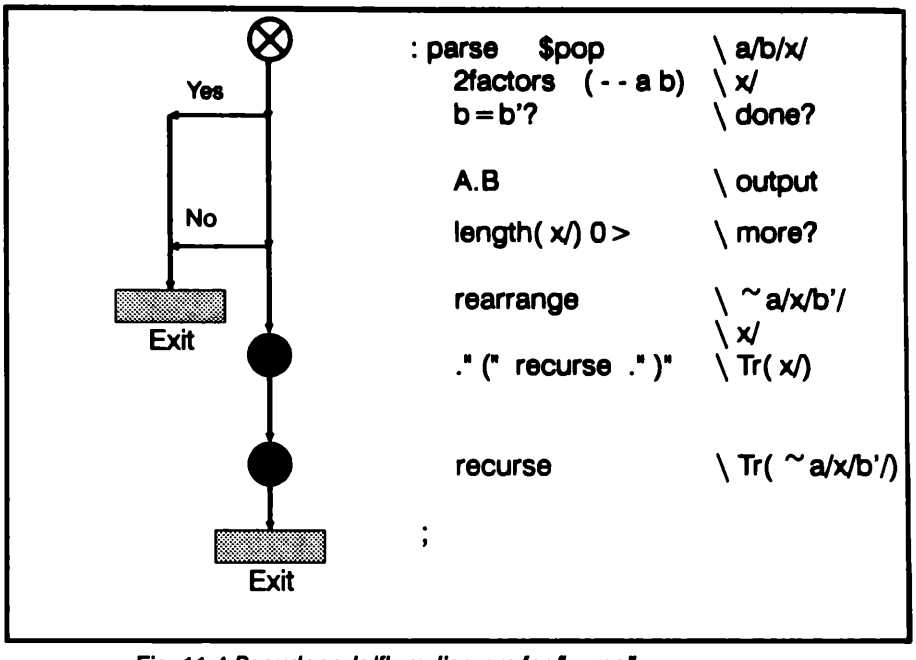

Fig. 11-4 *Pseudocode/flow diagram for "parse"*

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
```

© Julian V. Noble 1992 – All rights reserved.

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

© Julian V. Noble 1992 – All rights reserved.

ıput:	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'/
ıction(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.*,

$$\begin{aligned} &\mathrm{Tr}\left((\not{A} + m_A)(\not{B} + m_B)\gamma^5 \not{C} (\not{D} + m_D)\not{E} \ldots \right) \\ &\equiv \mathrm{Tr}\left(\gamma^5 \not{C} (\not{D} + m_D)\not{E} \ldots (\not{A} + m_A)(\not{B} + m_B) \right), \end{aligned} \qquad (12)$$

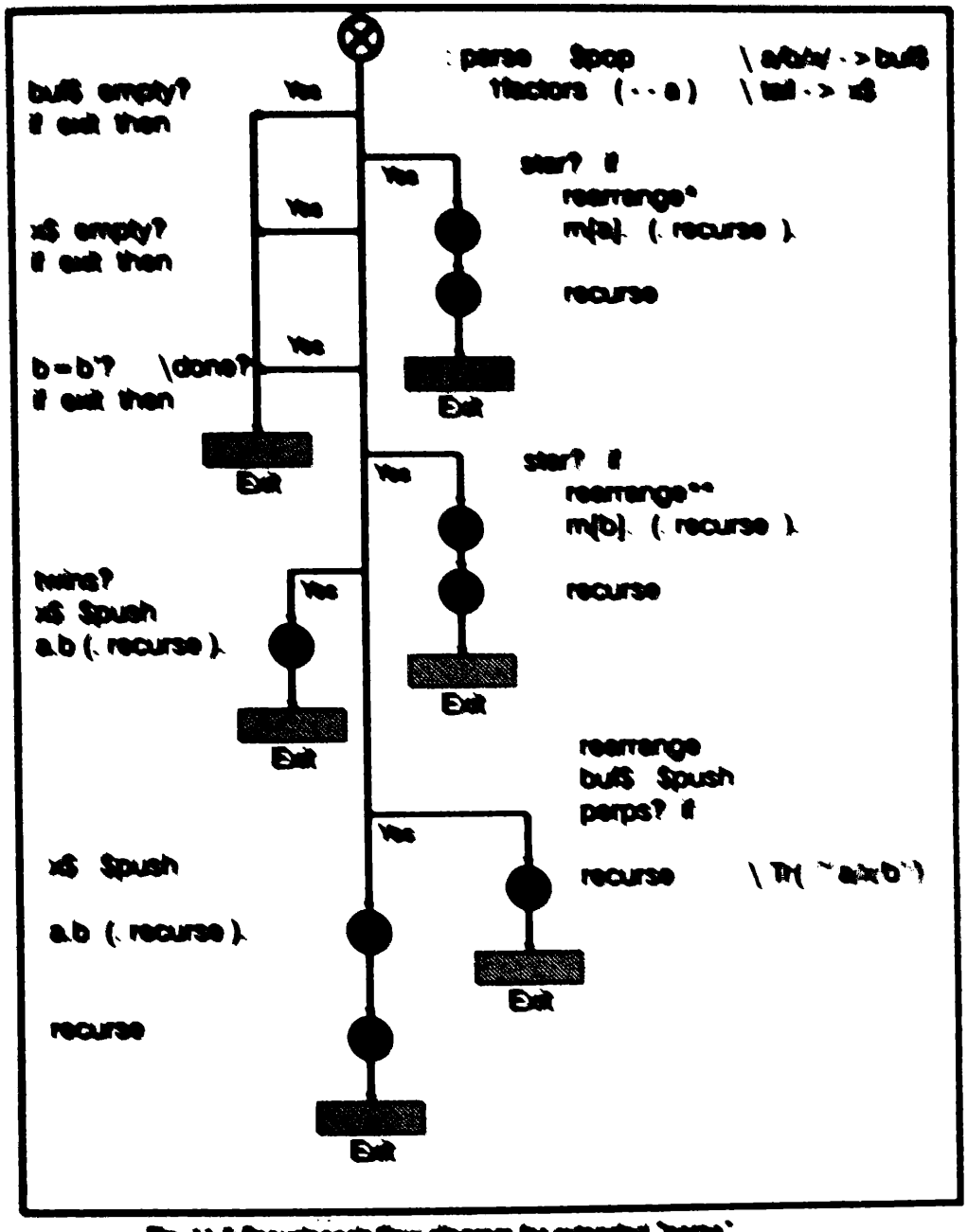

Fig 11-5 Pseudocode/flow diagram for extended 'parse'

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

© Julian V. Noble 1992 – All rights reserved

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

© Julian V. Noble 1992 – All rights reserved.

and factors. However, to reduce the number of parentheses, FORTRAN adopted a heirarchy of operators that has been followed by all other languages that incorporate semi-algebraic replacement statements like the above. The heirarchy is

0. FUNCTION
1. EXPONENTIATION (^ or **)
2. * or /
3. + or –
4. " , " (argument separator in lists)

The translator must both enforce these rules and resolve ambiguities involving operators at the same heirarchical level. Thus, *e.g.*, does the fragment

```
A/B*C
```

mean A/(B*C) or (A/B)*C ? Many FORTRAN compilers follow the latter convention, so we will maintain this tradition.

A second issue is the function library. The FORmula TRANslator must recognize functions, and be able to determine whether a given function is in the standard library. In the example above, F" recognized EXP and SIN as standard library functions and emitted the FORTH code to invoke them. A beauty of FORTH is that there are several easy ways to accomplish this, using components of the FORTH kernel.

A third issue is the ability of a true FORTRAN compiler to perform mixed-mode arithmetic, combining INTEGER*2, INTEGER*4, INTEGER*8, REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 types *ad libitem*. FORTRAN does this using the information contained in the type declarations at the beginning of a routine. A pure FORmula TRANslator has no such noncontextual information available to it, hence has no way to decide how to insert the proper FORTH words during compilation. To get around this we employ the generic data and operator conventions developed in Chapter 5 §1.

§§3 Parsing

Let us hand-parse the example, reproduced below:

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

Clearly, we must apply the first rule

```
\ <assignment>   -> <subject> = <expression>
```

embodied in the word **<assignment>**. We split at the " = " sign, and interpret the text to its left as a **SCALAR**. Since we want to emit the phrase A FS> last, yet have parsed it first, we have to hold it somewhere. Clearly the buffer where we store it will be a first-in last-out type; and by induction, last-in, first-out also. But a LIFO buffer is a stack. Hence the fundamental data structure needed in our parsing algorithm is a string stack. So we might imagine that after the first parsing step the string stack contains two strings

$STACK	Notes
A FS>	\ <subject>
-15.3E7*EXP(7/X) + Z/(W-SIN(THETA*PI/180)/4)	\ <expressio

Next we apply the rule

```
\ <expr'n>          -> <term> | < term> & <expr'n>
```

This breaks the top expression at the + sign between ") " and Z. We should think of the two terms

```
-15.3E7*EXP(7/X)
```

and

```
Z/(W-SIN(THETA*PI/180)/4)
```

as numbers on the ifstack; hence the code to evaluate each should be emitted before the addition operator (that is, these expressions are higher on the string stack than the addition operator **G+**). We adopt a rule that the right term is pushed before the left, so the $stack now looks like

© Julian V. Noble 1992 – All rights reserved.

$STACK	Notes
A FS>	\ <subject>
Z/(W-SIN(THETA*PI/180)/4) G+	\ <term>
-15.3E7*EXP(7/X)	\ <term>

We now anticipate a new problem: suppose we have somehow – no need to worry about details yet – emitted the code for the <term> **-15.3E7*EXP(7/X)** on top of the $stack. Then we would have to parse the line **Z/(W-SIN(THETA*PI/180)/4) G+** . Assuming the program knows how to handle the first part, **Z/(W-SIN(THETA*PI/180)/4)**, how will it deal with the **G+** ? We do not want to use the space as a delimiter (an obvious out) because this will cause trouble with **A FS>** .

The difficulty came from placing **G+** on the same line as **-15.3E7*EXP(7/X)**. What if we had placed the operator on the line above, as in

$STACK	Notes
A FS>	\ <subject>
G+	\ operator
Z/(W-SIN(THETA*PI/180)/4)	\ <term>
-15.3E7*EXP(7/X)	\ <term>

Eventually we see this merely exchanges one problem for another of equal difficulty: How do we distinguish a <factor> or <term> that contains no more operators or functions – and is therefore ready to be emitted as code – from the operator **G+**, which contains a "+" sign? Now we need complex expression recognition, which will lead to a slow, complicated program.

When this sort of impasse arises (and I am pretending it had been realized early in the design process, although the difficulty did not register until somewhat later) it signals that a key issue has been overlooked. Here, we failed to distinguish FORTH words, **FS>** and **G+**, from FORTRAN expressions. We have, in effect, mixed disparate data types (like trying to add scalars and vectors). Worse, we discarded too soon information that might

have been useful at a later stage. This leads to a programming tip, *a la* Brodie[23]:

TIP: *Never* discard information. You might need it later.

Phrased this way, the solution becomes obvious: keep the operators on a separate stack, whose level parallels the expressions. So we now envision an expression stack and an operator stack, which we call E/S and O/S for short. On two stacks,

E/S	O/S	Notes
A	FS>	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G+	\ <term>
-15.3E7*EXP(7/X)	NOP	\ <term>

Why the **NOP**s ("no operation") on the O/S? We want to keep the stack levels the same (so we do not have to check when POPping off code strings); we thus have to put **NOP** on the O/S to balance a string on the E/S.

The TOS now contains a <term>, so we apply the rules

\ <function> -> <id> <arglist>
\ <term> -> <factor> | <factor> **%** <term>
\ <factor> -> <id> | <fp#> | (<expr'n>) | <func>

We note there is an operator at the " **%** " priority level (the "*" in the TOS). We split the top <term> at this point, issuing a **G***.

E/S	O/S	Notes
A	FS>	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G+	\ <term>
EXP(7/X)	G*	\ <term>
-15.3E7	NOP	

23. Leo Brodie, *Thinking FORTH* (Prentice-Hall, Inc., NJ, 1984).

The parsing has now reached a turning point: the top string on the E/S can be reduced no further. The program must recognize this and emit the corresponding line of code (see Ch. 5):

```
% -15.3E7 REAL*4 F>FS
```

leaving

E/S	O/S	Notes
A	FS>	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G+	\ <term>
NULL	G*	
EXP(7/X)	NOP	\ <function>

What is **NULL** and why have we pushed it onto the E/S? Simply, it is not yet time to emit the **G*** so we have to save it; however, we have another operator, **G+**, to associate with Z/(W-SIN(THETA*PI/180)/4) . Thus we have no choice but to define a placeholder for the E/S, analogous to **NOP** on the O/S.

TOS now contains a function. Assuming we can recognize it as such, we want to check that it is in the library and put the correct operator on the E/S. Thus we want to decompose to

E/S	O/S	Notes
A	FS>	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G+	\ <term>
NULL	G*	
NULL	GEXP	\ <function>
(7/X)	NOP	\ <arglist>

The parentheses around the <arglist> on TOS serve no purpose, so drop them.

We see, once again, an operator of the priority-level **%** (the "/" between **7** and **X**), so we again apply the rule

\ <term> -> <factor> | <factor> **%** <term>

to obtain

E/S	O/S	Notes
A	FS >	\ < subject >
Z/(W-SIN(THETA*PI/180)/4)	G +	\ < term >
NULL	G*	
NULL	GEXP	
X	G/	\ < id >
7	NOP	\ < fp# >

Once again we can emit a number, so we do it:

```
% 7 REAL*8  F > FS
```

Wait! Why did we say REAL*4 with -15.3E7, but REAL*8 with 7 just now? Can't we make up our minds? The answer is that we want to respect precision over-rides *via* FORTRAN's E (single precision, so we say REAL*4) or D (double precision – REAL*8) exponent prefixes. However, where we are free to choose, it makes sense to keep maximum precision.

We continue, emitting the next simple items on the $stack:

```
X G/  GEXP  G*
```

leaving

E/S	O/S	Notes
A	FS >	\ < subject >
Z/(W-SIN(THETA*PI/180)/4)	G +	\ < term >

Once again we find the most exposed operator to be " / ", which we split with the rule

```
\ < term >   ->    < factor >  |  < factor >  %  < term >
```

E/S	O/S	Notes
A	FS >	\ < subject >
NULL	G +	\ < term >
(W-SIN(THETA*PI/180)/4)	G/	\ (< expr >)
Z	NOP	

Emit the TOS:

```
Z >FS
```

and apply the rule (first drop the parentheses)

```
\ <expr'n> ->    <term> | <term> & <expr'n>
```

E/S	O/S	Notes
A	FS>	\ <subject>
NULL	G+	\ <term>
NULL	G/	
-SIN(THETA*PI/180)/4	G+	
W	NOP	

Why did we issue **G+** and keep the leading "–" sign with **SIN**? Simple: any 9th grader can tell the difference between a "–" binary operator (**binop**) and a "–" unary operator (**unop**) in an expression. But, while not impossible, it is unnecessarily difficult to program this distinction. The FORTH philosophy is "Keep it simple!" Simplicity dictates that we embrace every opportunity to avoid a decision, such as that between "–" binop and "–" unop. The algebraic identity

$$X - Y \equiv X + (-Y)$$

lets us issue only **G+** , as long as we agree always to attach "–" signs as unops to the expressions that follow them. Eventually, of course, we shall have to deal with the distinction between negative literals (**–15.3E7**, *e.g.*) and negation of variables. The first we can leave alone, since the literal-handling word **%** ("treat the following number as floating point and put it on the 87stack") surely knows how to handle a unary "–" sign; whereas the second case will require us to issue a strategic **GNEGATE**.

A consequence of this method for handling "–" signs is that the compiler will resolve the ambiguous expression

$$-X^Y \stackrel{?}{=} -(X^Y) \text{ or } (-X)^Y$$

in favor of the former alternative. If the latter is intended, it must be specified with explicit parentheses.

After sending forth the phrase

```
W >FS
```

the leading "–" preceding **SIN(...)/4** must be dealt with. To preserve the proper ordering on emission we will want a word **LEADING–** that puts the token for **GNEGATE** on the O/S and moves the string **SIN(THETA*PI/180)/4** to the TOS, issuing a **NOP** on the E/S, obtaining

E/S	O/S	Notes
A	FS >	\ < subject >
NULL	G +	\ < term >
NULL	G/	
NULL	G +	
NULL	GNEGATE	
SIN(THETA*PI/180)/4	NOP	

The next exposed operator is at " **%** " -level. We apply **< term >** once more, to get:

E/S	O/S	Notes
...	...	\ ...
NULL	GNEGATE	
4	G/	
SIN(THETA*PI/180)	NOP	\ (< expr'n >)

After handling the function as before we find the successive stacks and FORTH code emissions

E/S	O/S	Notes
A	FS >	\ < subject >
NULL	G +	\ < term >
NULL	G/	
NULL	G +	
NULL	GNEGATE	
4	G/	
NULL	GSIN	
(THETA*PI/180)	NOP	

E/S	O/S	Notes
A	FS>	\ <subject>
NULL	G+	\ <term>
NULL	G/	
NULL	G+	
NULL	GNEGATE	
4	G/	
NULL	GSIN	
180	G/	
PI	G*	
THETA	NOP	

```
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
```

§§4 Coding the FORmula TRANslator

We proceed in the usual bottom-up manner. The first question is how to define the $stack. In the interest of brevity, I chose not to push the actual strings on the E/S, but rather pointers to their beginnings and ends. By using a token to represent the operator, we can define a 6-byte wide stack which will point to the text of interest (which itself resides in a buffer), and will hold the token for the operator at the current level. This way only one stack is pushed or popped and the levels never get out of synchronization.

Again at the lowest level, we can develop the components that recognize patterns, *e.g.*, whether a piece of text is a floating point number. The word that does the latter is **fp#?**, already described in §2§§1.

A function is defined by the rule

```
\ <arglist>      -> ( <expr'n> { , <expr'n> } ^ )
\ <function>     -> <id>  <arglist>
```

We may therefore identify a function by splitting at the first left parenthesis,

© Julian V. Noble 1992 – All rights reserved.

```
: >(      ( $end $beg - - $end $beg')       \ find first "("
   1- BEGIN 1+ DUPC@ ASCII ( =  >R
        DDUP = R> OR UNTIL ;
```

and then applying appropriately defined FSMs to determine whether the pieces are as they should be.

```
: <function> ( $end $beg - - f )
   DUP>R  >(      ( - - $end $beg')
   UNDER 1- R> <id>
   -ROT SWAP <arglist> AND ;
```

The FSM **<arglist>** must be smart enough to exclude cases such as

```
SIN(A+B)/(C-D)
```

and

```
SIN(A+B)/EXP(C-D)
```

that is, compound expressions that might contain functions; it must also correctly recognize, *e.g.*,

```
SIN((A+B)/EXP(C-D))
```

as a function.

In FORTH there is no distinction between library functions and functions we define ourselves. In either case, the protocol defined in Chapter 8 will work fine. Thus the code generator for **<function>** emits the code

```
USE( fn.name arg1 arg2 ... argN  F(x)
```

What, however, do we do about translating standard FORTRAN names such as SIN, COS and EXP to their generic FORTH equivalents? The simplest method defines words with the names of the FORTRAN library functions. The FORTH-83 word **FIND** locates the code-field address of a name residing in a string. Thus we could have (**note**: **.GSIN** is a **CONSTANT** containing a token)

```
.GSIN   CONSTANT  SIN                    \ etc.
```

© Julian V. Noble 1992 – All rights reserved.

```
: LIBRARY? ( $end $beg - - cfa | 0 )
   -ROT UNDER - DUP>R
   PAD 1+ SWAP CMOVE R> PAD C! \ make $
   PAD FIND ( - - cfa n) 0= NOT AND ;
```

now we can define **function!** which, assuming pointers to the <id> and <arglist> are on the stack, rearranges the $stack like this:

E/S	O/S	Notes
...	...	...
(arg1, arg2, ..., argN)	.F(X)	\ an op.
NULL	.lib_name	\ if library function

or, if it is a user-defined function, like this:

E/S	O/S	Notes
...	...	...
(arg1, arg2, ..., argN)	.F(X)	\ an op.
name	NOP	\ user function

The code that does all this is

```
: function! ( $end2 $beg2 $end1 $beg1 - - )
   .F(X) $push                    \ push arglist
   DDUP LIBRARY? DUP 0=
   IF    DROP .NOP $push          \ user fn
   ELSE EXECUTE                   \ in lib.
        NULL ROT $PUSH DDROP
   THEN ." USE( " ;
```

For the program itself[24] we work from the last word, **<assignment>**, to the first (which we do not know the name of yet). We shall describe the program in pseudocode only, in the interest of saving space. Clearly,

```
: <assignment>          \ input $ assumed in buffer
   <subj> = <expr>      \ split at " = " ( - - f )
   IF subj! THEN        \ put subj and its code on $stack
   .NOP $push ;         \ then put expr on $stack
```

24. File FTRAN.FTH on the program diskette.

Certain decisions need to be made here: for example, do we want **F"** to be able to parse an expression that is *not* an assignment (that is, generating code which evaluates the expression and leaves the result on the ifstack)? We allowed this with the **IF...THEN**.

Next we pseudocode **< expression >**:

```
: < expression >  empty? IF EXIT THEN
   $pop
   < trm > & < expr >          ( - - f)  \ split at &
   IF trm&expr!  RECURSE
   ELSE  NULL  ROT  $push
         .NOP  $push  < term >
   THEN  ;
```

Defined recursively in this way, **< expression >** will keep working on the TOS until it has broken it up into term s.

We similarly define **< term >** recursively, so it will break up any term s into all their factors. It should also recognize < arglist > s. Thus:

```
: < term >  empty? IF EXIT THEN
   $pop  < arglist >
   IF arglist!  < expression >  EXIT THEN
   < factor > % < term >        ( - - f ) \ split at % = "*/"
   IF     fct%trm!  RECURSE
   ELSE  .NOP $push  < factor >    \ term = factor
   THEN  ;
```

And finally, we define **< factor >**, again recursively,

```
: < factor >  empty? IF EXIT THEN          \ done
   $POP   < fp# >                          \ fp#?
         IF  fp#!  RECURSE  EXIT THEN
   leading -?
         IF  leading-!  < expression >     \ forward ref.
         EXIT THEN
   < id > IF  id!         < expression >     \ forward ref.
         EXIT THEN
   < f > ^ < f >                           \ exponent?
         IF  f^f!  RECURSE  RECURSE
         EXIT THEN
\ cont'd ...
```

© Julian V. Noble 1992 – All rights reserved.

```
< function >
        IF  function!  < expression >       \ forward ref.
        EXIT  THEN
( < expression > )              \ expression inside ( )?
        IF  expose!  < expression >       \ forward ref.
        ELSE  ." Not a factor!" ABORT  THEN  ;
```

Note the forward references found in **< factor >**; since **< expression >** is defined later, we must use vectored execution or some similar method to permit this recursive call.

With this we conclude our discussion of rule-based programming. The complete code for the FORmula TRANslator is too lengthy to print, hence it will be found on the included diskette.