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

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.

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

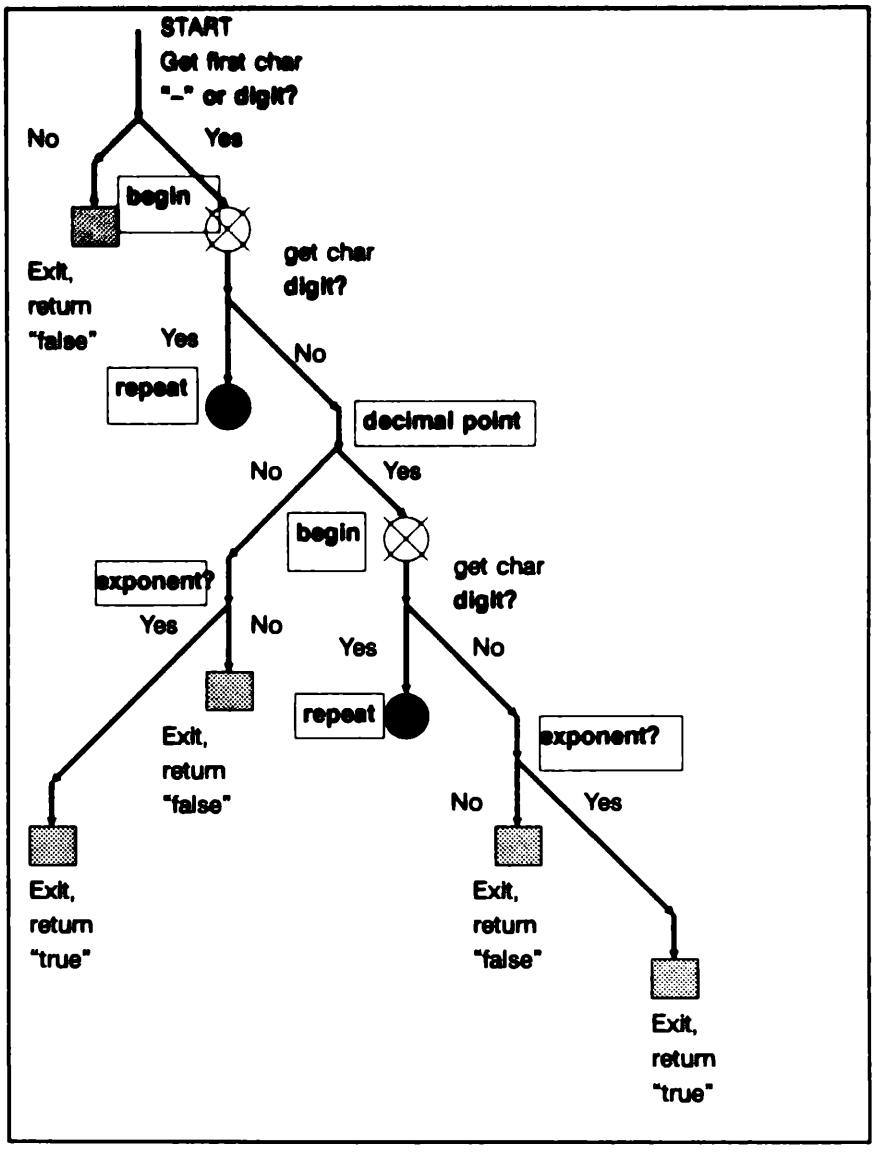

Fig. 11-1 *Logic tree for <floating point #>*

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).

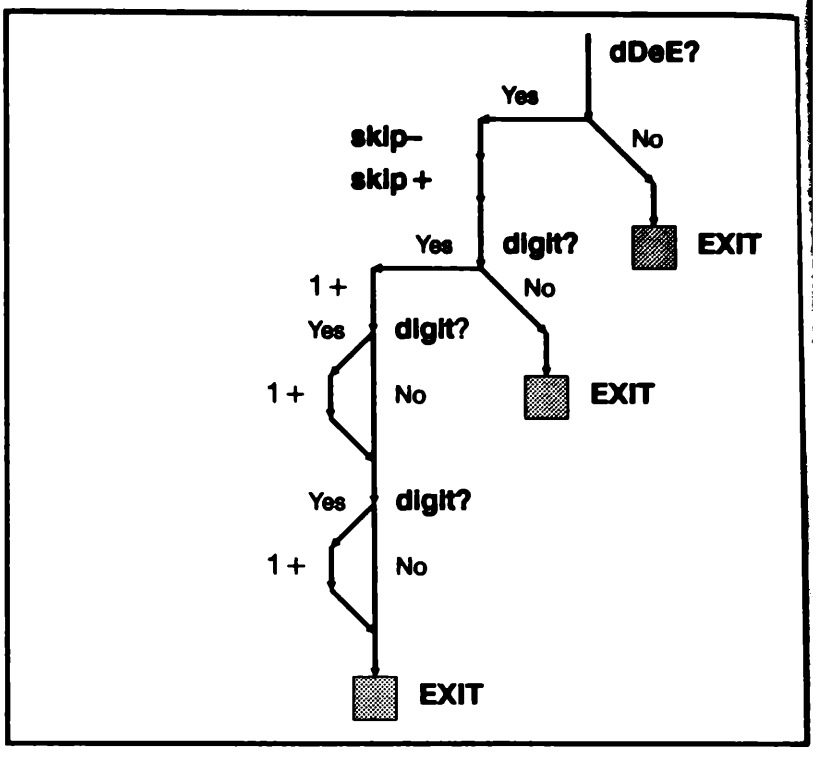

Fig. 11-2 *Logic tree for <exponent>*

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.

```
                ' 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.

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.

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#)
```

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