Scientific Computation in FORTH

Julian Noble

Can FORTH replace Fortran as the dominant language of scientific computing?

FORTH has been called "...one of the best kept secrets in the computing world."[1] Originally invented to acquire data from and control radio telescopes, FORTH spread rapidly through the astronomical community. But the only headway it has made among physicists has been for interfacing computers with laboratory instruments[2].(Both directly and via commercial instrumentation systems that are FORTH dialects; e.g., the ASYST package from ASYST Software Technologies, Inc.) FORTH is so simple it is often the first language implemented on a new computer; moreover, several processors use it for native machine code. This simplicity, combined with fast execution and economical memory use has led to FORTH's widespread application in instrumentation, robotics, automation, process control, computer games, etc.

Oddly, most devotees of FORTH think it unsuited to scientific computation. This prejudice arose in part from the deliberate omission of floating point arithmetic from FORTH's native repertoire. It is also true that poorly written FORTH can be even more heiroglyphic than poorly written programs in other languages, gaining FORTH the undeserved reputation of being a "write-only" language. The same can be said for poorly written code in other languages such as C; however well-written FORTH is delightfully clear.

Shortly after acquiring a PC I became mildly interested in FORTH and acquired a system to play with. This turned out to be a real stroke of luck when I found myself at CERN with a (trans)portable PC and only an infernally slow Fortran compiler for all my numerical work. At some point—probably while enduring the compile-and-link of a short Fortran program—it struck me that the math chip on a PC is a machine, and that FORTH could direct it through a calculation. The chief virtue of number crunching in FORTH was the speed of writing and debugging programs, once I got used to its unusual programming style. Isolated from the community of FORTH enthusiasts I developed my own extensions to FORTH—specialized techniques that fostered clear and simple scientific programming. More experienced FORTH programmers have found these extensions interesting and encouraged me to share them with the physics community. Hence this article.

Scientific/Technical Computing

Languages intended for scientific applications must provide a certain minimum functionality. Fortran provides the essentials through predefined data structures and predefined operations which account for 80% or more of routine tasks. But unorthodox requirements lead to a

Julian Noble is Professor of Physics at the Institute for Nuclear and Particle Physics, University of Virginia.

quagmire of assembly-language programming with strict interfacing conventions, and with the need to assemble and re-link to test each new routine. It can be done, but the system groans and thrashes like a dinosaur in a tar pit.

Scientific computing demands

- floating point arithmetic
- double precision (floating point) arithmetic
- complex arithmetic
- simple array notation
- subroutines
- functions
- a library of built-in functions
- useful control structures
- simple I/O
- formula translation

Fortran provides them, and its popularity endures. But a scientific programming language requires more than minimum functionality. It must be as machine-independent (that is, *portable*) as possible, in order to support the development of a standard library of useful subroutines; as well as to accomodate hardware upgrades and changes. Fortran has become sufficiently standardized that it meets this criterion.

A good language should support various programming styles, such as procedural, object-oriented, functional, modular, multi-tasking, parallel, recursive, data-stream and so forth. Here Fortran—a purely procedural language—falls down compared with more modern languages such as C, Modula-2 and, yes, FORTH.

Good languages encourage good programming practices. Six month old programs have much larger entropy than freshlywritten ones. This is especially true of programs written by students and assistants. Fortan permits—and encourages in the name of speed—such heinous practices as "spaghetti code", filled with **GOTO**'s, assigned **GOTO**'s and computed **GOTO**'s that make it nearly impossible to follow execution flow. Fortran's 7-letter naming convention fills the code with cryptic names that require discursive comments and a glossary to decipher. Both are generally omitted because they are tedious to write.

In Fig. 1 below, I exhibit a typical Fortran subroutine[3] that obeys the best precepts of "structured" coding, yet lacking comments and accompanying documentation the program is unclear. Its algorithm is not easily deduced from a superficial reading. Contrast this with the corresponding FORTH translation, whose flow diagram is shown in Fig. 2 and the main subroutine in Fig. 3. The FORTH version is almost self documenting because the words have descriptive and telegraphic names. (Only the main subroutine (word) and one of the subsidiary words are shown.) The low overhead of subroutine calls in FORTH encourages decomposing algorithms into small subroutines for ease of development and debugging. The FORTH translation of this Fortran program guarantees the resulting code will be structured. FORTH permits neither statement labels nor **GOTO**'s. FORTH words have only one entry and one exit. This discipline also pays large dividends in debugging.

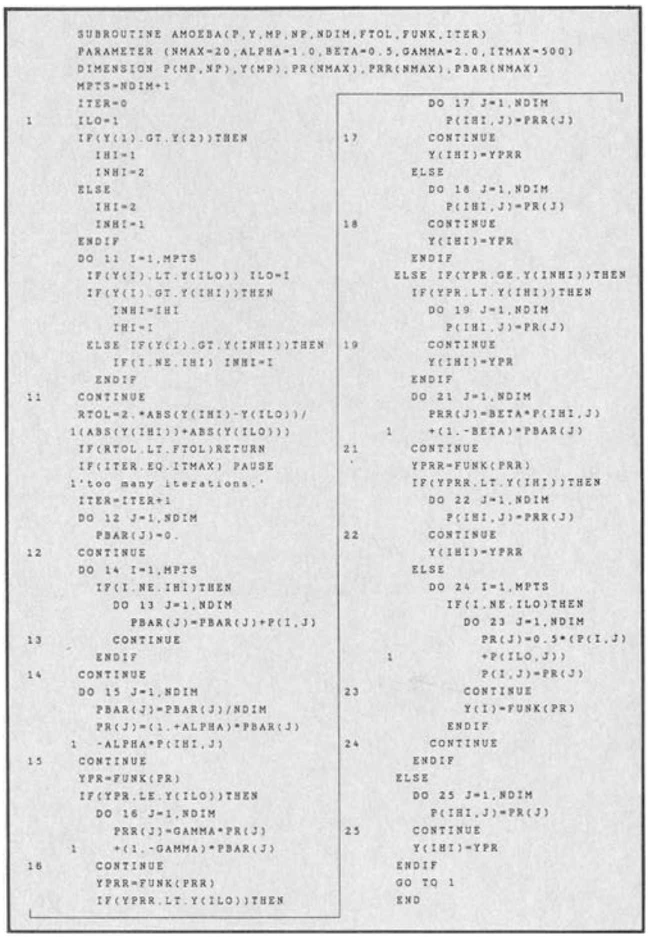
What is FORTH?

The core of Fortran is the FORmula TRANslator that gave it its name. A Fortran program is a sequence of algebraic expressions to be evaluated, with the order determined by control statements such as **DO, IF...THEN, GOTO,** etc. Since many calculations can be organized as sequential evaluation of expressions, a Fortran program is often the most direct route to the goal.

A FORTH program differs radically from its Fortran analog, both in structure and in concept. A typical program is a list of names of simple subroutines called words. A word is executed simply by naming it, so a string of words performs a sequence of subroutines. Programmers try to choose telegraphic and expressive names so the resulting FORTH program reads like instructions to a thick-witted assistant. The algorithm is usually obvious and self-documenting in programs written like this. Consider, e.g., the definition of a word to solve linear equations by Gaussian elimination:

: }}SOLVE TRIANGULARIZE BACK.SOLVE ;

The definition of **}}SOLVE** begins with a colon **:** and ends with a semicolon **;**, FORTH words whose functions will be explained below. The definition expresses clearly the two stages of Gaussian elimination without need for additional comments. With **A{{** and **B{** , respectively, the names of



```
      SUBROUTINE AMOEBA(P,Y,MP,NP,NDIM,FTOL,FUNK,ITER)
      PARAMETER (NMAX=20,ALPHA=1.0,BETA=0.5,GAMMA=2.0,ITMAX=500)
      DIMENSION P(MP,NP),Y(MP),PR(NMAX),PRR(NMAX),PBAR(NMAX)
      MPTS=NDIM+1
      ITER=0
1     ILO=1
      IF(Y(1).GT.Y(2))THEN
        IHI=1
        INHI=2
      ELSE
        IHI=2
        INHI=1
      ENDIF
      DO 11 I=1,MPTS
       IF(Y(I).LT.Y(ILO)) ILO=I
       IF(Y(I).GT.Y(IHI))THEN
          INHI=IHI
          IHI=I
       ELSE IF(Y(I).GT.Y(INHI))THEN
          IF(I.NE.IHI) INHI=I
        ENDIF
11    CONTINUE
      RTOL=2.*ABS(Y(IHI)-Y(ILO))/
     1(ABS(Y(IHI))+ABS(Y(ILO)))
      IF(RTOL.LT.FTOL)RETURN
      IF(ITER.EQ.ITMAX) PAUSE
     1'too many iterations.'
      ITER=ITER+1
      DO 12 J=1,NDIM
        PBAR(J)=0.
12    CONTINUE
      DO 14 I=1,MPTS
        IF(I.NE.IHI)THEN
          DO 13 J=1,NDIM
            PBAR(J)=PBAR(J)+P(I,J)
13        CONTINUE
        ENDIF
14    CONTINUE
      DO 15 J=1,NDIM
        PBAR(J)=PBAR(J)/NDIM
        PR(J)=(1.+ALPHA)*PBAR(J)
     1  -ALPHA*P(IHI,J)
15    CONTINUE
      YPR=FUNK(PR)
      IF(YPR.LE.Y(ILO))THEN
        DO 16 J=1,NDIM
          PRR(J)=GAMMA*PR(J)
     1    +(1.-GAMMA)*PBAR(J)
16      CONTINUE
        YPRR=FUNK(PRR)
        IF(YPRR.LT.Y(ILO))THEN
          DO 17 J=1,NDIM
            P(IHI,J)=PRR(J)
17        CONTINUE
          Y(IHI)=YPRR
        ELSE
          DO 18 J=1,NDIM
            P(IHI,J)=PR(J)
18        CONTINUE
          Y(IHI)=YPR
        ENDIF
      ELSE IF(YPR.GE.Y(INHI))THEN
        IF(YPR.LT.Y(IHI))THEN
          DO 19 J=1,NDIM
            P(IHI,J)=PR(J)
19        CONTINUE
          Y(IHI)=YPR
        ENDIF
        DO 21 J=1,NDIM
          PRR(J)=BETA*P(IHI,J)
     1    +(1.-BETA)*PBAR(J)
21      CONTINUE
        YPRR=FUNK(PRR)
        IF(YPRR.LT.Y(IHI))THEN
          DO 22 J=1,NDIM
            P(IHI,J)=PRR(J)
22        CONTINUE
          Y(IHI)=YPRR
        ELSE
          DO 24 I=1,MPTS
            IF(I.NE.ILO)THEN
              DO 23 J=1,NDIM
                PR(J)=0.5*(P(I,J)
     1          +P(ILO,J))
                P(I,J)=PR(J)
23            CONTINUE
              Y(I)=FUNK(PR)
            ENDIF
24        CONTINUE
        ENDIF
      ELSE
        DO 25 J=1,NDIM
          P(IHI,J)=PR(J)
25      CONTINUE
        Y(IHI)=YPR
      ENDIF
      GO TO 1
      END
```

Fig.1: A Fortran subroutine for minimizing by the simplex algorithm.

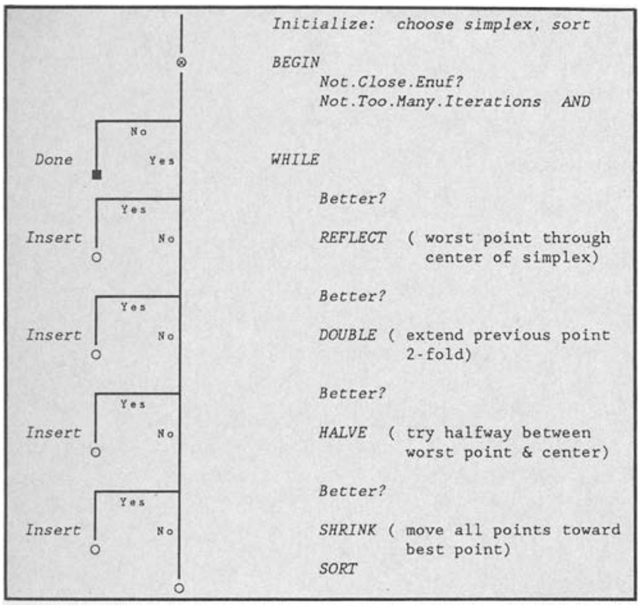

Initialize: choose simplex, sort
BEGIN
Not.Close.Enuf?
Not.Too.Many.Iterations AND
No
Done
Yes
WHILE
Yes
Better?
Insert
No
REFLECT (worst point through center of simplex)
Yes
Better?
Insert
No
DOUBLE (extend previous point 2-fold)
Yes
Better?
Insert
No
HALVE (try halfway between worst point & center)
Yes
Better?
Insert
No
SHRINK (move all points toward best point)
SORT

Fig.2: Flow diagram of simplex algorithm in FORTH.

the matrix and the inhomogeneous term, the input line **A{{ B{ }}SOLVE** instructs the computer to solve the linear equation(s)

$$Ax = B$$

and to overwrite **B{** with the solution. Two noteworthy points emerge from this example: First, the natural order of precedence in FORTH is reverse-Polish, or *postfix* as with Hewlett-Packard calculators—arguments *precede* operators. Second, the period **.** in **BACK.SOLVE** and the curly braces { and } in **A{{** , **B{** and **}}SOLVE** have no special significance—unlike C code, where such reserved characters benefit the compiler-writer more than the C user. They reflect FORTH's ability to use *any* ASCII characters in its names (except line-feeds, bells, et al.). My array notation uses braces as mnemonic devices in names: the most natural way to translate the FORTRAN array element A(I,J) to FORTH would be **I J A**. But then it would be unclear whether **I J A** represented an array address or something else. So I invented the notation **A{{ I J }}**, (and the analagous one **V{ I }** for vectors), where **}}** is an operator that computes the address using information previously placed on the stack by **A{{** , **I** and **J**.

High level languages must be translated into machine code. In the early days of programming there seemed to be but two ways to accomplish this: either to translate (and store!) the entire program, once and for all—*compiling* the

```
\ SIMPLEX MINIMIZATION ALGORITHM
\ Usage: USE( Fn.Name ERROR % E )MINIMIZE

   : )MINIMIZE    INITIALIZE
       BEGIN      Close.Enuf?  Too.Many.Iterations  OR   NOT
       WHILE          REFLECT  Better?  NOT
                  IF  DOUBLE   Better?  NOT
                  IF  HALVE    Better?  NOT
                  IF  SHRINK   SORT  THEN  THEN  THEN
       REPEAT  ;

   : Better?   New.Point   Worst.Point   Lower?
       DUP   IF   New.Point   Worst.Point   INSERT  SORT  THEN  ;
```

Fig.3: A FORTH variant of the simplex minimization algorithm.

program; or to translate one line at a time "on the fly", each time the program is run—*interpreting* the program. Interpretation is immediate. The interpreter can even execute interactively, as in the familiar BASICA sequence

```
PRINT SQR(36.) <cr>
5.999999
OK
```

Interpretation is useful in testing program fragments and leads to rapid program development. A change in the program takes no extra time, since it will be translated to machine code only when the program execution encounters the change.

Interpreted languages use memory economically: source code is usually much more compact than its machine code image. An interpreter translates a small portion of source code at a time, executes it and then forgets the translation, so the entire machine code image never resides in memory. But the immediacy, economy and flexibility of interpretation cost: interpreted execution is far slower than compiled execution of equivalent programs because of the translation overhead each time a given statement—inside a loop, say—is executed. So why not compile all the time, if compiled programs execute so much faster? nfortunately, compiling to efficient code can be painfully slow, and most compilers also require a slow linking step. Developing a complex compiled program on a microcomputer can be endlessly frustrating. For this reason, modern development systems try to achieve the best of both worlds by providing interpreted and compiled versions of the same language.

FORTH is a *threaded interpretive language*, inhabiting a niche between interpreted languages (BASICA, APL, LISP) and compiled languages (Fortran, C, PASCAL, MODULA-2). It combines the virtues of interpretation—interactivity, memory economy, flexibility—with a form of compilation. The program translates to machine code as it is entered—*incremental compilation*—and not as it is executed. The resulting object code usually executes slower than that of an optimizing front end compiler (although much faster than a more standard interpreter) but needs as little memory as interpreted code. The speed disadvantage relative to traditional compiled code is not serious as FORTH can easily be optimized to run as fast as the best optimizing compilers.(On some chips like the Novix NC4016 and Harris FORCE chip set, whose native code is optimized FORTH, execution speed exceeds that of optimized C. On a PC, FORTH's speed can easily approach that of assembly language. For example, my linear equations program solves a dense system of 350 equations in 16 minutes on a 10 MHz PC. On a Definicon DSI-785 it will take about 3 minutes. Such efficiencies give workstations quicker turnaround than most mainframes and minis.) The threading technique at the heart of FORTH has been applied to some of the incrementally compiled commmercial development systems alluded to above, such as Microsoft QuickBASIC.

The best way to convey the flavor of FORTH is to pretend we are sitting at a FORTH workstation. Type in

DECIMAL 2 3 + . <cr> 5 ok

(<cr> means "carriage return". What follows <cr> is the systems's response to our input.)

What happened? FORTH consists entirely of subroutines called words that are invoked when they are named. The names of words are stored in the dictionary. The interpreter is an endless loop that waits for input (from the keyboard, a disk drive or some other device), and when it gets some, parses it according to the rules:

1. Parse the input into distinct words (separated by spaces).
2. Look up each word (in turn) in the dictionary. If it is found, **EXECUTE** it.
3. If it is not found, try to interpret the text as a number. If it is a number, push it onto the data stack. (A stackis a specialized data structure used heavily in computer architecture. It is like a stack of cards, each bearing a number, and new cards can be added or removed only at the top.)
4. If it is neither a word nor a number, issue an error message.

These rules generate the following sequence of actions:

DECIMAL is looked up, found and executed. (It makes the base of the number system 10 (ten). **HEX** would correspondingly convert arithmetic to base sixteen.)

Then **2** and **3** are interpreted as numbers and pushed on the stack.

The next item, **+**, is found and executed. Its code consumes the top two numbers on the stack, leaving their sum on top of the stack.

The word **.** (called "emit") consumes the top number on the stack and displays it on the console.

Finally, "ok" is FORTH's way of saying "Everything went smoothly and you are back in the interpreter loop again, waiting for input."

But if FORTH is a compiled language, where is the compiler? FORTH uses the simplest compiling technique of any language I know. We invoke it by defining new FORTH words in terms of previously-defined ones:

```
: 2+/ ( a b -- a/[b+2]) 2 + / ; ok
```

(From now on, the carriage return, <cr>, at the end of a line will be implicit.)

The new definition includes a stack comment (a b -- a/[b + 2]) set off by parentheses, that shows what happens to the top two numbers on the stack when the new word executes. (FORTH ignores anything from the word (to the next) in the input stream, inclusive. This is why we use brackets [,] to enclose b + 2. Since (is a word, the space following it is is significant.) Now we test the new word:

```
12 4 2+/ . 2 ok
```

The new word **2+/** is now part of the dictionary. This is the sense in which FORTH is extensible—there is no difference in status between system-supplied words and user-defined words: they are all looked up and executed the same way.

A major virtue of FORTH is *uniformity*. There is no distinction between a main program, a function, a subroutine, a data structure or an operation. They are all words. Typically, a FORTH **VARIABLE** places its memory address on the stack. A FORTH **CONSTANT** places its *value* on the stack. A FORTH operator such as * (integer multiply) expects its operands on the stack and consumes them, leaving the product. Thus, to divide a **VARIABLE X** by a **CONSTANT P1**, leaving the remainder (an operation used in a "hashed" table lookup scheme) we would say (here @ — "fetch" — retrieves the contents of the memory cell whose address is on the stack)

```
X @ P1 MOD
```

We have already given an example of defining a new arithmetic operator, **2+/** . The same principle lets us extend FORTH's arithmetic to, e.g., complex arithmetic operators such as **X*** or **X*F** (multiply two complex numbers, and multiply a complex number by a real). Thus, e.g., complex arithmetic is as painless in FORTH as in Fortran. The same cannot be said for languages like PASCAL or C, where if it ain't built in, it cain't be had. Thus, e.g., we define **X*F** as

```
: X*F (:: y x a -- y*a x*a )
  FUNDER (:: y x a -- y a x a )
  F*  F-ROT  F*  FSWAP  ;
```

and **X*** as

```
X* (:: y x a b -- b*x+y*a a*x-b*y)
FSWAP XDUP 5 FROLL X*F
XSWAP 5 FROLL X*F F-ROT F+ F-ROT F+ ;
```

How does compilation actually work? The endless loop mechanism (the *outer interpreter*) has two states, *interpret* and *compile*. The word **:** performs two main tasks. First it makes a new dictionary entry whose name is the next piece of text in the input stream. In the above example, the name was **2+/** . Second, **:** switches the system from *interpret* to *compile*. (In fact, **:** is itself defined in terms of the words, **CREATE** and **]**. **CREATE** makes the new dictionary heading, while **]** turns on the compiler. Correspondingly, **[** switches back from compile to interpret and is part of ;.)

In compile mode, the parser looks up each piece of incoming text and if it has been previously defined, the parser inserts the address of the found word's "parameter field" into the parameter field of the new word. (This happened to **+** and **/** above). When the new word is executed, these addresses establish the sequence of subroutines that do its work. *Threading* is an appropriate term for this technique.

If the text is not found, control reverts to the word **NUMBER** that determines if it is a number. A number is stored in the parameter field, together with some special code that retrieves it and pushes it on the stack when the word containing the number is run.

Finally, the terminating semicolon **;** is a special kind of word. It is *immediate* so **;** is **EXECUTE**d even when the system is in the *compile* state. Execution of **;** installs the terminating code (called **NEXT**) into the new definition, and then switches the system back from *compile* to *interpret* mode. Simple and incredibly elegant!

Unique to FORTH is the availability to the programmer of all the components of the compiler. For example, the components **CREATE** and **]** play key roles in defining generic operators (discussed below) that permit one version of a program to work with single- and double-precision real and complex numbers. Ordinary compiled languages cannot duplicate such feats.

A Scientific FORTH dialect

Although floating point arithmetic can be implemented in software, the ubiquity of inexpensive arithmetic coprocessors (FPU's) makes this pointless. The Intel 80×87 and Motorola 6888× chips add fast arithmetic to their respective 80×86 and 680×0 CPUs. The InMos Transputer T800 includes its floating point unit (FPU) on the same chip. FORTH's built-in assembly language vocabulary permits direct control of these FPUs and hence simple extension to floating point arithmetic.

FORTH is not presently as standardized as Fortran (and no official standards at all exist for floating point or complex arithmetic). Nevertheless some *de facto* standards have developed over the past few years[4]. Most FORTH vendors have realized the necessity for a separate floating point stack (fstack) and a separate set of words to manipulate it. This follows from the fact that the standard co-processors are stack-based. The 8087/80287/80387 chips, e.g., have a built-in stack of limited depth (in fact 8 deep). It is far faster to get a number from this intrinsic fstack than it would be to get a number from memory. Optimizing for speed involves maximum use of the intrinsic fstack to store intermediate results, frequently used constants, etc.

The standard scientific extensions to FORTH consist of floating point and complex arithmetic and a function library for each. My own innovations include a method for implementing typed data (REAL, COMPLEX, DOUBLE PRECISION, and DOUBLE COMPLEX) at run-time, described below; a Fortran-like array notation; and a formula translator that converts a line of Fortran into (generally more than one line of) FORTH. Although the details of these extensions have been published elsewhere[5], the typed data technique so well illustrates the power and abstracting ability of FORTH that it is worth highlighting here.

The magic is accomplished with the most powerful part of FORTH, the **CREATE...DOES>** construct, justly called "...the pearl of FORTH[6]". These words let us define *defining* words, whose job is to make new dictionary entries with specific run-time behavior. Defining words are exemplified by the (high-level) definition of **CONSTANT**

```
: CONSTANT CREATE , DOES> @ ;
```

CONSTANT lets us define a new constant with the phrase

```
6659 CONSTANT PRIME1
```

Now test the new constant:

```
PRIME1 . 6659 ok
```

The defining word **CONSTANT** used **CREATE** to make a new dictionary header with the name **PRIME1**. Then the comma **,** took the number 6659 off the stack and stored it in the first cell of **PRIME1**'s parameter field. The word **DOES>** specified the run-time action common to all **CONSTANT**'s daughter words. Any word defined with **CREATE** places its parameter-field address on the stack at run-time. Hence, specifying the further run-time action of **@** (fetch) places the contents of **PRIME1** on the stack whenever we say **PRIME1**.

Now, to define generic operators, we must first define generic data structures that—in addition to their numerical contents—also contain a label specifying their type. When such variables are fetched to the fstack, their type-label must be fetched to a tstack (type-stack) that parallels the fstack. We imagine this has been worked out, and that we now want to define generic arithmetic.

Efficient generic operators borrow a technique—the jump table—from assembly language programming. By giving access to its compiler, FORTH simplifies the construction of jump tables, compared to other languages. We can illustrate this interactively with a terminal session (explanations have been added following the "ignore rest of line" symbol,\ ):

```
CREATE JUMP.TAB          \make a header for JUMP.TAB

          ] * +          \start compiler and compile
                         \addresses of * and +

          [              \turn off compiler

                         \now test:

2 4                      \place 2 and 4 on stack

JUMP.TAB @ EXECUTE       \get address of * and execute
.8 ok                    \emit result: 8 = 2*4

2 4 JUMP.TAB 2+ @ EXECUTE\repeat, except get address of +
.6 ok                    \emit result: 6 = 2 + 4
```

Now that we know how to make a jump table (called *vectoring*, in FORTH jargon), how do we use it? We shall have to specify a run-time action that selects the desired operation using information contained in the tstack. The same technique applies to defining words for generic unary and binary operators. A unary operator such as **FNEGATE** or **FEXP** expects one argument and leaves one result. With an FPU the only distinction is between real or complex. This distinction is contained in the second bit of the type label, and can be extracted *via* the code fragment

```
(type -- 0 = real| 2 = complex) 2 AND
```

Most unary operators produce results of the same type as their argument. Thus, to save time we can copy the tstack rather than popping it, with the result

```
:GU: CREATE ] DOES> (--pfa)
    T@ 2 AND + @ EXECUTE ;
```

When we use **GU:** in the form

```
GU: GNEGATE  FNEGATE XNEGATE ;
```

CREATE produces a dictionary entry for **GNEGATE**, **]** turns on the compiler so the previously defined <u>words</u> **FNEGATE** and **XNEGATE** have their addresses compiled into **GNEGATE**'s parameter field, and **DOES>** attaches the run-time code. The run-time code converts the real/complex bit into an offset, 0 or 2 which is added to the address of the daughter <u>word</u> to get the address where the pointer to the actual code is stored. This pointer is fetched and **EXECUTEd**.

A few unary operators like **XABS** (complex absolute value) return real values from complex arguments. If we want to use **GU:** to define, say, **GABS**, we must remember to define **XABS** so it zeros the second bit of the type descriptor on the <u>tstack</u>. This is just a logical **AND** so it is very fast.

A binary operator (one that takes two arguments) expects its arguments on the <u>fstack</u> and their types on the <u>tstack</u>. There is no distinction between single- and double-precision arithmetic on most numeric coprocessors. However, the result must leave the proper type label on the <u>tstack</u>. Here is what we want to happen, illustrated as a matrix of <u>tstack</u> combinations:

	R	D	X	DX
R	R	R	X	X
D	R	D	X	DX
X	X	X	X	X
DX	X	DX	X	DX

If we think of the indices and entries in this matrix as numbers 0, 1, 2, 3 (so we can use them as indices into a table) rather than as letters, an immediate algorithm emerges: the first and second bits of the result are, respectively, the logical-AND of the first bits, and the logical-OR of the second bits of the operands. Although we would program this in assembler for speed, the high-level definition is

```
: NEW.TYPE                    ( a b--a2 + b2 + a1b1)
     DDUP                                 (--a b a b)
     AND                                   (--a b ab)
     1 AND                              (--a b [ab]1)
     -ROT OR                           (--[ab]1 a+b)
     2 AND                          (--[ab]1 [a+b]2)
     + ;                           (--a2 + b2 + a1b1)
```

Since only logical operations are used, **NEW.TYPE** is much faster than table lookup or branching. Note that in programming this key <u>word</u> we have obeyed the central FORTH precept: "Keep it simple!" by choosing a data structure (the numeric type tokens 0-3) that is easily manipulated.

We will also need a way to select the appropriate operator from a jump table of addresses. Given that the precision (internal) is irrelevant, again all that matters is whether the number is real or complex, i.e. the second bits of the numbers. The first operation must then be to divide by 2 (right-shift by one bit). We then have the matrix

$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & RR & RX \\ 1 & XR & XX \end{array} \rightarrow \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 2 & 3 \end{array}$$

where RR stands for real-real, etc. The numerical elements are generated as 2*J + I. This leads to the <u>word</u>

```
: WHICH.OP (a b--c) 2/ SWAP 2 AND + ;
```

Thus,

```
:GB:CREATE ] DOES> (--pfa)
T> T> DDUP                 (-- t1 t2 t1 t2)
NEW.TYPE >T                \make result-type
WHICH.OP 2* + @ EXECUTE ;\select binop
```

with the usage

```
GB: G* F* F*X X*F X* ;
```

The generic multiply picks out which of the four routines to use at run-time. By using only logical or shift operations we have made even the high-level definitions fairly quick in comparison with the times of floating point operations. The only instance where one might forego the overhead penalty paid for the convenience of generic coding would be in nested inner loops, such as occur in matrix operations.

The astute reader has doubtless recognized a marked absence of error checking in the above examples. The FORTH coding philosophy holds that since it is easy to certify each <u>word</u> as it is defined, the global error detection schemes that typify other languages are unnecessary in FORTH. For those who prefer suspenders *and* a belt, FORTH offers simple ways to include debugging and error checking code, and equally simple ways to strip it out after the program is bug-free. I rarely bother, and as a consequence crash my system about once in ten tries when I am programming.

FORTH programming example: Adaptive quadrature

This example shows how to translate Fortran pseudocode to FORTH, permitting a program impossible to implement in Fortran. To keep it simple, I eschew the typed data and array techniques alluded to above, but use the Fortran to FORTH FORmula TRANSlator. The same technique has been used to build a generic routine that I frequently use in my numerical work.

The program performs adaptive quadrature based on the trapezoidal rule with Richardson extrapolation. The idea is to divide an interval in two and compare the sum of the integrals on the two pieces with that on the whole interval. If they agree, the integral on that interval is considered converged, and it is stored. Then the program works on any remaining part of the original interval. If the integral has not converged, then the program works on the right half. This will be evident from the output from the example. The simplest way to program the algorithm is recursively. That is, if a Fortran subroutine could call itself, we could write (note all the comment lines!)

```
      FUNCTION TZ(A,B,F1,F2)
C
C TRAPEZOIDAL SUB-INTEGRAL
C
      TZ = (B-A)*(F1 + F2)/2
      RETURN
      END
      FUNCTION ADAPT(A,B,F1,F2,ERROR)
      EXTERNAL F
C
C COMPUTE MIDPOINT OF INTERVAL
C
      C = (A + B)/2
C
C AND NEW VALUE OF FUNCTION
C
      F3 = F(C)
C
C TEST FOR CONVERGENCE
C
      I2 = TZ(A,B,F1,F2)
      T4 = TZ(A,C,F1,F3)  +  TZ(C,B,F3,F2)
      DT = T4-I2
C
C CONVERGED?
C
      IF ABS(DT) .LT. ERROR
C
C IF SO, RETURN RICHARDSON EXTRAPOLATION
C
      THEN ADAPT = T4 + DT/3
C
C OTHERWISE, RECURSE
C
      ELSE ADAPT = ADAPT(A,C,F1,F3,ERROR/2) +
            ADAPT(C,B,F3,F2,ERROR/2)
C
      END IF
      RETURN
      END
```

My Fortran → FORTH FORmula TRANslator converts the arithmetic statements of the above program into the following FORTH code fragments:

```
\ TZ = (B-A)*(F1 + F2)/2
% 2  F2  F1 F +  F\  A FNEGATE B F +  F* IS TZ

\ C = (A + B)/2
% 2 B  A F +  F\  IS C

\ F3 = F(C)
C  FUNC{  F  }TION IS F3

\ I2 = TZ(A,B,F1,F2)
F2  F1  B  A  FUNC{  TZ  }TION IS I2

\ I4 = TZ(A,C,F1,F3) + TZ(C,B,F3,F2)
F2  F3  B  C  FUNC{  TZ  }TION
F3  F1  C  A  FUNC{  TZ  }TION F +
IS I4

\ DT = I4-I2
I2 FNEGATE I4  F +  IS DT

\ ADAPT = I4 + DT/3
% 3  DT  F I4 F +  IS ADAPT

\ ADAPT = ADAPT(A,C,F1,F3,ERROR/2) + ADAPT(C,B,F3,F2,ERROR/2)
% 2  ERROR F\F2  F3  B  C  FUNC{  ADAPT  }TION
% 2  ERROR F\  F3  F1  C  A  FUNC{  ADAPT  }TION F +
IS ADAPT
```

The translator follows certain conventions, namely it assumes all variable names are appropriately typed **VAR**'s, and it identifies user-defined functions such as TZ with the notation FUNC{ TZ }TION. Fortran library functions are translated to their FORTH equivalents. One note about control structures: in this program we use only **IF...ELSE...THEN** and loop *via* recursion. In FORTH **THEN** means **ENDIF**. Thus **IF** expects a true/false flag on the stack and skips to **THEN** or **ELSE** (if present) if the flag is false.

The main virtue of the "translation" is it makes clear how the function arguments must be presented on the fstack, and guides us to write the FORTH code shown below. Rather than give all the details, I have commented the file heavily. Normally, far fewer comments would suffice for documentation.

```
\ REAL-VALUED INTEGRATION BY RECURSION--
\ an illustration of the concept
\ Usage: USE(FN.NAME from A to B error E )INTEGRAL
\ Warning: FN.NAME  must not overflow the 8087 fstack--
\       use FS>  and  >FS to offload temps to the fstack

\ COPYRIGHT 1989  JULIAN V. NOBLE
\ FOR NON-COMMERCIAL USE ONLY

TASK R.INT                    \ analogous to a program name

\A method for passing function names as arguments

VARIABLE  <F>
: USE( [COMPILE]  '           \get address of word following
                              USE(
 CFA  <F> ! ;                 \convert to code-field address
                              and store


: F(X)  <F>  @  EXECUTE ;     \ get function's cfa and execute it
BEHEAD'  <F>                  \ make <F> local to USE( and
                              F(X)


SYNONYM from %                \% puts a fp # on the fstack
SYNONYM to %                  \these are strictly for
SYNONYM error %               \clarity in usage

SYNONYM FVAR R32VAR           \ an R32VAR is a 32-bit VAR
                              \-- see below


FINIT                         \initialize coprocessor

F = 0 FVAR A                  \define FVAR's and zero them
F = 0 FVAR B                  \I use this method mainly
F = 0 FVAR C                  \for clarity in the word
F = 0 FVAR E                  \CONVERGED? below
F = 0 FVAR F1
F = 0 FVAR F2                 \an FVAR is a hybrid between
F = 0 FVAR F3                 \an FVARIABLE and an FCON-
                              STANT
F = 0 FVAR old.I              \A puts A's value on the fstack
                              \IS A puts the TOFS into A

: )tz.integral ( 87:: A B F1 F2--[b-a]*f1 + f2]/2 )
 F +  F2/ F-ROT  FR-  F*  ;

: FSTACK.to.ARGs  (:: A B F1 F2 E--)
FS>  IS E FS>  IS F2 FS>  IS F1  FS>  IS B FS>  IS A ;

: 2.INTERVALS A B F +  F2/  FDUP IS C F(X) IS F3 ;

: CONVERGED? (-- f ::-- I4 I4-I2 )
 A C F1 F3 )tz.integral
 C B F3 F2 )tz.integral
 F +  FDUP ( 87:: -- I4 I4)
 A B F1 F2 )tz.integral    ( 87:: -- I4 I4 I2)
 F- FDUP              ( 87:: -- I4 I4-I2 I4-I2)
 FABS E F <  ;          ( --f 87:: -- I4 I4-I2)

: DOUBLE.FSTACK ( :: -- A C F1 F3 E/2 C B F3 F2 E/2 )
  A >FS  C >FS F1 >FS F3 >FS E F2/  FDUP >FS
  C >FS  B >FS F3 >FS F2 >FS >FS  ;

: ACCUMULATE.I  % 3  F/ F +  old.I  F +  IS old.I ;

: ADAPT ( :: A B F1 F2 E -- )
  FDPTH 1 >                   \ any more sub-intervals?
 IF FSTACK.to.FVARs 2.INTER-
VALS
   CONVERGED?                 \double and integrate
   IF ACCUMULATE.I
   ELSE  FDROP  FDROP         \clear coprocessor registers
     DOUBLE.FSTACK            \arrange arguments for 2 calls
     RECURSE  RECURSE         \call ADAPT twice
   THEN
  THEN ;
```

```
: INITIALIZE ( 87:: A B E -- :: A B
F1 F2 E )
  IS E  IS B  IS A  F = 0  IS old.I
  A F(X) IS F1
  B F(X) IS F2                     \compute function values
  FSINIT                           \clear fstack
 A >FS B >FS  F1 >FS  F2
>FS  E >FS  ;
: )INTEGRAL ( 87:: A B E -- I[A,B] )
   INITIALIZE  ADAPT  old.I  ;


USE(FSQRT from 0 to 2 error 1.E-3)INTEGRAL F. 1.8856177 ok
USE(FSQRT from 0 to 2 error 1.E-4)INTEGRAL F. 1.8856186 ok
```

The exact result is

$$\int_0^2 dx\sqrt{x} = 1.88561808...$$

so the results are pretty fair. The inaccuracy in the 7th place is roundoff error from using only 32-bit precision.

Conclusions

In this article I have tried to render the flavor of FORTH programming as I use it for scientific number crunching. The FORTH system I have been using is HS/FORTH by Harvard Softworks. It is MS-DOS file-oriented. Recently I have acquired a Definicon board and UniFORTH (by Unified Systems) for it. This is a block-oriented FORTH, with a quite different, more traditionally FORTH-like user interface. Once past my unfamiliarity with storing code in blocks rather than files, I experienced no major difficulty in porting my Intel 8086/8087 code to the Motorola 68020/68881 environment.

The FORTH programming innovations described in this article, namely run-time data typing and the FORmula TRANslator, are placed in the public domain and will be made available on the East Coast FORTH Bulletin Board (Tel.703-442-8695).

RAY TRACING

on your PC, XT, AT, PS/2 or compatible!

BEAM TWO

for coaxial systems:
• lenses, mirrors, irises • flats, spheres, conic sections
• exact 3-D monochromatic trace • convenient built-in table editor • 2-D system and ray layouts
• diagnostic plots • Monte-Carlo spot diagrams
• least-squares optimizer • CGA, EGA, VGA, & Hercules

$89 postpaid if prepaid, quantity 1–9
CA add 7% sales tax

BEAM THREE

for advanced work:
• all the above functions • 3-D optics placement
• tilts and decenters • polynomials & torics
• diffraction gratings • refractive index tables
• CGA, EGA, VGA, & Hercules

$289 postpaid if prepaid, quantity 1–9
CA add 7% sales tax

STELLAR SOFTWARE
P.O. BOX 10183
BERKELEY, CA 94709

Circle number 16 on Reader Service Card

Acknowledgements: I am grateful to Jim Callahan, Mahlon Kelly and Larry Forsley for advice and encouragement. ■

FOR FURTHER READING

[1]Dick Pountain, *Object-oriented FORTH*, (Academic Press, London, 1987).

[2]B. Palmer, *FORTH: The Choice for Scientific Instruments*, Computers in Physics, Mar/Apr 1988, p. 54.

[3]W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipes* (Cambridge University Press, Cambridge, 1986), p. 289 ff.

[4]R. Duncan and M. Tracy, *The FORTH Vendors Group Standard Floating Point Extension*, Dr. Dobb's Journal, September 1984, p.110.

Journal of FORTH Applications and Research 5, 257 (1989), and to be published.

Michael Ham, Dr. Dobb's Journal, October 1986.

L. Brodie, *Starting FORTH*, 2nd ed. (Prentice-Hall, NJ, 1986);

L. Brodie, *Thinking FORTH* (Prentice-Hall, NJ, 1984).

M. Kelly and N. Spies, *FORTH: a Text and Reference*, (Prentice-Hall, NJ, 1986).