Toward Scientific FORTH

Contents

This book presents extensions to the FORTH programming language suitable for scientific and technical computation. The aim is to retain FORTRAN's good points while taking advantage of the simplicity, flexibility, extensibility, and control offered by FORTH.

The resulting dialect has many advantages over more traditonal languages, both for small, casual, throw-away programs as well as for large, complex projects. FORTH lends itself to many programming styles, including procedural, object-oriented or event-driven. Its speed and economical use of memory suit FORTH for real-time, on-line data pre-processing as well as as off-line analysis and computation.

Because FORTH is a **threaded, interpretive** language[1] its structure and philosophy differ radically from those of traditional languages like FORTRAN and BASIC. This introductory chapter

1. This description refers to FORTH's compilation scheme. See, *e.g.*, R.G. Loeliger, *Threaded Interpretive Languages* (Byte Publications, Inc., Peterborough, NH, 1981). We shall have more to say about it in Chapter 2.

explains some of the differences by contrasting FORTRAN with FORTH.

§1 Overview of FORTRAN

FORTRAN is a **compiled** high level language[2]. The programmer writes a **source code** program using FORTRAN's grammatical rules, data structures and operators; a special computer program (the **compiler**) then translates the source into a (relocatable) machine language version (**object code**). Another machine language program (the **linker**) then links modules of object code into an **executable** program that can be run under the control of the operating system of the computer.

Compilation produces executable programs that run fast, without the tedium of writing them directly in machine code (or **assembly language**). The source code will run virtually the same on any machine for which a compiler exists. That is, the source code is **portable**. Among other things, portability makes possible the development of standard libraries of reusable code for performing standard tasks like solving linear equations or computing Bessel functions.

The chief disadvantage of compilation is its tedium. Testing small portions of a program in isolation is virtually impossible – either an "exercise" program must be written and compiled with the module being tested, or else the entire program must be compiled as a unit. This process is so time-consuming it discourages fine-grained decomposition of programs into small, comprehensible components.

§§1 Programs and sub-programs

A FORTRAN program consists of a master, or **main** program that either stands alone or can **call** (transfer control to) sub-programs. Sub-programs fall into two classes: **subroutines** and

---

2. Although some interpreted FORTRANs such as WATFOR have been developed.

functions. Both receive **arguments** (input) from the calling program; they differ in how **results** (output) are returned to the calling program. Subroutines are called by the phrase

```
CALL SUB1(A,B,RESULT)
```

where A and B are arguments and RESULT is the result (which is returned in the argument list). By contrast, a function is called by having its name placed in an arithmetic expression. When the expression is evaluated, the value of the function (at its given arguments) is inserted in the expression where the function name appeared. That is, we might have a phrase like

```
OPSIDE = HYPOT*SIN(3.14159*ANGLE/180.).
```

Here the argument of SIN is also an expression which must be evaluated before being passed to the SIN subroutine. When SIN is evaluated, its value is returned, multiplied by HYPOT and the product stored in the area labelled OPSIDE.

There is no specific calling heirarchy in FORTRAN – a function can call a subroutine or *vice-versa* and the called sub-program can call still further sub-programs.

§§2 Arithmetic statements

FORTRAN arithmetic is performed by "smart" operators acting on **typed variables** and **literals**. A variable is simply a name that refers to a specific location in memory. The **type** declaration is a way to let the compiler know how much memory to allot for that variable. A literal is an explicit number that appears in the program, such as the values 3.14159 and 180. in the preceding example.

FORTRAN arithmetic expressions can freely mix types. To make this possible, the arithmetic operators are **overloaded** in the sense that the plus sign –say– can add floating point numbers, integers or complex numbers in any combination, mixture or order.

Consider, *e.g.*, the actions performed by the FORTRAN compiler in parsing the arithmetic assignment statement

```
A =B1*3 +B2*1.2E-5 - H(3)/3.14159265358979D14 + K
```

keeping in mind that in FORTRAN, data types can be declared explicitly or implicitly[3] :

- Define and reserve space for a floating-point single precision variable A (implicit type REAL) if A has not been defined previously (perhaps as something else);
- Convert the literal integer constant 3 to floating point and multiply it by the (implicit-REAL) variable B1's current value (fetch from memory), placing the product in temporary storage (TEMP).
- Fetch (implicit-REAL) variable B2 and multiply it by the REAL literal 1.2E-5;
- Add the second product to the contents of TEMP;
- Fetch the 3rd element of the (implicit-REAL) array H;
- Divide by the DREAL (double-precision) literal 3.14159265358979D-14 ($=\pi$), converting to and from DREAL format as necessary;
- Convert the dividend to REAL and subtract from TEMP;
- Convert (implicit) INTEGER variable K to REAL and add to TEMP;
- Move the result from TEMP to the memory reserved for A.

These actions can be over-ridden by explicit type declarations. For example, if the program had contained the following statements in its first few lines:

```
INTEGER A, H(15), B1, B2
REAL K
```

the conversions and assignments would have been floating point to integer, rather than *vice-versa*.

---

3. The original version of FORTRAN included naming conventions such that names beginning with letters I, J, K, L, M and N are assumed to be integers, while those beginning with other letters are assumed to be single-precision floating point numbers. Subsequent versions have maintained this convention for backward compatibility.

To achieve the simplicity of mixed-mode expressions, the FORTRAN compiler must be prepared for any eventuality. The operators "+", "–", "*", "/" and "=" must be "smart" (overloaded) – they must "know" (or at least be able to figure out) what kinds of numbers are going to be used and what kinds of arithmetic will be used to combine them. The FORTRAN exponentiation operator "**" must similarly "know" whether the base is INTEGER, REAL, DREAL or COMPLEX (some FORTRAN's even permit DCOMPLEX), and the same for the exponent. That is, it must be able to compile 16 (or 25) versions of **, depending on circumstances. The compiler must contain decision branches to handle every eventuality. Compilers for languages such as FORTRAN, PASCAL, C or Modula-2 are therefore complex and slow.

Smart operators benefit the user by simplifying source code. The benefit is only partial, however, since the programmer must still keep track of types in calling sequences for subroutines, and in declaring global variables with COMMON and EQUIVALENCE statements.

Since FORTRAN subroutines can be compiled separately, many a subtle bug has been introduced by omitting an argument from a long calling sequence, or by inverting arguments in a list (thereby, for example, telling a subroutine to interpret a REAL as a very large INTEGER). I can vouch for these problems from long, sad experience debugging FORTRAN.

FORTRAN provides a limited suite of data types: INTEGER, LONG-INTEGER, REAL, DREAL, COMPLEX, DCOMPLEX, LOGICAL and CHARACTER. It provides no facilities for defining any new types (other than arrays of the above). Arrays must be declared according to a strict format – up to 3 indices are permitted.

FORTRAN's array notation is simple, logical and follows the conventions of algebra: parentheses replace subscripts *via*

$A_{ij} \rightarrow A(I,J)$.

FORTRAN provides facilities for initializing constants and variables at run-time: the DATA statement within a program or subroutine, and the BLOCK DATA subprogram for initializing global variables in COMMON.

Limited control of memory allocation is provided: placed at the beginning of a program or subprogram, COMMON, BLOCK COMMON and EQUIVALENCE specification statements allow local variables to be made global or partially global, under the same or different names. DIMENSION allocates memory for arrays. (Dynamic re-allocation is not permitted.)

Finally, EXTERNAL directs the compiler (more precisely, the linker and loader) to search outside the subprogram for the specified name: for example, the usage

```
SUBROUTINE MYSUB(X,DUMMY,ANSWER)
```

permits the name of a function or subroutine to be inserted as an argument into the calling string at runtime. This facility is essential to separately compiled modules, of course.

Modern FORTRAN has evolved by accretion, with additions designed not to obsolesce older methods of accomplishing tasks. Thus FORTRAN has several ways to define functions, through external subprograms and through inline definitions; and several ways to allocate memory for arrays. Data types can be changed explicitly *via* functions and implicitly *via* replacement statements, leading to such redundancies as

```
A = FLOAT(K)
A = K
```

or

```
K = IFIX(A)
K = A .
```

§§3 Function library

Crucial to FORTRAN's utility in scientific programming is the mathematical function library, including REAL, DREAL and COMPLEX (at least!) versions of trigonometric functions, exponentials, logarithms, inverse trigonometric functions, some-

times hyperbolic functions and their inverses, and often a random number generator of uncertain quality.

FORTRAN supports modularity through separate compilation of functions and subroutines. For example, we can write a library function to compute complex Legendre polynomials:

```
      COMPLEX FUNCTION CPLEG(Z,N)
      COMPLEX  Z, CP0,  CP1,  CMPLX
      CPLEG  =  CMPLX( 1., 0. )
      IF  ( N .EQ. 0 )  RETURN
      CP0  =  CMPLX( 0. ,  0. )
      K = 0
    1 CP1 = CPLEG
      K1 = K + 1
      CPLEG = ( ( K + K1 ) * Z * CP1  -  K * CP0 ) / K1
      IF  ( K .EQ. N )  RETURN
      CP0  = CP1
      GOTO  1
      END
```

Because all the decisions as to which overloaded operator to use must be made when the function is compiled, a single-precision REAL Legendre polynomial routine will require a separate version from the above.

Worse, because the typical function or subroutine calling sequence wastes memory and execution time, there are severe penalties in efficiency that militate against fine-grained decomposition. That is, the code in one routine is unlikely to be re-used in another routine. Instead, it must be repeated, wasting memory.

§2 What is FORTH ?

When I first encountered FORTH, it appeared to me as Looking Glass Land must have, to Alice. Twenty-five years' experience with FORTRAN colored my perceptions, making FORTH seem very strange indeed.

FORTH makes no essential distinctions between data structures, operators, functions or subroutines. *Every*thing in FORTH is the *same* thing: a **word**. In appearance, words are strings of text separated by spaces. Functionally, words are **subroutines**. To execute a word, type its name, then a carriage return. No GOSUBs, CALLs or RETURNs are needed. This simple grammar is beautiful because it leaves nothing to remember.

Whereas FORTRAN imposes stringent naming conventions – names must begin with a letter, may be no longer than seven characters, and may use only letters and digits – FORTH has no such restrictions. FORTH names can be much more expressive than those in FORTRAN or even Pascal and C, for that matter.

For a preview of FORTH's flavor, consider the FORTH version of the Legendre polynomial function[4]:

```
\ Gx are generic operations (Real or Complex)
: S->FS   S->F   REAL*8   F>FS ;
: PLEG          ( [z] n -- :: -- p[z, n] )
   >R DUP>R  >FS        ( -- :: -- z )
   R@  G=1 R>  G=0 R>  ( -- n :: -- z P1  P0 )
   ?DUP  IF                  \ loop n times, if n >0
   0 DO                      \ begin loop
     I  S->FS   G*           ( :: -- z P1  P0*I)
     FS>F  GOVER  GOVER
     G*  I 2* 1+   S->FS
     G*  F>FS  G-
     I 1+  S->FS   G/        ( :: -- z P1  P2 )
     GSWAP                   ( :: -- z P2  P1 )
   LOOP                      \ end loop
   THEN                      \ end IF. . . THEN clause
   GDROP   GPLUCK   ;        \ clean up stacks
```

4. The items between parentheses, (...), and following a backslash, "\", are comments.

We note the following similarities and differences between the FORTRAN and FORTH versions:

- They are of similar length. The FORTH version contains more explicit steps and looks more cryptic. The FORTRAN version looks more like algebraic formulae.
- The FORTH function lacks an argument list. Functions and subroutines generally look for arguments on stacks[5] built into the system.
- The code uses both primitive words from the FORTH "kernel", as well as advanced concepts from *Scientific FORTH*. In particular, the FORTH version employs generic operations with "run-time binding", so one version works with REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 data types. By contrast, in FORTRAN one needs a separate Legendre function for each type desired.
- FORTH looks more cryptic than FORTRAN because it uses postfix ("reverse Polish") notation, just like a Hewlett-Packard calculator. Thus, while FORTRAN lets us display the algorithm in almost-algebraic form, FORTH's postfix arithmetic conceals the algorithm by decomposing it. This disadvantage can be overcome by suitable commenting, through telegraphic choices of names, or by employing the FORmula TRANslator from Chapter 11.

FORTH 's simple linguistic structure permits almost self-commenting code[6], through clever naming of data structures and operations. In Chapter 2 we shall comment in detail on this and other differences between FORTRAN and FORTH.

Every operation that FORTRAN is capable of can be programmed easily in FORTH. For example, the EXTERNAL specification of FORTRAN has its analogue in "vectoring".

---

5. A stack is a data structure like a pile of cards, each containing a number. New numbers are added by placing them atop the pile, numbers are also deleted from the top. In essence, a stack is a "last-in, first-out" buffer.
6. L. Brodie, *Thinking Forth* (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984). M. Ham, "Structured Programming", *Dr. Dobb's Journal*, July 1986.

But FORTH can not only imitate FORTRAN —using far less memory, compiling and debugging much faster, and often executing faster as well— it can perform tricks that FORTRAN accomplishes barely or not at all. The programming examples sprinkled throughout the book, and concentrated in Chapters 6, 8 and 11 offer repeated concrete proof for these assertions.

My experience with FORTH following 25 or so years in which FORTRAN (and sometimes BASIC) were my staple languages leads me to believe the chief advantage of FORTH over the more common procedural languages is its potential for directness and clarity of algorithmic expression.

One reason FORTH has not yet realized its potential in scientific computing may be that scientists and programmers tend to reside in orthogonal communities, so that no one has until now troubled to publicize the extensions that make FORTH convenient for scientific problem-solving. My sincere hope is that this book will in some measure mitigate this lack.