Symbolic programming and computer algebra

One of the most revolutionary computer applications of recent years has been the development of programs that perform various algebraic and mathematical operations on symbols rather than on numbers. Programs such as REDUCE, SCHOONSCIP, MACSYMA®, DERIVE®, Mathematica® or Maple® have eliminated much of the tedium of lengthy paper-and-pencil manipulations, not to mention reducing the incidence of algebraic errors[1].

Symbolic programming is based on rules—sets of generalized instructions that tell 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 (as binary numbers) in executable form. Higher on the scale of complexity, a *compiler* inputs high-level language constructs formed according to specific rules (a "grammar") and outputs an executable program in another language such as assembler or machine code. To take but one example, the FORTRAN compiler recognizes (and translates them into machine language) text strings representing mathematical relations in quasi-algebraic form.

What do rules have to do with computational methods of physics? The crucial element in the rule-based style of programming is the ability to specify general patterns or even classes of patterns so 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 various particles impinge on detectors might be 10^7 discrete events per second. Since each such event might be represented by 5-10 numbers, the storage requirements for recording the results of a typical experiment, lasting 3-6 months of running time, might be 10^{16}–10^{17} bytes, or 10^6–10^7 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 to 20 μsec—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.

Since the philosophy of this book is that students should know what is inside the black boxes, we illustrate rule-based programming with some simple applications: a pared-down FORmula TRANslator[2] suitable for parsing integer expressions into Forth[3]; a simple program for manipulating the γ matrices of quantum field theory; and a program for symbolic differentiation.

1. R. Pavelle, M. Rothstein and J. Fitch, "Computer Algebra", *Scientific American* **245**, #6 (Dec. 1981) 136.
2. This is a pun—the language FORTRAN derives its name from "FORmula TRANslator".
3. A more sophisticated version for serious computation is found with the included programs.

1. Integer FORmula TRANslator

The evaluation of mathematical expressions is based on recognizing generalized patterns based on rules. First we state the rules for translating simple formulas.

Rules

To specify rules we need a language to express them in. That is, we need to be able to describe the grammar of the rules. One standard notation represents rules as *regular expressions*[4]. The following description of a reduced FORTRAN illustrate how this works:

```
\ Rules for integer FORmula TRANslator

\ NOTATION:
\ |       -> “or”,
\ ^       -> “unlimited repetitions”
\ ^n      -> “0-n repetitions”
\ Q       -> “empty set”
\ &       -> + | -
\ %       -> * | /

\ <integer>      -> { - | Q} {<digit> <digit>^8}

\ FORMULAS:
\ <id>           -> <letter> {<letter>|<digit>}^6
\ <assign>       -> <id> = <expression>
\ <expressionn>  -> <term> | <term> & <expression>
\ <term>         -> <factor>|<factor> % <term>
\ <factor>       -> <id> | integer | ( <expression> )
```

Angle brackets denote grammatical elements, such as `<expression>`; arrows `->` mean “is defined by”; other notational conventions, such as “|” to stand for “or”, are listed in the `NOTATION` section of the rules list for mnemonic convenenience. A statement such as

```
\ <integer>      -> {- | Q} {<digit> <digit>^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>       -> <id> = <expression>
```

means

> An assignment statement consists of an identifier—a symbol representing an address in memory—followed by an equals sign, followed by an expression.

---

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

Note that some of these definitions are recursive: a statement such as

```
\ <expressionn>  -> <term> | <term> & <expression>
```

seems to be defined in terms of itself. So it is a good bet that a program that recognizes and translates an expression can be written recursively[5].

Tools

To apply a rule stated as a regular expression, the program must be able to recognize a given pattern. That is, given a string, we need to be able to decide whether it is an integer or something else. We do this by stepping through the string one character at a time, following the rule

```
\ <integer>      -> {- | Q} {<digit> <digit>^8}
```

This pattern begins with a minus or nothing, followed by a digit and zero to eight more digits.

Pattern recognizers

One often sees pattern recognizers expressed as complex logic trees, *i.e.* as sequences of nested conditionals, as in the flow diagram on the next page. As we see, the tree is five levels deep, even though we have concealed the decisions pertaining to the exponent part of the number in a word `exponent?`. When programmed conventionally with `IF... ELSE... THEN` statements, the program becomes too complex for comprehension or maintainance[6].

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* (FSM). Here is an example, written in standard Forth:

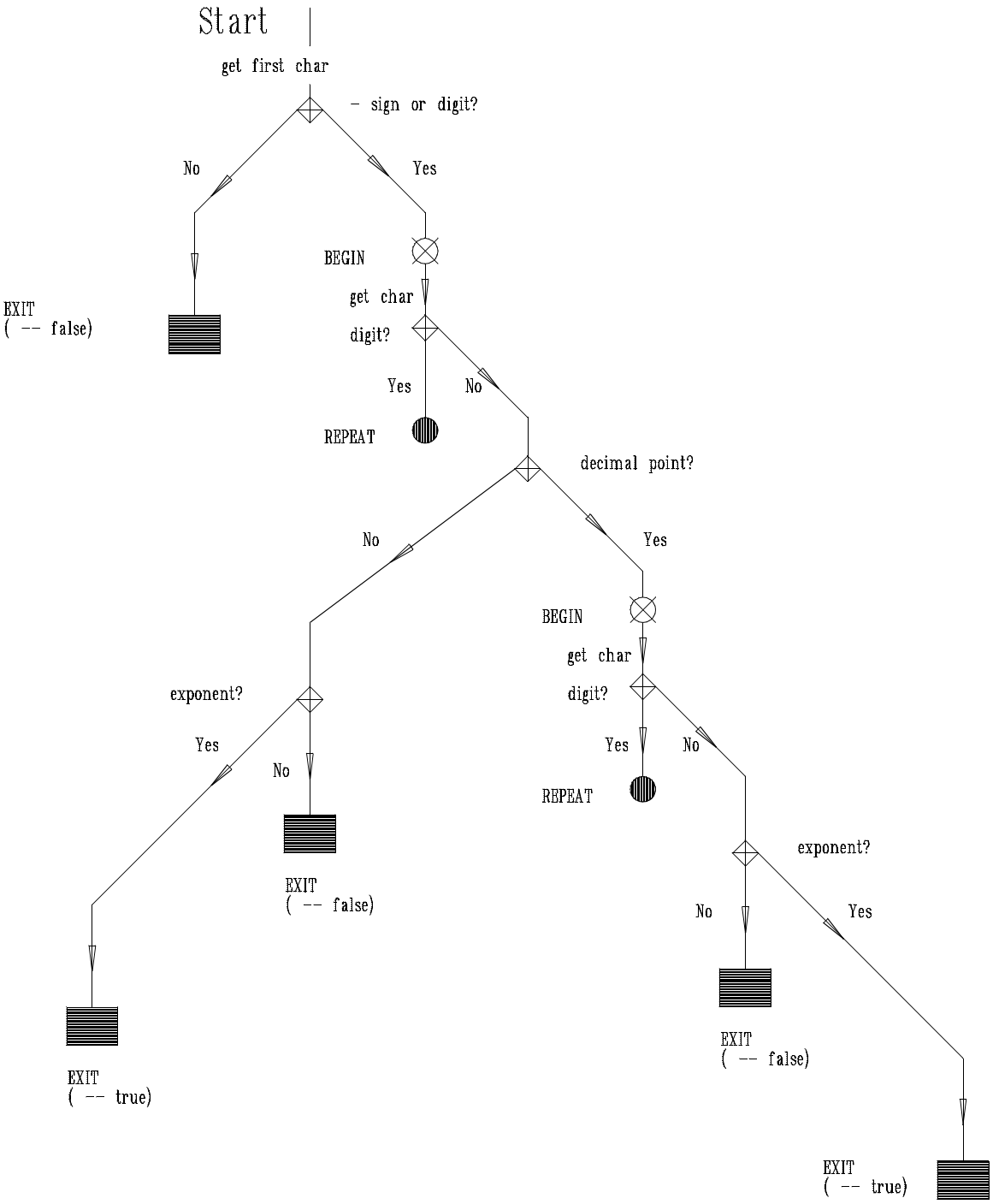
```
\ determine whether the string at $adr is a fp#
: skip_char  ( adr char -- adr')  OVER C@  =  ABS +  ;
\ assumes "true" = -1 or +1

: skip-   ( adr -- adr')    [CHAR] -  skip_char  ;
: skip_dp ( adr -- adr')    [CHAR] .  skip_char  ;

: digit?  ( char -- f)     [CHAR] 0   [ CHAR 9 1+ ] LITERAL   WITHIN  ;
: skip_dig   ( adr2 adr1 - -  adr2 adr1')
   BEGIN     2DUP  >   OVER  C@   digit?  AND
   WHILE     1+
   REPEAT    ;
```

---

5. By theorem a recursive program can always be written non-recursively. However this process often hides the structure of the program and makes it unnecessarily complex.
6. 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.


Start
get first char
- sign or digit?
No
Yes
EXIT
(-- false)
BEGIN
get char
digit?
Yes
No
REPEAT
decimal point?
No
Yes
BEGIN
get char
digit?
Yes
No
REPEAT
exponent?
Yes
No
EXIT
(-- false)
exponent?
No
Yes
EXIT
(-- false)
EXIT
(-- true)
EXIT
(-- true)

```
: 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                ( end beg' )
   TUCK                         ( beg' end beg' )
   =                            \ beg' = end ?
   SWAP  C@  0=   AND  ;        \ and last_char = 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.
- 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. `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 the unique terminal? If so, report "true", else report "false".

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

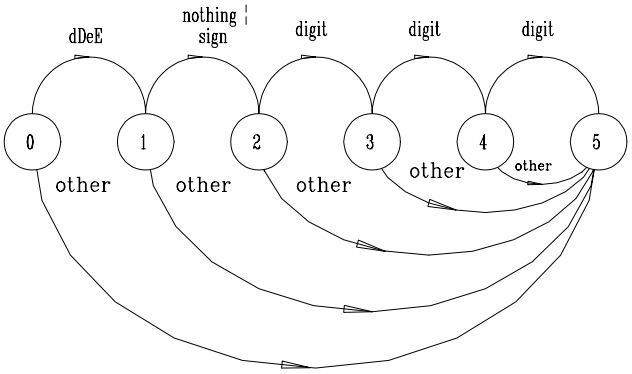
```
: skip+   ( adr -- adr')  [CHAR] +   skip_char  ;
: dDeE?   ( char -- f)
   >ucase                    \ change to uppercase
   [CHAR] D   [ CHAR E 1+ ]  WITHIN   ;
: skip_exponent  ( adr -- adr')
   DUP  C@  dDeE?      IF  1+  ELSE   EXIT  THEN
   skip-   skip+
   DUP  C@   digit?    IF  1+  ELSE   EXIT  THEN
   DUP  C@   digit?    IF  1+  ELSE   EXIT  THEN
   DUP  C@   digit?    IF  1+  ELSE   EXIT  THEN  ;
```

That is,

- skip an initial exponent significator, `dD` or `eE`
- skip an initial algebraic sign
- skip up to 3 digits (exponents can be up to 3 digits long).

Finite state machines

Just as we need 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. A finite state machine[7,8] is a program (originally it was a hard-wired switching circuit[9]) that takes a set of discrete, mutually exclusive, inputs and maintains a state variable that tracks the history of the machine's inputs. According to which state the machine is currently in, a given input will produce different results. The FSM program can be expressed either as a state transition diagram or as a transition table:



State transition diagram for `skip_exponent`

Tabular representation for finite state machine `skip_exponent`

state ↓ \ input:	other	dDeE	sign	digit
0 (initial)	noop → 5	1+ → 1	error → 5	error → 5
1	noop → 5	error → 5	1+ → 2	1+ → 3
2	noop → 5	error → 5	error → 5	1+ → 3
3	noop → 5	error → 5	error → 5	1+ → 4
4	noop → 5	error → 5	error → 5	1+ → 5
5 (terminal)				

(which I find clearer). In the tabular representation each cell contains two components: an action, followed by a transition to another state. The possible actions in this example are

```
noop      \ do nothing
1+        \ increment pointer
error     \ display an error message.
```

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

The tabular representation of a FSM is cleaner than nested logic. In a FSM the inputs must be mutually exclusive[10] and exhaustive[11], thereby eliminating conditions that cannot be fulfilled—that is, conditions leading to "dead" code—something that, owing to human frailty, frequently happens with nested logic. The incidence of bugs in tabular FSMs is much less than with logic trees.

FORTRAN, BASIC or Assembler can implement FSMs with computed **GOTO**s. In BASIC, *e.g.*, we could write

```
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
   ' 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 **GOTO**s is simplicity; its disadvantage is the linear format of the program, which hides the state table's structure. The Eaker **CASE** statement—a control structure available in C, Pascal, QuickBASIC or Forth—produces code that is no clearer. To achieve clarity with such tools we must write out the state table in comment form in a documentation section, thereby increasing the length (and decreasing the readability) of the source code.

However, extensible languages like Forth, Lisp and C++ permit us to define code that will create a FSM from its state-table representation. In Forth this looks like

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

```
4 wide fsm: (exponent)           \ begin fsm, giving # of columns
\ input   | other   |  dDeE   |   sign     | digit |
\ state   ----------------------------------------
   ( 0)   | noop >5 | 1+ >1   | err >5     | err >5
   ( 1)   | noop >5 | err >5  | 1+  >2     | 1+  >3  \ "nothing"
   ( 2)   | noop >5 | err >5  | err >5     | 1+  >3
   ( 3)   | noop >5 | err >5  | err >5     | 1+  >4
   ( 4)   | noop >5 | err >5  | err >5     | 1+  >5
;fsm                            \ terminate fsm
```

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

Putting it together

Once we have the tools for recognizing patterns in strings, we create the parts, bottom up, to translate a formula. The program given in Appendix A shows how this is done.

2. Computer algebra

The study of symbolic manipulation has led to rich new areas of pure mathematics[12]. 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 executed much faster on a slow, old PC than REDUCE (a symbolic manipulation program with built-in γ-matrix algebra features) did on a large mainframe, so the programming effort was definitely worthwhile.

Stating the problem

Dirac γ-matrices are 4×4 traceless, complex matrices defined by a set of (anti)commutation relations[13]. These are

---

12. See, *e.g.*, M. Mignotte, *Mathematics for Computer Algebra* (Springer-Verlag, Berlin, 1992).
13. 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.

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

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

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

In Feynman's notation we form matrices $\not{A}$ by taking the dot product of a 4-vector with a γ-matrix:

$$\not{A} \overset{df}{=} A_\mu \gamma^\mu \equiv A^0\gamma^0 - A^1\gamma^1 - A^2\gamma^2 - A^3\gamma^3\,.$$

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

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

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 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

$$\mathrm{Tr}(\not{A}\not{B}) \equiv \frac{1}{2}\left[\mathrm{Tr}(\not{A}\not{B}) + \mathrm{Tr}(\not{B}\not{A})\right] = \frac{1}{2}\mathrm{Tr}(\not{A}\not{B} + \not{B}\not{A}) \tag{5}$$

$$= \frac{1}{2}\,2\,A\cdot B\ \mathrm{Tr}(I) \equiv 4\,A\cdot B\,.$$

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

$$\mathrm{Tr}(\not{A}\not{B}\not{C}\not{D}) = 2A\cdot B\,\mathrm{Tr}(\not{C}\not{D}) - \mathrm{Tr}(\not{B}\not{A}\not{C}\not{D}) \tag{6}$$

$$\equiv 2A\cdot B\,\mathrm{Tr}(\not{C}\not{D}) - \mathrm{Tr}(\not{A}\not{C}\not{D}\not{B})$$

$$\equiv 4\left(A\cdot B\ C\cdot D - A\cdot C\ B\cdot D + A\cdot D\ B\cdot C\right)$$

We include Eq. 6 for testing purposes. The factors of 4 in Eq. 5 and 6 do nothing useful, so we might as well suppress them from the output, in the interest of a simpler program.

The rules

We apply formal rules analogous to those used in parsing a language. 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:

$$\not{B}\not{B} \equiv B \cdot B\,; \tag{7}$$

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 evaluate the trace of a product times the special γ-matrix[14]

$$\gamma^5 = i\,\gamma^0\gamma^1\gamma^2\gamma^3 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{8}$$

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

$$\gamma^5 \not{A}\not{B}\not{C} \equiv i\,\varepsilon_{\mu\nu\kappa\lambda} A^\mu B^\nu C^\kappa \gamma^\lambda\,, \tag{9}$$

where as usual, $i = \sqrt{-1}$.

A complete gamma matrix package will include 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
```

---

14. Note: Berestetskiĭ, *et al.* (*op. cit.*) define γ^5 with an overall "–" sign relative to Eq. 9.

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

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

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

```
: TEXT     WORD   HERE  PAD  $!  ;
: get$   [CHAR] )   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 $(N/2)!$.

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

```
: tr(
   get$          \ get a $ (terminated with ")" )
   setup         \ push $ on $stack
   do_trace   ;
```

The real work is done by `do_trace`, whose pseudocode looks like

```
: do_trace       ( $: a/b/x/ )
   $pop                \ pop the current sub-string off the $-stack
   2factors            \ get the 2 leftmost factors
   b=b'?               \ is the 2nd factor marked (i.e. has it come home)?
   IF EXIT THEN        \ output nothing
   a.b                 \ output dot product
   more?               \ is x/ <> "" (empty string)?
   IF  rearrange       ( $: ~a/x/b'  x/)
      ." (" RECURSE ." )" \ print a left-(, evaluate x/ and print a right-)
      RECURSE             \ evaluate the item left on the $-stack
   THEN  ;
```

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, 14–16 is the largest practicable number of factors. So if we make provision for expressions 20 factors long, that should be large enough for any practical purpose.

To simplify the manipulations and reduce the size of the string stack we will *tokenize* the input string. That is, each factor-name in the original will be represented by a 1-byte integer from 0 to 19h, *i.e.* smaller than 32d. This leaves the 5th, 6th and 7th bits free to serve as flags to indicate the properties of the factors. That is, we need to indicate whether a factor is "starred", *i.e.* whether it represents $(\not{A} + m_A I)$ or $\not{A}$, according to the rule

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

Again, we need to be able to indicate "redness", 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 "red".

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 "~".)

For safety we ought to inquire how deep the `$-stack` can get. That is, we certainly want to allow enough space that we will not overwrite program memory by having the stack get too deep. The rule

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

suggests that for an expression of length $n = 2k$, the maximum depth will be only $k + 1$. Thus we should plan for a stack depth of at least 11, perhaps 12 for safety. Since we are tokenizing the input, this means at most 240 bytes of memory, a trivial amount on today's computers.

Programming these aspects is simple enough that we need not dwell on it. The entire program appears in Appendix B below.

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 below.

$stack rearrangements and actions for posssible inputs					
input:	a/b/x/	*a/b/x/	a/*b/x/	a/a/x/	a/b/x/
resulting	~a/x/b/	a/b/x/	a/b/x/	…	…

$stack rearrangements and actions for posssible inputs					
$stack:	x/	b/x/	a/x/	x/	~a/x/b/
action(s)†:	*a.b* (RECURSE) RECURSE	*m_a* (RECURSE) RECURSE	*m_b* (RECURSE) RECURSE	*a.a* (RECURSE)	RECURSE

†Note: characters shown in ***italic*** are emitted, and `m_a` means m_a .

To avoid excessively convoluted logic we eschew nested branching constructs. A finite state machine would be ideal for clarity; however, as the above table makes clear, the logic is not really that of a FSM, besides which, the FSM compiler described above would have to be modified to keep its state variable on the stack, since otherwise it could not support recursion. The resulting pseudocode program is

```
: do_trace   ( --)
   $pop   empty?    IF  EXIT  THEN
   1factor   ( -- a)   \ tail -> x$
   star?     IF   REARRANGE*  .m_  (.  RECURSE  ).  RECURSE  EXIT   THEN
   x$ empty$ $=     IF  EXIT  THEN
   b=red?           IF  EXIT  THEN      \ done?
   star?     IF   REARRANGE**  .m_  (.  RECURSE  ).  RECURSE  EXIT   THEN
   twins?    IF   x$ $push   a.b  (.  RECURSE  ).   EXIT   THEN
   permute  buf$  $push
   perps?    IF   RECURSE   EXIT   THEN
   x$ $push   a.b  (.  RECURSE  ).  RECURSE
;
```

Implementing the code is now straightforward, so we omit the details, such as how to define `perp` to mark the symbols appropriately. 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

$$\mathrm{Tr}(\gamma^5 \not{A}\not{B}\not{C}\not{D}\ldots) \equiv i\,\mathrm{Tr}(\varepsilon_{\mu\nu\kappa\lambda} A^\mu B^\nu C^\kappa \gamma^\lambda \not{D}\ldots), \tag{10}$$

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

$$\begin{aligned}&\mathrm{Tr}((\not{A}+m_A)(\not{B}+m_B)\gamma^5\not{C}(\not{D}+m_D)\not{E}\ldots) \\ &\quad\equiv \mathrm{Tr}(\gamma^5\not{C}(\not{D}+m_D)\not{E}\ldots(\not{A}+m_A)(\not{B}+m_B)),\end{aligned} \tag{11}$$

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 202. This token is marked orthogonal to all three of the vectors it represents, at the time it is inserted.

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

The only other significant tasks are to extend the output routine to

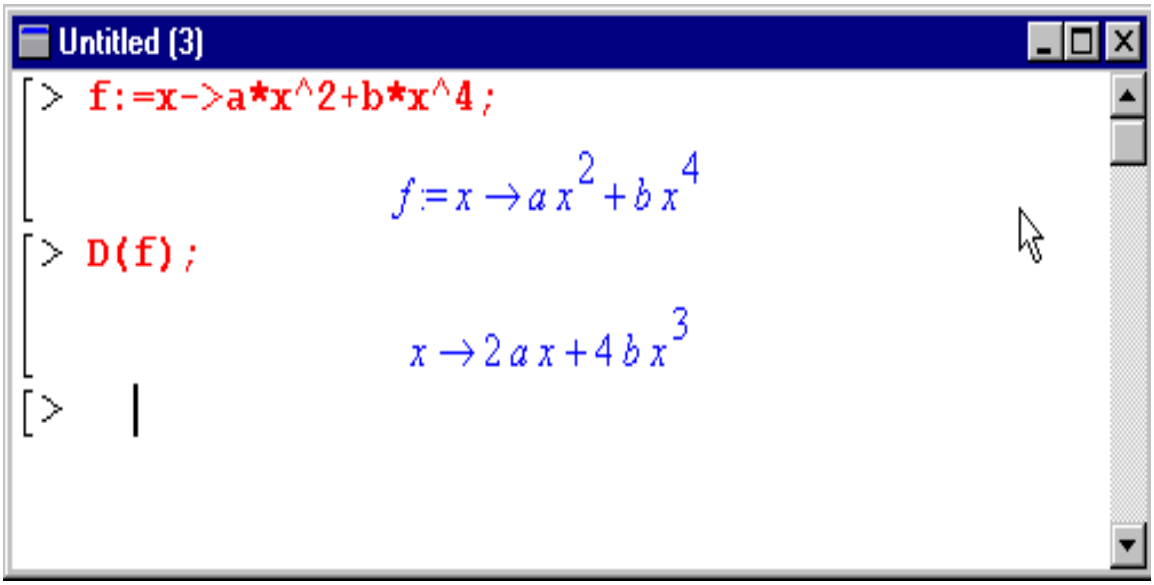
a) recognize the special "^" token; and

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

The program for all this is given in Appendix B.

3. Symbolic differentiation

As our third example of rule-based programming we consider computing the derivative of a function with respect to an independent variable. The screen snapshot below[15], of a window in the Maple® computer algebra system illustrates one common user interface for symbolic differentiation of a function.



Suppose we define the user interface by the rule

```
D(f,x)        -> Df = df/dx
```

Then the rules of differentiation are

```
D(f & g,x)    -> Df & Dg                \ distributive rule (& = + or -)
D(f * g,x)    -> Df * g + f * Dg        \ product rule
D(f/g,x)      -> Df / g - (f/g^2) * Dg  \ quotient rule
D(f(g),x)     -> D(f,g) * Dg            \ chain rule
D(f^g,x)      -> f^g * D(log(f) * g,x)  \ exponent rule

D(c,x)        -> 0                      \ specific functions
D(x,x)        -> 1
D(exp(x),x)   -> exp(x)
D(log(x),x)   -> x^(-1)
D(sin(x),x)   -> cos(x)
D(cos(x),x)   -> -sin(x)
etc.
```

15. …including cursor and mouse pointer.

Thus the program will employ recursion, and must be able to recognize the following elements:

```
expression      -> term & expression          \ & = + or -
term            -> factor % term              \ % = * or /
factor          -> constant                   \ does not depend on x
                -> x
                -> function                   \ does depend on x
                ->  factor ^ factor

function        -> id ( expression )
```

We see from the above that the differentiating function will share many of the elements of a FORmula TRANslator. The programming is therefore fairly straightforward, using ideas we have already encountered—pattern recognition and parsing. The code appears in Appendix X of this book.