Finite State Machines in Forth

J. V. Noble

Institute for Nuclear and Particle Physics
University of Virginia
Charlottesville, Virginia 22901

Abstract

This note provides methods for constructing deterministic and nondeterministic finite state automata in FORTH. The "best" method produces a one-to-one relation between the definition and the state table of the automaton. An important feature of the technique is the absence of (slow) nested `IF` clauses.

Introduction

Certain programming problems are difficult to solve procedurally even using structured code, but simple to solve using abstract finite state machines (FSMs) [2, 10]. For example, a compiler must distinguish a text string representing–say–a floating point number, from an algebraic expression that might well contain similar characters in similar order. Or a machine controller must select responses to pre-determined inputs that occur in random order.

Such problems are interesting because a program that responds to indefinite input is closer to a "thinking machine" than a mere sequential program. Thus, a string that represents a floating point number is defined by a set of rules; it has neither a definite length nor do the symbols appear in a definite order. Worse, more than one form for the same number may be permissible-user-friendliness demands a certain flexibility of format.

Although generic pattern recognition can be implemented through logical expressions (i.e. by concatenating sufficiently many `IF`s, `ELSE`s and `THEN`s) the resulting code is generally hard to read, debug, or modify. Worse, this approach is anything but structured, no matter how "prettily" the code is laid out: indentation can only do so much. And programs consisting mainly of logical expressions can be slow because many processors dump their pipelines upon branching [8]. These defects of the nested-`IF` approach are attested by the profusion of commercial tools[1] to overcome them: Stirling Castle's Logic Gem (that translates and simplifies logical expressions), Matrix Software's Matrix Layout (that translates a tabular representation of a FSM into one of several languages such as BASIC, Modula-2, Pascal or C), or AYECO, Inc.'s COMPEDITOR (that performs a similar translation).

FORTH is a particularly well-structured language that encourages natural, readable ways to generate FSMs. This note describes several high-level FORTH implementations. Finite state machines have been discussed previously in this journal [3, 9]. The present approach improves on prior methods.

A Simple Example

Consider the task of accepting numerical input from the keyboard. An unfriendly program lets the user enter the entire number before informing him that he typed two decimal points after the first digit. A friendly program, by contrast, refuses to recognize or display illegal characters. It waits instead for a legal character or carriage return (signifying the end of input). It permits backtracking, allowing erasure of incorrect input.

[1] These CASE tools were available at least as recently as 1993 from *The Programmer's Shop* and other developer-oriented software discounters.

To keep the example small, our number input routine allows signed decimal numbers without power-of-10 exponents (fixed-point, in FORTRAN parlance). Decimal points, numerals and leading minus signs are legal, but no other ASCII characters (including spaces) will be recognized. Here are some examples of legal numbers:

`0.123`, `.123`, `1.23`, `-1.23`, `123`, etc.

From these examples we derive the rules:

- Characters other than `0`–`9`, `-` and `.` are illegal.
- Numerals `0`–`9` are legal.
- The first character can be `-`, `0`–`9` or a decimal point.
- After the first character, `-` is illegal.
- After the first decimal point, decimal points are illegal.

A traditional procedural approach might look something like

```
VARIABLE PREVIOUS.MINUS?      \ history semaphores
VARIABLE PREVIOUS.DP?

: DIGIT?   ( c -- f)   ASCII 0  ASCII 9 WITHIN  ;      \ tests
: DP?      ( c -- f)   ASCII .  =  ;
: MINUS?   ( c -- f)   ASCII -  =  ;
: FIRST.MINUS?   MINUS?  PREVIOUS.MINUS?  @   NOT  AND   ;
: FIRST.DP?      DP?     PREVIOUS.DP?     @   NOT  AND   ;
: LEGAL?   ( c -- f)          \ horrible example
             DUP  DIGIT?
             IF    DROP  TRUE  DUP   PREVIOUS.MINUS?  !
             ELSE  DUP    FIRST.MINUS?
                   IF     DROP   TRUE  DUP   PREVIOUS.MINUS?  !
                   ELSE   FIRST.DP?
                          IF    TRUE  DUP   PREVIOUS.DP?  !
                          ELSE  FALSE
                          THEN
                   THEN
             THEN  ;
```

The word that does the work is (with apologies to Uderzo and Goscinny, creators of Asterix)

```
: Getafix
     FALSE  PREVIOUS.MINUS?  !    FALSE  PREVIOUS.DP?  !
                  \ initialize history semaphores
     BEGIN  KEY  DUP  CR <>   WHILE
         LEGAL?   IF   DUP  ECHO  APPEND   THEN
         REPEAT  ;
```

What makes this example—whose analogs appear frequently in published code in virtually every language—horrible? Each character whose legality is time-dependent requires a history semaphore. It is therefore difficult to tell by inspection that the word `LEGAL?`'s logic is actually incorrect, despite the simplification obtained by partial factoring and logical arithmetic.

Input State	OTHER?	DIGIT?	MINUS?	DP?
0	X → 0	E → 1	E → 1	E → 2
1	X → 1	E → 1	X → 1	E → 2
2	X → 2	E → 2	X → 2	X → 2

Figure 1: State table summarizing the rules for fixed point numbers.
E stands for "echo" (to the CRT) and X for "do nothing".

Forth Finite State Machines

The FSM approach replaces the true/false historical semaphores with one state variable. The rules can be embodied in a state table that expresses the response to each possible input in terms of a concrete action and a state transition, as shown below in Figure 1.

In the state table,

- The illegality of "other" characters is expressed by the uniform action X and the absence of state transitions.
- The special status of the first character is expressed by the fact that all acceptable characters lead to transitions out of the initial state (0).
- An initial - sign or digit leads to state 1, where a - sign is unacceptable.
- A decimal point always moves the system to state 2, where decimal points are not accepted.

While some FSMs can be synthesized with `BEGIN...WHILE...REPEAT` or `BEGIN...UNTIL` loops, keyboard input does not readily lend itself to this approach. We now explore three implementations of the state table of Figure 1 as FORTH FSMs.

Brute-Force FSM

The "brute-force" FSM uses the Eaker `CASE` statement, either in its original form [6] or with a simplified construct from HS/FORTH [7]. HS/FORTH provides defining words `CASE: ;CASE` whose daughter words execute one of several words in their definition, as in:

```
CASE: CHOICE   WORD0  WORD1  WORD2  WORD3 ... WORDn  ;CASE
3 CHOICE   ( executes WORD3 )
```

HS/FORTH's `CASE: ...;CASE` incurs virtually no run time speed penalty relative to executing the words themselves. Now, how do we use `CASE: ...;CASE` to implement a FSM? First we need a state variable (initialized to 0) that can assume the values 0, 1 and 2. To test whether an input character is a numeral, minus sign, decimal point or "other", we define `WITHIN` as used here returns `TRUE` if $a \leq n \leq b$, which is different from the ANS specification[2].

```
VARIABLE mystate    mystate 0!
: WITHIN   ( n a b -- f) DDUP MIN  -ROT  MAX  ROT
             UNDER MIN  -ROT MAX  = ;
: DIGIT?  ( c -- f )   ASCII 0  ASCII 9 WITHIN  ;
: DP?     ( c -- f )   ASCII .  = ;
: MINUS?  ( c -- f )   ASCII -  = ;
```

Now, to use `CASE: ;CASE` we define 3 words to handle the tests in each state:

[2]The ANSI Standard [12] renames `ASCII` to `CHAR` and `UNDER` to `TUCK`; also `DDUP` is specific to HS/FORTH and should be replaced with `2DUP` for ANSI compliance.

```
: (0)    ( char -- )   DUP
         DIGIT?   OVER   MINUS?   OR
         IF   EMIT   1 mystate !   ELSE   DUP   DP?
         IF   EMIT   2 mystate !   ELSE   DROP   THEN    THEN  ;

: (1)    ( char -- )   DUP   DIGIT?
         IF   EMIT   1 mystate !   ELSE   DUP  MINUS?
         IF   1 mystate !          ELSE   DUP   DP?
         IF   EMIT   2 mystate !   ELSE   DROP
         THEN    THEN     THEN  ;

: (2)    ( char -- )   DUP
         DIGIT?   IF   EMIT   ELSE   DROP   THEN   ;
```

Finally, we define the words that use the above:

```
CASE:   <Fixed.Pt>    (0)  (1)  (2)   ;CASE

: Getafix    0 mystate !                          \ initialize state
             BEGIN
                 KEY   DUP   13 <>                \ not CR ?
             WHILE   mystate @   <Fixed.Pt>       \ execute FSM
             REPEAT  ;
```

A Better FSM

While the approach outlined above in *Brute-Force FSM* (essentially the method described recently by Berrian [4]) both works and produces much clearer code than the binary logic tree of *A Simple Example*, it nevertheless can be improved. The words `(0)`, `(1)` and `(2)` are inadequately factored (they contain the tests performed on the input character). They also contain `IF...ELSE...THEN` branches (which we prefer to avoid for the sake of speed and structure). Finally, each FSM must be hand crafted from numerous subsidiary definitions.

We want to translate the state table in Figure 1 into a program. The preceding attempt was too indirect—each state was represented by its own word that did too much. Perhaps we can achieve the desired simplicity by translating more directly. In FORTH such translations are most naturally accomplished via defining words. Suppose we visualize the state table as a matrix, whose cells contain action specifications (addresses or execution tokens), whose columns represent input categories, and whose rows are states. If we translate input categories to column numbers, the category and the current value of the state variable (row index) determine a unique cell address, whose content can be fetched and executed.

Translating the input to a column number factors the tests into a single word that executes once per character. This word should avoid time-wasting branching instructions so all decisions (as to which cell of the table to `EXECUTE`) will be computed rather than decided. For our test example, the preliminary definitions are:

```
VARIABLE mystate   0 mystate !

: WITHIN   ( n a b -- f) DDUP MIN  -ROT  MAX
             ROT TUCK MIN  -ROT MAX  = ;

: DIGIT?  ( n -- f )   ASCII 0  ASCII 9 WITHIN  ;
```

```
: DP?  ASCII .  = ;

: MINUS?  ASCII -  = ;
```

and the input translation is carried out by

```
: cat->col#   ( n -- n')
        DUP   DIGIT?   1 AND                    \ digit   -> 1
        OVER  MINUS?   2 AND  +                 \ -       -> 2
        SWAP  DP?      3 AND  +                 \ dp      -> 3
;                                               \ other   -> 0
```

Now we must plan the state-table compiler. In general, we define an action word for each cell of the table that will perform the required action and state change. At compile time the defining word will compile an array of the execution addresses (execution tokens in ANS-FORTH parlance [11]) of these action words. At run time the child word computes the address of the appropriate matrix cell from the user-supplied column number and the current value of `mystate`, fetches the execution address from its matrix cell, and `EXECUTE`s the appropriate action. Since a table can have arbitrarily many columns the number of columns must be supplied at compile time. These requirements lead to the definitions:

```
: TUCK       COMPILE UNDER  ;                   \ ANS compatibility
: WIDE   ;                                      \ NOOP for clarity
: CELLS      COMPILE  2*  ;                     \ ANS compatibility
: CELL+      COMPILE  2+  ;                     \ ANS compatibility
: PERFORM    COMPILE @   COMPILE EXECUTE  ;     \ alias
: FSM:       ( width -- )  CREATE  ,   ]
       DOES>  ( n adr -- )
              TUCK  @  mystate @  *   +   CELLS   CELL+  +
              ( adr')  PERFORM  ;
```

Here `CREATE` makes a new header in the dictionary, `,` stores the top number on the stack in the first cell of the parameter field, and `]` switches to compile mode. The run time code computes the address of the cell containing the vector to the desired action, fetches that vector and executes the action[3].

Now we apply this powerful new word to our example problem. From Figure 1 we see that transitions (changes of state) occur only in cells (0,0), (0,1), (0,2), (1,0), and (1,2). These are always associated with `EMIT` (E in the Figure). No change of state accompanies a wrong input and the associated action is to `DROP` the character. There are thus only 2 distinct state-changing actions we need define:

```
: (00)   EMIT  1 mystate ! ;
: (02)   EMIT  2 mystate ! ;
```

Since we must test for 4 conditions on the input character, the state table will be 4 columns wide:

```
4 WIDE FSM: <Fixed.Pt#>   ( action# -- )
\            other   num    -     .      \ state
             DROP    (00)  (00)  (02)    \ 0
             DROP    (00)  DROP  (02)    \ 1
             DROP    (02)  DROP  DROP  ; \ 2
```

The word that does the work is

[3]This simple and elegant implementation only works with indirect-threaded FORTHs. An ANS Standard alternative is provided in the Appendix.

```
: Getafix   0 mystate !   BEGIN   KEY   DUP   13 <>   \ not CR
         WHILE   DUP   cat->col#   <Fixed.Pt#>   REPEAT   ;
```

We can immediately test the FSM, as follows:

```
FLOAD F:X.1 Loading F:X.1  ok
ASCII 3 cat->col# .  1  ok
ASCII 0 cat->col# .  1  ok
ASCII - cat->col# .  2  ok
ASCII . cat->col# .  3  ok
ASCII A cat->col# .  0  ok
: Getafix   0 mystate !   BEGIN  KEY  DUP  13 <>  WHILE
    DUP   cat->col#   <Fixed.Pt#>   REPEAT  ;
  Getafix -3.1415975 ok
Getafix 55.3259 ok
```

The incorrect input of excess decimal points, incorrect minus signs or non-numeric characters does not show because, as intended, they were dropped without echoing to the screen.

An Elegant FSM

The defining word `FSM:` of *A Better FSM*, while useful, nevertheless has room for improvement. This version hides the state transitions within the action words compiled into the child word's cells. A more thoroughly factored approach would explicitly specify transitions next to the actions they follow, within the definitions of each FSM. Definitions will become more readable since each looks just like its state table; that is, our ideal FSM definition will look like

```
4 WIDE FSM: <Fixed.Pt#>
\ input:  |  other?  |  num?   |  minus?  |   dp?     |
\ state:  ------------------------------------------
   ( 0 )     DROP  0    EMIT  1   EMIT  1     EMIT  2
   ( 1 )     DROP  1    EMIT  1   DROP  1     EMIT  2
   ( 2 )     DROP  2    EMIT  2   DROP  2     DROP  2 ;
```

so we would never need the words `(00)` and `(02)`. Fortunately it is not hard to redefine `FSM:` to include the transitions explicitly. We may use `CONSTANT`s to effect the state transitions:

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

and modify the run time portion of `FSM:` accordingly:

```
: FSM:   ( width -- )   CREATE  ,  ]   DOES>   ( col# -- )
    TUCK   @                                   ( -- adr col# width )
    mystate @  *  +  2*  CELLS  CELL+  +       ( -- offset )
    DUP   CELL+  PERFORM   mystate !   PERFORM   ;
```

Note that we have defined the run time code so that the change in state variable precedes the run time action. Sometimes the desired action is an `ABORT` and an error message. Changing the state variable first lets us avoid having to write a separate error handler for each cell of the FSM, yet we can tell where the `ABORT` took place. If the `ABORT` were first, the state would not have been updated.

The FSM is then defined as

```
4 WIDE FSM: <Fixed.Pt#>
\ input:  |  other?  |  num?   |  minus?  |   dp?     |
\ state:  ------------------------------------------
   ( 0 )     DROP >0    EMIT >1   EMIT >1     EMIT >2
   ( 1 )     DROP >1    EMIT >1   DROP >1     EMIT >2
   ( 2 )     DROP >2    EMIT >2   DROP >2     DROP >2 ;
```

which is clear and readable. Of course one could define the runtime (DOES>) portion of FSM: to avoid the need for extra CONSTANTs >0, >1, etc. The actual numbers could be stored in the table via:

```
4 WIDE FSM: <Fixed.Pt#>
\ input:  |   other?    |   num?    |   minus?   |    dp?     |
\ state:  ---------------------------------------------------
   ( 0 )   DROP [ 0 , ]  EMIT [ 1 , ] EMIT [ 1 , ] EMIT [ 2 , ]
   ( 1 )   DROP [ 1 , ]  EMIT [ 1 , ] DROP [ 1 , ] EMIT [ 2 , ]
   ( 2 )   DROP [ 2 , ]  EMIT [ 2 , ] DROP [ 2 , ] DROP [ 2 , ] ;
```

However, for reasons given in *The Best FSM So Far*, it is better to implement the state transition with CONSTANTs rather than with numeric literals.

The Best FSM So Far

Experience using the FSM approach to write programs has motivated two further improvements. First, suppose one needs to nest FSMs, i.e., to compile one into another, or even to RECURSE. The global variable mystate precludes such finesse. It therefore makes sense to include the state variable for each FSM in its data structure, just as with its WIDTH. This modification protects the state of a FSM from any accidental interactions at the cost of one more memory cell per FSM, since if the state has no name it cannot be invoked. Second, suppose one or other of the action words is supposed to leave something on the stack, and that, for some reason, it is desirable to alter the state after the action rather than before (this is, in fact, the more natural order of doing things). Since there is no way to know in advance what the stack effect will be, we use the return stack for temporary storage, to avoid collisions. The revision is thus:

```
: 2@     COMPILE  D@  ;   \ alias; D@ is ok in ANS
: WIDE   0 ;
: FSM:   ( width 0 -- )
      CREATE   ,  ,   ]
      DOES>               ( col# adr -- )
         DUP >R  2@  *  + ( -- col#+width*state )
         2*  2+  CELLS    ( -- offset-to-action)
         DUP >R           ( -- offset-to-action)
         PERFORM          ( ? )
         R> CELL+         ( -- offset-to-update)
         PERFORM          ( -- state')
         R> !   ;         \ update state
```

The revised keyboard input word of our example is

```
: Getafix   0  ' <Fixed.Pt#> !
            BEGIN   KEY   DUP   13 <>      WHILE
            DUP   cat->col#  <Fixed.pt#>   REPEAT ;
```

Note that the state variable is initialized to zero because we know it is stored in the first cell in the parameter field of the FSM which can be accessed by the phrase `' <Fixed.Pt#>` (or the equivalent in the FORTH dialect being used—see the *Appendix*).

Nondeterministic Finite State Machines

We are now in a position to explain why defining a `CONSTANT` to manage the state transitions is better than merely incorporating the next state's number. First, there is no obvious reason why states cannot be named rather than numbered. The FORTH outer interpreter itself is a state machine with two states, `COMPILE` and `INTERPRET`; i.e., names are often clearer than numbers.

The use of a word rather than a number to effect the transition permits a more far-reaching modification. Our code defines a compiler for deterministic FSMs, in which each cell in the table contains a transition to a definite next state. What if we allowed branching to any of several next states, following a given action? FSMs that can do this are called nondeterministic. Despite the nomenclature, and despite the misleading descriptions of such FSMs occasionally found in the literature [5], there need be nothing random or "guesslike" about the next transition. What permits multiple possibilities is additional information, external to the current state and current input.

Here is a simple nondeterministic FSM used in my FORmula TRANslator [1]. The problem is to determine whether a piece of text is a proper identifier (that is, the name of a variable, subroutine or function) according to the rules of FORTRAN. An *id* must begin with a letter, and can be up to seven characters long, with characters that are letters or digits. To accelerate the process of determining whether an ASCII character code represents a letter, digit or "other", we define a decoder (fast table translator):

```
: TAB:   ( #bytes -- )
           CREATE   HERE  OVER  ALLOT   SWAP  0 FILL  DOES>  +  C@  ;
```

and a method to fill it quickly:

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

The translation table we need for detecting *id*'s is

```
128 TAB: [id]
1 ' [id]  ASCII Z  ASCII A  install
1 ' [id]  ASCII z  ASCII a  install
2 ' [id]  ASCII 9  ASCII 0  install
```

This follows the F79 convention that `'` returns the *pfa*. To convert to F83 or ANS replace `'` by `' >BODY .`

Thus, e.g.

```
ASCII R [id] .  1  ok
ASCII s [id] .  1  ok
ASCII 3 [id] .  2  ok
ASCII + [id] .  0  ok
\ to convert to ANSI replace ASCII by CHAR
```

Now how do we embody the id rules in a FSM? Our first attempt might look like

```
3 WIDE FSM: (id)
\ input:     |  other  |  letter   |    digit    |
\ state      ------------------------------------
   ( 0 )       NOOP >8     1+ >1       NOOP >2
   ( 1 )       NOOP >8     1+ >2       1+   >2
```

```
   ( 2 )       NOOP >8     1+ >3       1+   >3
   ( 3 )       NOOP >8     1+ >4       1+   >4
   ( 4 )       NOOP >8     1+ >5       1+   >5
   ( 5 )       NOOP >8     1+ >6       1+   >6
   ( 6 )       NOOP >8     1+ >7       1+   >7
   ( 7 )       NOOP >8   NOOP >8       NOOP >8  ;

: state<     [COMPILE]  '   LITERAL  ; IMMEDIATE

\ compile address of "state" cell of a FSM
\ for F83  or ANS replace  '  with  ' >BODY

: <id>   ( $end $beg -- f)                  \ f = T for id, F else
         0  state< (id) !                   \ initialize state to 0
         BEGIN   DUP C@  [id]  (id)         \ run fsm
                 DDUP  >                    \ $end > $beg ?
                 state<  (id) @  2 <        \ not terminated ?
                 AND                        \ combine flags
         WHILE   1+                         \ $beg = $beg+1
         REPEAT   DDROP                     \ finish loop, clean up
         state< (id)  @   8 <  ;            \ leave flag
```

To make sure the *id* is at most 7 characters long we had to provide 8 states. The table is rather repetitious. A simpler alternative uses a `VARIABLE` to count the characters as they come in. The revision is

```
VARIABLE id.len   0 id.len !
: +id.len   id.len  @  1+   id.len  !  ;    \ increment counter
: >1?    id.len @  7 <  DUP  1 AND  SWAP  NOT  2 AND +  ;
( -- 1 if id.len < 7, 2 otherwise)

3 WIDE FSM: (id)
\ input:     |  other  |  letter     |    digit    |
\ state      ---------------------------------------
   ( 0 )       NOOP >2   +id.len >1    NOOP >2
   ( 1 )       NOOP >2   +id.len >1?   +id.len >1?  ;

: <id>   ( $end $beg -- f)                  \ f = -1 for id, 0 else
         0 id.len !   0 state< (id) !       \ initialize
         BEGIN   DUP C@  [id]  (id)         \ run fsm
                 DDUP  >                    \ $end > $beg ?
                 state< (id) @  2 <         \ not terminated ?
                 AND                        \ combine flags
         WHILE   1+  REPEAT   DDROP         \ $beg = $beg+1
         state< (id) @   1 =  ;             \ leave flag
```

The resulting FSM is nondeterministic because the word `>1?` induces a transition either to state 1 or to (the terminal) state 2. It has fewer states and consumes less memory despite the extra definitions. Because we have stuck to subroutines for mediating state transitions, going from a definite transition (via a `CONSTANT` such as `>0` or `>1`) to an indefinite one requires no redefinitions.

The following FSM detects properly formed floating point numbers:

```
128 TAB: [fp#]                                    \ decoder for fp#
1 ' [fp#]  ASCII E  ASCII D  install
1 ' [fp#]  ASCII e  ASCII d  install
2 ' [fp#]  ASCII 9  ASCII 0  install
3 ' [fp#]  ASCII +  +  C!
3 ' [fp#]  ASCII -  +  C!
4 ' [fp#]  ASCII .  +  C!

: #err   CRT       \ restore normal output
       ." Not a correctly formed fp#" ABORT   ;  \ fp# error handler

5 WIDE FSM: (fp#)
\ input:     |  other  |   dDeE    |   digit  | + or -  |    dp    |
\ state:     ------------------------------------------------------
   ( 0 )       NOOP >6    NOOP >6     1+  >0    NOOP >6   1+     >1
   ( 1 )       NOOP >6    1+   >2     1+  >1    #err >6   #err   >6
   ( 2 )       NOOP >6    #err >6    NOOP >4    1+   >3   #err   >6
   ( 3 )       NOOP >6    #err >6     1+  >4    #err >6   #err   >6
   ( 4 )       NOOP >6    #err >6     1+  >5    #err >6   #err   >6
   ( 5 )       NOOP >6    #err >6    #err >6    #err >6   #err   >6 ;

: skip-  DUP C@  ASCII - =  -  ;                  \ skip a leading -
\ Environmental dependency: assumes "true" is -1

: <fp#>   ( $end $beg -- f)
        0 state< (fp#) !                          \ initialize state
        skip-                                     \ ignore leading - sign
     1-  BEGIN   1+  DUP C@  [fp#]  (fp#)         \ run fsm
                 DDUP  <                          \ $end < $beg ?
                 state< (fp#) @   6 =  OR         \ terminated by error ?
         UNTIL   DDROP                            \ clean up
         state< (fp#) @   6 <   ;                 \ leave flag
```

The FSM `(fp#)` does not count digits in the mantissa, but limits those in the exponent to two or fewer. A nondeterministic version of `(fp#)` reduces the number of states, while counting digits in both the mantissa and exponent of the number:

```
5 WIDE FSM: (fp#)
\ input:     |  other  |   dDeE    |   digit  | + or -  |    dp    |
\ state:     ------------------------------------------------------
   ( 0 )       NOOP >4    NOOP >4   +mant >0?   NOOP >4   1+     >1
   ( 1 )       NOOP >4    ?1+  >2   +mant >1?   #err >4   #err   >4
   ( 2 )       NOOP >4    #err >4   +exp  >3    1+   >3   #err   >4
   ( 3 )       NOOP >4    #err >4   +exp  >3?   #err >4   #err   >4  ;
```

The task of fleshing out the details—specifically, the words `+mant`, `+exp`, `?+1`, `>0?`, `>1?` and `>3?`—is left as an exercise for the reader.

Acknowledgments

I am grateful to Rick Van Norman and Lloyd Prentice for positive feedback about applications of the FSM compiler in areas as diverse as gas pipeline control and educational computer games.

Appendix

Here is a high-level definition of the HS/FORTH `CASE: ... code;CASE` (albeit it imposes more overhead than HS/FORTH's version) that works in indirect-threaded systems:

```
: CASE:   CREATE  ]   DOES>  ( n -- )  OVER  + +  @  EXECUTE ;
: ;CASE   [COMPILE] ;   ;  IMMEDIATE   ( or just use ; )
```

However, it will not work with a direct-threaded FORTH like F-PC. It fails because what is normally compiled into the body of the definition (by using `]` to turn on the compiler) in a direct-threaded system is not a list of execution tokens. The simplest alternative (it also works with indirect-threaded systems) factors the compilation function out of `CASE:`

```
: CASE:      CREATE   ;
: |     ’  ,  ;                                  \ F83 and ANS version
: ;CASE      DOES>  OVER + +  PERFORM  ;         \ no error checking
```

Here is a usage example:

```
CASE: TEST | * | / | + | - ;CASE
3 4 0 TEST . 12  ok
12 4 1 TEST .  3  ok
5 7 2 TEST . 12  ok
5 7 3 TEST . -2  ok
```

Although the `CASE` statement can be made to extend over several lines if one likes, readability and good factoring suggest such definitions be kept short.

The same technique can be used to provide a version of `FSM:` that works with F-PC and other F83-based or ANS-compliant systems, for those who want to experiment with `FSM:` but lack HS/FORTH to try the version of *The Best FSM So Far* with. The definitions are:

```
: |     ’  ,  ;              \ F83 and ANS version
: WIDE   0 ;

: FSM:   ( width 0 -- )      CREATE   ,  ,   ;
: ;FSM   DOES>                 ( col# adr -- )

          DUP >R  2@  *  +     ( -- col#+width*state )

          2*  2+  CELLS        ( -- offset-to-action)

          DUP >R               ( -- offset-to-action)
          PERFORM              ( ? )
          R> CELL+             ( -- ? offset-to-update)
          PERFORM              ( -- ? state’)

          R> !   ;  ( ? )       \ update state
```

The FSM of *An Elegant FSM* now takes the form:

```
4 WIDE FSM: <Fixed.Pt#>
\ input:  |    other?  |   num?     |    minus?   |    dp?     |
\ state:  ------------------------------------------------------
   ( 0 )  | DROP | >0  | EMIT | >1  | EMIT  | >1  |  EMIT  | >2
   ( 1 )  | DROP | >1  | EMIT | >1  | DROP  | >1  |  EMIT  | >2
   ( 2 )  | DROP | >2  | EMIT | >2  | DROP  | >2  |  DROP  | >2  ;FSM
```

```
: state<    ' >BODY   LITERAL  ; IMMEDIATE
: Getafix   0   state< <Fixed.Pt#>  !
            BEGIN   KEY   DUP   13 <>      WHILE
            DUP   cat->col#  <Fixed.pt#>   REPEAT ;
```

References

[1] Scientific Forth: a modern language for scientific computing. *J. V. Noble.* Mechum Banks Publishing, Ivy, VA, 1992. See esp. Ch. 11 and included program disk.

[2] A. V. Aho R. Sethi J.D. Ullman. *Compilers: Principles, Tools and Techniques.* Addison Wesley Publishing Company, Reading, MA, 1986.

[3] J[ames] Basile. A forth finite state machine. *Journal of Forth Application and Research*, 1(2):76–78, 1982.

[4] D. W. Berrian. Forth based control of an Ion implanter. In *Proc. 1989 Rochester Forth Conf.*, pages 1–5. Inst. for Applied Forth Res., Inc., Rochester, NY, 1989.

[5] A. K. Dewdney. *The Turing Omnibus: 61 Excursions in Computer Science.* Computer Science Press, Rockville, MD, 1989.

[6] C. E. Eaker. The CASE statement. *Forth Dimensions*, 2(3):37–40, 1980.

[7] Harvard Softworks, P. O. Box 69, Springboro, OH 45066. *HS/FORTH.*

[8] J. V. Noble. Avoid decisions. *Computers in Physics*, 5(4):386, 1991.

[9] E. Rawson. State sequence handlers. *Journal of Forth Application and Research*, 3(4):75–64, 1986.

[10] R. Sedgewick. *Algorithms.* Addison Wesley Publishing Company, Reading, MA, 1983.

[11] J. Woehr. *Forth: The New Model.* M&T Books, San Mateo, CA, 1992.

[12] ANSI X3.215-1994. *American National Standard for Infomation Systems – Programming Languages – Forth.* American National Standards Institute, New York, NY, 1994.