Adventures in the Forth Dimension

Julian V. Noble

Forth is a language that, for most programmers, is like the back side of the Moon: they know it is there but have never laid eyes on it. Yet Forth is mature (about as old as Lisp or C); ubiquitous in everyday applications such as automated tellers, package tracking, airline reservations, telephone switching systems, and microcontrollers everywhere; and built into millions of personal computers and workstations in the guise of IEEE Open Firmware. The page description language PostScript also has its roots in Forth.

Since Forth is, like the game of Go, simple yet deep, scientists and engineers should be better acquainted with it. This article provides an overview of the language with emphasis on availability[1], ease of use[2], structure, readability, and features considered important or essential in modern languages. I will also discuss execution speed and code size. To compensate for the omissions imposed by brevity I include an Appendix of Web links to sources of further information.

Forth is usually considered a language for embedded programming and control. However it is frequently applied in the less familiar contexts of robotics, data processing, security systems, music, and number crunching. Though not often taught in academic settings, Forth is a good pedagogical language since it offers access to all the computer's facilities and few predefined amenities. A student learns intimately how a heap or linked list works

[1]Taygeta Scientific Inc. posts a large online collection of Forth information, including public domain software, and systems for many machines and environments: http://www.taygeta.com/

[2]Forth tutorials are available at:
http://Landau1.phys.virginia.edu/classes/551/primer.txt
http://www.softsynth.com/pforth/pf_tut.htm

because he must construct it himself. I use Forth as the main illustration language in my course, *Computational Methods of Physics*[3], because program flow is clear, development is rapid, and lessons are reinforced through immediate feedback.

Forth came to my attention in the mid-1980's, at a time when my mainstay, Fortran, had let me down. Since it was easy to learn and a great time saver in program development, I came to prefer it to Fortran or C for virtually all numerical work. For applications such as symbolic manipulation I have found it to be in a class with Lisp. One reason I chose Forth over C or Pascal is that it is easily extended to include both data structures and arithmetic operators for complex arithmetic (or quaternions, or anything else one might conceive).

The Forth language is both compiled and interpretive, with a simple grammar: programs consist of sequences of words and numbers, separated by spaces (blank characters). The words are subroutines that are executed when named. Since the language is interactive, a program can be typed in at the terminal; or it can be read from a file *via* commands like

```
FLOAD  filename.ext
```

or

```
INCLUDE  filename.ext
```

Again, I find the root of a transcendental equation by invoking my hybrid *regula falsi* solver:

```
: f1     FDUP  FEXP  F*  1.0e0  F-  ;  ok
USE( f1 0e0 2e0 1e-6 )FALSI   FS. 5.67143E-1  ok
```

The first line defines the function $f_1(x) = xe^x - 1$; the second line finds the root of `f1` and displays it to the screen. Of the other arguments, `0e0` and

[3]http://www.phys.virginia.edu/classes/551/

`2e0` are literal floating point numbers delimiting the range of the search, and `1e-6` is the desired precision of the result. The words `USE( , )FALSI` and `FS.` are subroutines that find the root and display the result. "ok" is what the interpreter says if all went well.

Forth is incrementally compiled: each subroutine—such as `f1` above—compiles as it is entered. Since most Forth subroutines can be executed interactively, we can test a new definition as soon as it has been entered, without writing a separate test program or elaborate scaffolding. This accelerates the program development cycle relative to traditional languages like C or Fortran. My own experience indicates a 10-fold speedup over Fortran; Forth programmers who are also C-adept claim a similar ratio over development in C.

Two key themes underlie Forth's simplicity of structure and grammar. First, Forth simplifies communication between subroutines by providing direct access to the cpu stack. Conventional languages like C or Pascal communicate *via* temporary "stack frames" that hold arguments and parameters. A calling program constructs the frame and saves the current state of the system on it, before transferring control to the subprogram; upon exit the system state is restored and the frame deconstructed. This elaborate communication protocol can double or treble the execution time of a short subroutine. For programs in such languages to reduce significantly the subroutine calling overhead, they must reduce correspondingly the number of subroutines called. That is, languages with significant communication overhead necessarily encourage programs featuring lengthy, versatile subroutines that perform several actions. Further speed optimizations are obtained by "in-lining" functions (rather than treating them as subroutines), "unrolling" loops, and other hallowed tricks that trade memory usage for execution speed.

Direct access to the stack lets Forth subroutines expect their arguments—if any—on the stack, and leave their results—if any—on the stack. This leads naturally to a "reverse-Polish" or "postfix" programming style, in which arguments precede functions, as with the HP family of RPN calculators. Using the stack directly to communicate between subprograms eliminates the stack-frame overhead and encourages programs consisting of small subroutines performing single functions. Program flow becomes clearer and debugging simpler. Repeated operations are generally turned into subroutines and

given their own names. This *factoring* process, the opposite of in-lining, shortens source codes and their corresponding executables.

The second key to simplicity is that Forth recognizes only numbers and subroutine names. That is, operators, compiler directives, control structures, data structures, commands, functions—in fact any programming constructs yet conceived—are subroutines that do their jobs when executed. For example, a named `VARIABLE` is a subroutine that places the address of the storage allocated to it on the stack. We define and test a new `VARIABLE` *via* the input lines

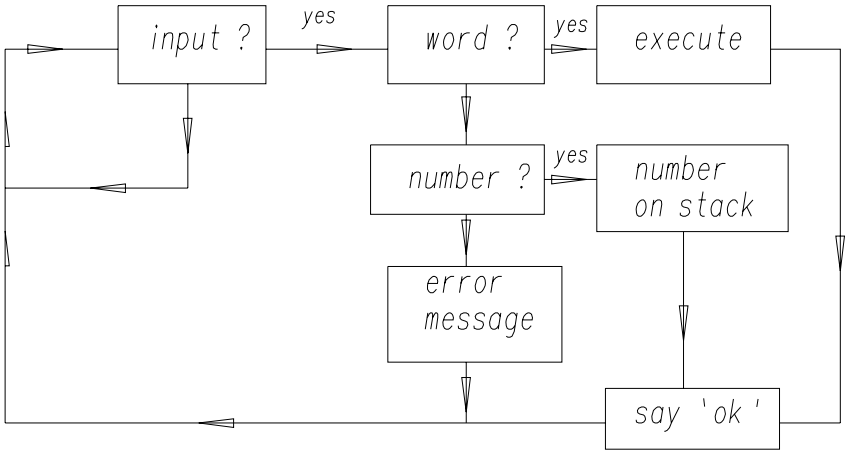
```
VARIABLE x  ok
7 x !  ok
x @  .  7  ok
```

The first line defines a variable named `x`. The second line inputs the string 7 and converts it to an integer (which is pushed on the stack), then executes `x` (which puts its own address on the stack). The subroutine `!` (pronounced "store") expects a memory address on top of the stack and a number just below it. It stores the number to the address, consuming both stack items. The third line executes the new variable `x` (putting its address on the stack) and then executes the subroutine `@` ("fetch"), which consumes an address on the stack and replaces it with the contents of that memory location. The subroutine "dot" (.) consumes the number on top of the stack—in this case, 7—and displays it on the standard output device—in this case the CRT.

Another standard data structure, `CONSTANT`, is a subroutine that places the contents of its storage on the stack. We define and test a new `CONSTANT` by saying

```
17 CONSTANT x  ok
x .  17  ok
```

Now I turn to the structure and compilation method of Forth. A Forth interpreter is a simple endless loop that awaits input as shown below:



Each incoming item of text, delimited by spaces, is interpreted according to the following rules:

1. Search the dictionary to see whether it is the name of a previously defined subroutine (a *word* in Forth parlance, since it is kept in a dictionary).

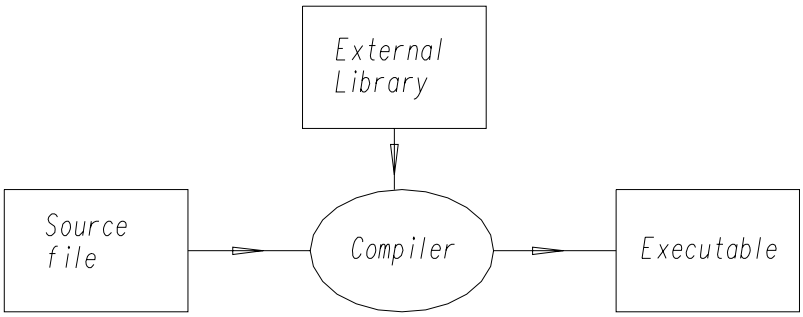
2. If found, pass its code address (or *execution token*) to the subroutine `EXECUTE` which executes the found subroutine, then issue the message "ok".

3. If not found, the subroutine `NUMBER` tries to convert the text to an integer (in the current base), and if successful pushes the result onto the stack and says "ok". (In a system extended to include floating point operations, `NUMBER` is modified so it recognizes a floating point number—such as `2e0`—converts it appropriately, and pushes it onto the floating point stack. Of course this requires the system to be set for base-ten arithmetic, since 2e0 is a perfectly acceptable hexadecimal integer.)

4. If the text cannot be interpreted as a number, issue an error message.

5. Return to the beginning of the loop and await the next item of input.

I have been speaking of the interpreter. But what of the compiler? In most compiled languages the compiler is a separate program that ingests a file of source code, links it appropriately to external libraries, and outputs the executable, as shown schematically in the following figure:



The user controls some aspects of compilation with "compiler directives"—commands input with the source file—but generally cannot extend or modify the compiler itself without rewriting it completely.

The Forth compiler is part of the interpreter, which actually has two states, *interpret* and *compile*. The system variable `STATE` is `FALSE` when the system is interpreting, `TRUE` when it is compiling. The state determines what `EXECUTE` and `NUMBER` do: during interpretion both act as described above. But when the system is in *compile* mode, `EXECUTE` records the address of a routine instead of executing it; and `NUMBER`, after converting the numeric text to a number, records it and installs code that will place the number on the stack during execution of the newly-minted subroutine.

Compilation is most easily explained with an example. We can use Forth as an interactive RPN calculator, as with the input line

```
3 4 5  *  +  .   23  ok
```

What happened? The interpreter interpreted sequentially the strings 3, 4 and 5 as integers and placed them on the stack (with the most recent, 5, on top). The subroutine $*$ then multiplied the top two integers (4 and 5), consuming

them and leaving their product (20) on the stack. Next the subroutine + added the top two integers (now 3 and 20), consuming them and leaving their sum (23). And finally, the subroutine "dot" (.) printed the top integer on the stack to the display. Having executed this sequence of operations successfully without running into undefined text, the interpreter said "ok".

Suppose the sequence `* +` appeared many times in a program. It might be useful to define a distinct subroutine that does both[4]:

```
: *+    *  +  ;   ok
```

We call this compilation. A new subroutine definition begins with "colon" (:) which is itself a subroutine. Colon interprets the first piece of text following it in the input stream as the name of a new subroutine, creating a new entry (under that name) in the dictionary. Colon then switches the interpreter into the *compile* mode. In this state, a word located in the dictionary is not executed: instead the address of its code is copied into a list (that is, *compiled*) in the body of the new subroutine. (A literal number would be copied into the same list, along with the address of a special piece of code—`LITERAL` —that places the number on the stack when the new subroutine is executed.) In this example, pointers to the code of $*$ and + were compiled. The terminal semicolon (;) is an `IMMEDIATE` word that executes even when the system is in *compile* mode. Semicolon installs some terminating code, then switches the system back to *interpret* mode. The "ok" shows semicolon encountered no snags. This compilation method is called *threading*.

Now we test—enter the same three numbers, followed by the newly defined subroutine `*+`, to see whether it works correctly:

```
3 4 5  *+    .   23  ok
```

It gives the correct answer and says "ok", so `*+` is now a tested and debugged subroutine.

The preceding example illustrates several important points:

[4]. . . for example `1+` increments the integer on top of the stack by 1, replacing the sequence `1 +`

- Forth does not restrict the characters forming the name of a subroutine. Good Forth style employs telegraphic names like `*+` that suggest what the subroutine does—one route to programming clarity.

- A Forth subroutine looks and acts just like interpreted Forth: a sequence of numeric literals and subroutine names that is executed left to right.

- A new subroutine compiles as it is entered, therefore can be tested immediately.

- Testing is simple: subroutines generally expect their arguments on the stack and leave their results there. Therefore testing consists of placing appropriate arguments on the stack, then invoking the subroutine. If it produces the right answer (without unexpected side effects) it can be considered debugged.

- A Forth subroutine, once defined, becomes part of the language, on a par with the routines that came predefined with the system. There is no distinction between "user-defined" and "system"—which is why Forth is so easily (and often!) modified by its users.

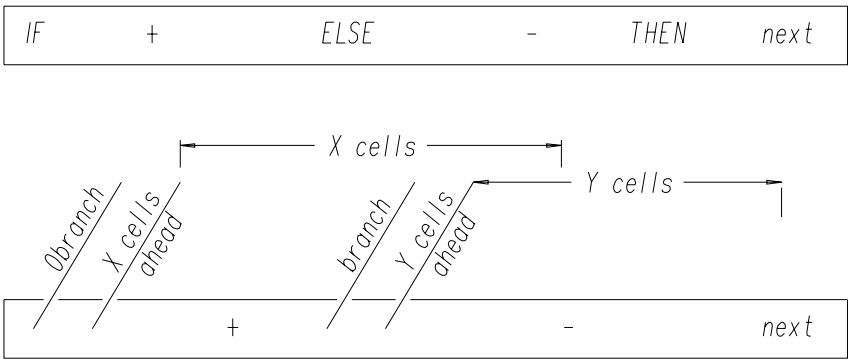
- A Forth program consists of subroutine definitions, ending with the subroutine that will actually perform the actions one wants. Thus a program to solve linear equations looks like

```
\ ... previous definitions of initialize,
\  triangularize, back_solve  and report

: }}solve     ( A{{  V{ --)
    initialize  triangularize  back_solve  report  ;

\ usage:
\     A{{  V{  }}solve
\     A{{ is the matrix and V{ the inhomogeneous term;
\     --both are overwritten.
\     The curly braces in the names are syntactic sugar
\     suggesting matrix manipulations according to the
\     conventions of the Forth Scientific Library project.
```

It is hardly surprising that arithmetic operators like + or * —or their floating point equivalents F+ or F* —are subroutines. But how can this be true of control structures? Conventional compilers handle control structures like IF ...ELSE ...THEN or DO...LOOP by parsing them as tokens and making decisions: was it an IF ? If so, install this piece of code, otherwise do something else, *etc.* In Forth, on the other hand, control structures are IMMEDIATE subroutines (that execute during compilation). A subroutine incorporating control structures might look something like

```
: +OR-    IF  +   ELSE  -  THEN    ;  ok
```

As we have seen, the colon (:) initiates, and the semicolon (;) terminates, compilation of +OR-. The IMMEDIATE subroutine IF lays down code that will branch when +OR- executes, and temporarily saves the address of that code on the stack. Compilation then continues normally (in this case compiling the subroutine +) until ELSE is reached. This word is also IMMEDIATE, hence executes to resolve the forward reference. It uses the saved address to compute how far forward the first branching code must jump when presented with a FALSE flag, and stores that information within the newly laid down branching code following IF (that is, branches are implemented by jump-relative instructions). ELSE then installs code for an unconditional forward jump and saves the address of this code on the stack. Words following ELSE are compiled normally until THEN is reached. This IMMEDIATE word (alias ENDIF in some Forths) then resolves the second forward reference by computing the length of the unconditional jump and storing it in its proper cell. The figure below shows schematically the source code and beneath it the jumps it installs (the conditional jump is called 0branch and the unconditional one is called branch):



This kind of behavior—computing rather than deciding—is typical of Forth [1], but quite different from the mechanism of conventional compilers. Testing, we have

```
3 5 TRUE  +OR- .  8  ok     \ added
3 5 FALSE +OR- .  -2  ok    \ subtracted
```

When the code installed by `IF` finds a `TRUE` flag on the stack it does nothing, thereby allowing the subroutines between it and `ELSE` to be executed. The unconditional branch installed by `ELSE` then causes execution to resume at the subroutine following `THEN`. With a `FALSE` flag (as in C, `FALSE = 0`), however, the branch skips the words between `IF` and `ELSE`, resuming execution at the first word past `ELSE`. As the diagram makes clear, no code is installed by `THEN` so execution continues as though `THEN` were not there.

Control structures for indefinite (`BEGIN...UNTIL`, `BEGIN...WHILE...REPEAT`) and definite `DO...LOOP` loops work more-or-less the same way.

Forth is a minimal language, which is why it has been implemented on so many different kinds of computers. A Forth can be reduced to somewhere between 20 and 30 routines that must be defined in machine language; the rest can be defined in terms of this basic kernel. The ANS Forth Standard[5] (1994) lists 133 subroutines in the `CORE` wordset. This is the number required

[5]... available in various formats including HTML

for a system to advertise itself as ANS-compliant. Whether a vendor wishes to provide only 20-30 machine-language primitives and define the remainder in Forth; or to provide optimized code for all 133 required CORE words is up to her.

The 1994 ANS Standard specifies 372 subroutines altogether, mandating standard names and behaviors for integer and floating point arithmetic; exponential, logarithmic, trigonometric and hyperbolic functions; file access; exception handling; generic keyboard and display I/O; generic memory management; and the built-in assembler, if present (most Forths provide one).

When I call Forth minimal, I mean it lacks predefined common constructs like linked lists or C-like *struct*s. However they can be added with little effort, since both commercial Forths and the vast body of public domain Forth programs (available on the Web) provide Standard implementations of queues, deques, stacks, heaps, lists, and so on. The Forth Scientific Library is a growing compendium of tested code for all the usual algorithms. In other words, availability of code examples and libraries is really no hindrance to someone who wants to use Forth.

Forth provides features lacking in other languages, such as the ability to specify the current arithmetic base for number conversion. Some algorithms are more simply expressed in octal or hexadecimal arithmetic, so the ability to change bases freely is a boon. Standard Forth also permits executing strings of Forth code as though they were input from the command line or a file. This allows straightforward creation of macros, not to mention (safe) self-modifying code. (The latter is usually considered unsafe practice, but it can be very useful in artificial intelligence programming or language translation.)

Conspicuously absent from the 1994 ANS Standard are standardized names and behaviors for complex arithmetic, graphics, GUI construction, or port access. This is hardly surprising since cross-platform standards for such things do not exist for any language. (I do not wish to imply that no Forths provide such features—many do. It is just that they have not yet been standardized and are not portable.)

One of Forth's nicest features is that it hides nothing. Compiler words

like : and CODE are comprised of components programmers can use freely. This makes it easy to modify or extend the compiler, or for that matter, to define multiple compilers for specialized tasks. Using the dictionary look-up subroutine "tick" (') and the subroutine "comma" (,) —that stores an integer on the stack into the next unused memory cell—we can, for example, hand-construct a jump table for rapid execution of multiple-choice programs by finding and storing the execution tokens of the subroutines in the table. However, if we need many jump tables, it might be better to define a mini-compiler of jump tables (a *constructor*, in other words) using components of Standard Forth, rather than constructing them individually:

```
: jtab:  ( Nmax --)      \ starts compilation
    CREATE               \ make a new dictionary entry
    1-  ,                \ store Nmax-1 in its body
;                        \ for bounds clipping

: >xt_address    ( n base_adr -- xt_addr)
    DUP  @       ( -- n base_adr Nmax-1)
    ROT          ( -- base_adr Nmax-1 n)
    MIN  0  MAX    \ bounds-clip for safety
    1+  CELLS+   ( -- xt_addr = base + 1_cell + offset)
;

: ;jtab   DOES>  ( n base_adr --)   \ ends compilation
                  >xt_address
                  @  EXECUTE  \ get token and execute it
;    \ appends table lookup & execute code

\ Example:
: Snickers   ." It's a Snickers Bar!"   ;   \ stub for test

\ more stubs

5 jtab:  CandyMachine
         ' Snickers  ,
         ' Payday    ,
         ' M&Ms      ,
```

```
          ' Hershey   ,
          ' AlmondJoy ,
;jtab

 3 CandyMachine  It's a Hershey Bar!   ok
 1 CandyMachine  It's a Payday!   ok
 7 CandyMachine  It's an Almond Joy!   ok
 0 CandyMachine  It's a Snickers Bar!   ok
-1 CandyMachine  It's a Snickers Bar!   ok
```

In my own work I have found mini-compilers a very useful and powerful technique:

- My book [2] describes a compiler of self-executing tables of random variates from arbitrary distributions—useful in Monte-Carlo simulations.
- When I translate formulas to reverse-Polish notation I must include comments containing the original formulas (or else I would not be able to make head or tail of the subroutine six months later!). At some point I realized that a FORmula TRANslator—in the form of a mini compiler—would let me eliminate the comment.
- The key to a compact recursive-descent FORmula TRANslator (about 500 lines of code and comments; 727 including white-space lines, documentation and conditional compilation sections) was the ability to construct finite state machines *ad libitum*. I wrote a mini compiler in a few lines of code, that compiles the desired FSM (expressed by a two-dimensional state-transition table) as a self-actuating jump table [3]. This is such a clear, efficient and expressive way to program FSMs that it has been applied to text processing and language translation, to computer game programming, and to gas/oil pipeline controllers, to name only applications I know about.
- The FSM mini-compiler was also useful in programs for γ-matrix algebra, as well as for multi-variate function minimization by the simplex algorithm.

- Mini compilers are also used (not by me!) for creating optimized native machine code on the fly, for example when programming embedded processors.

I am often asked how Forth stacks up against other languages. This depends on the criteria of comparison. So far I have tried to show that Forth is easy to obtain, use and understand. The next section addresses the areas of execution speed, development time, code size, structure, readability, and features considered important or essential in modern languages.

Forth codes typically run 3 to 10 times slower than (optimized) C or Fortran equivalents. In other words, unoptimized Forths, such as the public domain systems I have been using, are about as fast as unoptimized Lisp or C++. There are, however, several ways to achieve fast execution in Forth. Simplest is to invest in an optimizing native code compiler; there are several available whose execution of numeric-intensive software compares favorably with good C compilers—that is, about 1.5 to 2× slower than hand-tuned machine code.

For such specialized applications as linear equations, differential equations or Fourier transforms, commercial packages like MatLab® provide optimized libraries of code subroutines. These libraries can be linked to and called from Forth programs running under Linux or Windows using standard tools [4]. To use them one must of course know their matrix labelling and subroutine calling conventions.

Traditionally we optimize Forth for speed by identifying bottleneck subroutines using algorithmic analysis or a profiler, then rewriting the bottlenecks in assembler. Since Forths usually include assemblers, this is an easier route to take than it would be with other languages. In fact writing and testing assembly language subroutines is much easier in the Forth environment than in any other I know of [5] since neither test program nor linking step is required. A Forth subroutine defined in `CODE` operates exactly the same way as its high level equivalent. I have recently used this approach in the LU algorithm for linear equations. The innermost loop of the latter (repeated twice in the Fortran subroutine `LUDCMP` [6]) is a good candidate to be factored out and defined as a separate `CODE` subroutine since it contains the only instructions executed $\mathcal{O}(N^3)$ times and therefore dominates the asymptotic

running time. The instructions in this loop are also simple (two fetches, a multiplication and a subtraction) so it is easy to hand-code. Further optimization affects only the $\mathcal{O}(N)$ and $\mathcal{O}(N^2)$ terms in the running time. In other words, a single short `CODE` definition lets the Forth version of LU solve large dense systems at the maximum speed attainable by a non-vectorized machine.

Most often, however, my goals are development speed and program correctness rather than execution speed *per se*. By these criteria Forth beats the other languages I know. A typical instance was a program I wrote to simulate vehicular rollover accidents using realistic forces. It took me less than a day to write and debug the code, and a couple of hours to add a graphical display. From prior experience, I do not think I could have achieved this with Fortran in less than a week. Similarly, having never written any sort of parser or compiler before, I created a usable recursive-descent FORmula TRANslator in less than one week.

Since I am a physicist who programs, not a professional programmer, I cannot speak for the latter group. However, professionals report similar experiences. For example, the IEEE Open Firmware Specification evolved from Mitch Bradley's (secret) use of Forth to develop hardware drivers at Sun Computer Corp. According to Bradley, development went much faster if he prototyped in Forth and converted to the in-house language afterward. Again, the (public domain) Windows-compliant Win32Forth was written by Tom Zimmer because he found the Microsoft Windows SDK *cum* C++ too slow to meet a promised deadline. He delivered his code on time by first creating an object-oriented Forth, then using it to construct Windows programs.

Forth programs tend to be more compact than their equivalents in other languages. This is true of both source and executables. For example, the 16-bit DOS-based Forth (that was my mainstay before Windows) had a 32 Kbyte executable. The Windows compatible Win32Forth has a 52 Kb executable and a 400 Kb runtime library, which sounds like a lot until one compares it with Corel WordPerfect® 6.1 which is 3,775 Kb; or with a "lite" version of Borland Turbo C++ at 834 Kb. These are summarized in the Table below:

Program	**Size (Kb)**	**DLL's (Kb)**
HSFORTH 4.2 (16-bit)	32	
GForth 0.40 (32-bit DOS)	176	
Aztec Forth (32-bit DOS)	133	
Win32Forth 3.5	452	288
WinView	410	288
Turbo C++ Lite	834	
Ventura Publisher 4.2	997	1017
WordPerfect 6.1	3755	3029

That is, Windows programs tend to be large—Forth-based Windows programs somewhat smaller. Now when it comes to source code size, the source for WinView, an extensive Windows editor included with Win32Forth, is 268 Kb. The source for the assembler (for Intel cpu's and fpu's) is about 80 Kb. And the source for my FORmula TRANslator is 24 Kb (including extensive comments).

What about readability and maintainance? At one time structured programming was a central goal of computer pedagogy. Nicholas Wirth invented Pascal in reaction to "spaghetti code" produced by students. Wirth aimed to eliminate line labels and direct jumps (`GOTO`s), thereby forcing control flow to be clear and direct and making spaghetti code impossible. Paradoxically, Forth is the only truly structured language left in common use today, although that was not its raison d'être. It contains neither `GOTO`s nor line labels. A Forth subroutine has a single entry and a single exit point, and (usually) performs a single job.

Like every language, Forth can be written obscurely. I have certainly seen plenty of underdocumented, badly formatted, badly factored code with poorly named and excessively verbose subroutines. Of course this is true of C, Fortran or any other language—it is as easy to produce "write-only" code as to write muddy prose. Standard Forth offers the usual remedies: comments, stack diagrams, sensible and telegraphic naming conventions, proper layout. Names can help a lot, as in

```
: }}solve     ( A{{  V{ --)
    initialize  triangularize  back_solve  report  ;
```

from the linear equations example. The subroutine names portray the algorithm clearly; more comments would add nothing. Once again, a naming convention lent transparency to the *regula falsi* root finder:

```
USE( f1  0e0  2e0  1e-6 )FALSI  FS.
```

However Forth also makes available some unusual remedies, that with care can produce exceptionally clear programs. A self-executing jump-table (compiled by the mini compiler `jtab: ... ;jtab`) *looks* like a table:

```
5 jtab:  CandyMachine
         ' Snickers  ,   \ item 0
         ' Payday    ,
         ' M&Ms      ,
         ' Hershey   ,
         ' AlmondJoy ,   \ item 4
;jtab
```

—hardly any further comment or explanation is needed.

One aspect of Forth disconcerting for the newcomer is its lack of safety features. Some Forths perform rudimentary stack checks during compilation, but this is by no means mandated by the Standard and is provided at the discretion of the vendor. Since arrays in Forth are defined as needed by the programmer, they can incorporate bounds checking or not, as desired. The jump table mini compiler prevented out-of-bounds indexing by clipping—as good a way as any.

The omission of array bounds checking is not mere hacker machismo, however. Bugs that cause memory leaks are much rarer in Forth than in C. Forth errors tend to involve the stack—removing too many items or too few. Obviously a word that uses the stack incorrectly inside a moderately long loop can crash with the greatest of ease. A simple discipline of keeping word definitions short enough to understand, commenting them thoroughly—especially their stack effects—and testing each word as it is defined, eliminates the bulk of these errors. That is, I do not consider Forth programs inherently unsafe,

despite the dearth of safety nets. And one professional programmer I know specializes in safety-critical applications using Forth [7].

A propos of safety, what about debugging? "Serious" languages nowadays come with an "environment" that includes integrated editing and code debugging of various levels of sophistication. Although many Forths provide single-stepping code debuggers, and some provide integrated editors, experienced Forth programmers mostly do without these appurtenances. Again this has little to do with machismo, but a great deal to do with the way programs get written. Most subroutines are short enough to be correct the first time. They can be tested as they are entered, and quickly reveal errors, omissions or oversights. I have needed debuggers only a few times, usually because I let a subroutine get too verbose. Some languages permit "assertions" [8] that specify conditions—usually loop invariants—that must be satisfied at various points in the program. This is considered a key to guaranteeing program correctness. Assertions have been implemented in Standard Forth, but I think more for the challenge than because they were needed. They may be valuable in a large C program, but in Forth assertions seem like overkill.

Some consider modularity essential in "serious" languages. Languages that support modularity enable multi-programmer teams to develop different parts of a large code in separate modules, joining them only at the end. This programming paradigm is doubtless the source of the enormous improvements we have witnessed of late, in the quality, ease of use, and reliability of commercial applications. ;-)

Well, Forth can be made modular also. One of the most successful commercial Forths for large 16-bit DOS applications [9, 10] was designed around this concept. In fact, modularity—in whatever strength you need—is relatively easy to implement in Forth. The weakest form, suitable for most applications, is based on ANS Forth's support for partitioning the dictionary into distinct *wordlists*. Repeated subroutine names are not a problem because the compilation mechanism can be told which wordlists not to search. Many Forth programmers habitually distinguish public and private resources by defining the words `public` and `private`, that execute appropriate search-order switches.

Discussions in the `comp.lang.*` newsgroups often revolve around memory

management, particularly "garbage collection". As most readers will know, the need for this arises from dynamic memory allocation within a heap. It is not obvious how best to de-allocate memory that is no longer needed, since it can be physically located amidst memory that must be retained; worse, it is not always easy to determine whether a given chunk is, in fact, ready for reclamation. Some languages—such as Lisp—are more afflicted by garbage collection problems than others. In Forth, I am happy to report, garbage collection is mainly a non-issue. We tend to eschew variables, leaving temporary items on the stack (or at any rate, on *a* stack); this space is reclaimed immediately when a word exits. Memory allocation is directly under the programmer's control, so de-allocating it can be handled in the way best suited to the specific application.

Because Forth's programming philosophy runs to simple subroutines that do single tasks, arithmetic operators are not overloaded. The ordinary multiplication `*` is not the same as the floating point operator `F*`. Moreover since there is no telling what might be on the stack at a given moment, we do not expect `*` to recognize when a floating point number must be multiplied by an integer. To simplify some of my programs I once implemented a form of operator overloading to allow mixed real/complex arithmetic. Since the decisions take place at run time rather than compile time (late, not early binding) this slows program execution about 15%.

Forth may be the most portable language in use today. I have yet to hear of a program written entirely in ANS Forth that fails to perform correctly on another platform running ANS-compliant Forth. A mostly compliant program using a few non-Standard or environmentally-dependent subroutines generally can be ported successfully with little effort. Moreover, few computers cannot boast at least one ANS-compliant Forth.

Finally, there is the matter of the programming "style" that a given language encourages and/or supports. Raw Forth can look almost like assembly language—in fact Forth has often been dismissed as "nothing more than a high-level assembly language". But Forth's enormous extensibility has allowed it to take on many guises. Lisps, Prologs, BASICs, SmallTalk's and C's have been written in Forth, sometimes for serious reasons and sometimes just for fun. Standard Forth provides the tools to construct any linguistic paradigm one wishes, even object orientation with polymorphism and inheri-

tance: numerous Forths are object-oriented (but there is as yet no concensus in the Forth community either that such refinements are necessary or desirable; or, if they are, what the standard interface should be).

Many of my own programs look like the evaluation of arithmetic formulae, *a la* Fortran, because I use a FORmula TRANslator. For example, here is how I translate a short Fortran subroutine [11] to Forth:

```
      SUBROUTINE TRIDAG(A,B,C,R,U,N)
      PARAMETER (NMAX=100)
      DIMENSION GAM(NMAX),A(N),B(N),
     #          C(N),R(N),U(N)
      IF(B(1).EQ.0.)PAUSE
      BET=B(1)
      U(1)=R(1)/BET
      DO 11 J=2,N
        GAM(J)=C(J-1)/BET
        BET=B(J)-A(J)*GAM(J)
        IF(BET.EQ.0.)PAUSE
        U(J)=(R(J)-A(J)*U(J-1))/BET
11    CONTINUE
      DO 12 J=N-1,1,-1
        U(J)=U(J)-GAM(J+1)*U(J+1)
12    CONTINUE
      RETURN
      END
```

```
include arrays.f
include ftran111.f

100 VALUE Nmax

Nmax long 1 FLOATS 1array a{    \ input array
Nmax long 1 FLOATS 1array b{    \ as 3 vectors
Nmax long 1 FLOATS 1array c{

0 VALUE aa{   0 VALUE bb{   0 VALUE cc{   0 VALUE NN

Nmax long 1 FLOATS 1array r{    \ inhomogeneous term

Nmax long 1 FLOATS 1array L{    \ diagonal
Nmax long 1 FLOATS 1array U{    \ lower subdiagonal
Nmax long 1 FLOATS 1array x{    \ solution vector

: }triangularize    ( a{ b{ c{ n --)
    TO NN   TO cc{  TO bb{  TO aa{
    f" bb{0}"
    FDUP  F0=  ABORT" Reduce # of equations by 1"
    f" L{0}"  F!
    f" U{0} = cc{0} / L{0}"
    NN 1-  0  DO  f" U{I} = cc{I} / L{I}"
                  f" L{I_1+} = bb{I_1+} - aa{I_1+} * U{I}"
    LOOP   ;

: }backsolve    ( r{ x{ n --)
    TO NN   TO aa{  TO bb{
    f" bb{0} = bb{0} / L{0}"
    NN 1 DO   f" bb{I} = (bb{I} - a{I}*bb{I_1-}) / L{I}"
    LOOP
    f" aa{NN_1-} = bb{NN_1-}"
    0 NN 2 -  DO   f" aa{I} = bb{I} - U{I}*aa{I_1+}"
    -1 +LOOP   ;

\ say:   a{ b{ c{ n }triangularize  r{ x{ n }backsolve
```

The Fortran looks terser but this is illusory: it lacks comments and white space, not to mention a program to use it. The Forth version runs as is.

The urge to use Forth as if it were something else afflicts programmers

who have begun to appreciate Forth's power, but have not yet abandoned the habits of their previous language(s). I once attempted to write a full Fortran-to-Forth translator. However, as I soon discovered, good Fortran translated literally becomes terrible Forth, rather like Mark Twain's literal re-translation into English of the French version of "The Celebrated Jumping Frog" [12]. Having learned my lesson, whenever I am tempted nowadays to change Forth's innate style, I emulate Bennet Cerf and lie down until the urge passes.

Appendix: Useful Web sites in the world of Forth

MPE Ltd. (http://www.mpeltd.demon.co.uk/)

FORTH, Inc. (http://www.forth.com/)

Online articles and books of interest:

C.H.Moore and G.C. Leach, *FORTH—A Language for Interactive Computing* (http://www.ultratechnology.com/f70c2.html)

Philip J. Koopman, Jr., *Stack Computers: the new wave* (http://www.cs.cmu.edu/~koopman/stack_computers/index.html)

Journal of Forth Application and Research (peer-reviewed online journal) (http://dec.bournemouth.ac.uk/forth/index.html)

The FORTH Research Page. Maintained and validated by Dr. Peter Knaggs (pjk@bcs.org.uk) (http://dec.bournemouth.ac.uk/forth/jfar/index.html)

Phil Burk, *Forth Tutorial* (http://www.softsynth.com/pforth/pf_tut.htm)

Taygeta Scientific Incorporated: large online collection of Forth info (http://www.taygeta.com/)

Forth Interest Group Home Page (http://www.forth.org/)

Bill Muench, *eForth: A simple model Forth system* (http://members.aol.com/forth/)

J.V. Noble, *Computational Methods of Physics* (http://www.phys.virginia.edu/classes/551/)
Most of the programs alluded to in this article can be found there.

A Forth in Java (http://world.std.com/~wware/agj/fj.html)

References

[1] J.V. Noble, "Avoid decisions" *Computers in Physics* **5** #4 (1991) 386.

[2] J.V. Noble, *Scientific Forth: a modern language for scientific computing* (Mechum Banks Publishing, Ivy, VA, 1992) pp. 61-64.

[3] J.V. Noble, "Finite state machines in Forth", *Journal of Forth Application and Research* **7** http://www.jfar.org/article001.html

[4] http://www.er.ele.tue.nl/emv/frames/EMV_hendrix.htm

[5] J.V. Noble, "A call to assembly" *Forth Dimensions* in press.

[6] W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vettering, *Numerical Recipes* (Cambridge University Press, New York, 1986) p. 35ff.

[7] http://www.amleth.demon.co.uk/

[8] Jon Bentley, *Programming pearls* (Addison-Wesley Publishing Co., Reading, MA, 1989) p. 42.

[9] Microprocessor Engineering, Ltd. (http://www.mpeltd.demon.co.uk/)

[10] Paul Frenger, "Learning Forth with Modular Forth", *ACM/SIGPLAN Notices* **35** #3 (2000) 25.

[11] W.H. Press, *et al.*, *ibid.*, p. 40.

[12] Samuel Clemens, *The family Mark Twain* (Dorset Press, New York, 1988), pp. 1163ff.

Julian Noble is a professor of physics at the
Commonwealth Center of Nuclear and Particle Physics
University of Virginia, Charlottesville, Virginia 22901

Figure captions

1. Structure of the Forth interpreter
2. Conventional compilation
3. Jump structure of `IF ...ELSE ...THEN`