# Chapter 1

# Toward Scientific FORTH

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 traditional 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 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 FOTRAN and BASIC. This introductory chapter explains some of the differences by contrasting FORTRAN with FORTH.

## 1.1 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

[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 Chatper 2.

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

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.

**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 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 hierarchy in FORTRAN a function an call a subroutine or *vice-versa* and the called sub-program can all still further sub-programs.

**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 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)/3.14159265358969D14 + K
```

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

- Define and reserve space for 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-s;
- Add the second product to the contents Of TEMP;
- Fetch the 3rd element of the (implicit-REAL) array H;

[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.

- 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*.

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 INIEGER). 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} => A(I,J).
```

FORTRAN provides facilities for initializing constants and variables at runtime: 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
```

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

FORTRAN supports modularity through separate compilation of function 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 .EO. 0) RETURN
      CPO = 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 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.

## 1.2 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.

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 )
```

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

```
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
```

We note the 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. FORTRAN version looks more like algebraic formulae.
- 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

[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.

FORTH. For example, the EXTERNAL specification of FORTRAN has its analogue in "vectoring".

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 (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.

# Chapter 2

# Programming in FORTH

This chapter briefly reviews the main ideas of FORTH to let the reader understand the program fragments and subroutines that comprise the meat of this book. We make no pretense to complete coverage of standard FORTH programming methods. **Chapter 2 is not a programmers manual!**

Suppose the reader is stimulated to try FORTH - how can he proceed? Several excellent FORTH texts and references are available: *Starting FORTH* [1] and *Thinking FORTH* [2] by Leo Brodie; and *FORTH: a Text and Reference* [3] by M.Kelly and N.Spies. I strongly recommend reading **FTR** or **SF** (or both) before trying to use the ideas from this book on a FORTH system. (Or at least read one concurrently.)

The (commercial) GEnie information network maintains a session devoted to FORTH under the aegis of the Forth Interest Group (fiG).

FIG publishes a journal *Forth Dimensions* whose object is the exchange of programming ideas and clever tricks.

The Association for Computing Machinery (11 West 42nd St., New York, NY 10036) maintains a Special Interest Group on FORTH (SIGForth).

The Institute for Applied FORTH Research (Rochester, NY) publishes the refereed *Journal of FORTH Application and Research*, that serves as a vehicle for more scholarly and theoretical papers dealing with FORTH.

finally, an attempt to codify and standardize FORTH is underway, so by the time this book appears the first draft of an ANS FORTH and extensions may exist.

---

[1] L. Brodie, *Starting FORTH*, 2nd ed. (Prentice-Hall, NJ, 1986), referred to hereafter as **SF**.

[2] L. Brodie, *Thinking FORTH* (Prentice-Hall, NJ 1984), referred to hereafter as **TF**.

[3] M. Kelly and N. Spies, FORTH: a Tea and Reference (Prentice-Hall, NJ , 1986), referred to hereafter as **FTR**.

## 2.1 The structure of FORTH

THE "atom" of FORTH is a **word** a previously-defined operation (defined in terms of machine code or other, previously-defined words) whose definition is stored in a series of linked lists called the **dictionary**. The FORTH operating system is an endless loop (outer interpreter) that reads the console and interprets the input stream, consulting the dictionary as necessary. If the stream contains a word [4] in the dictionary the interpreter immediately executes that word.

Input? Yes No Interpret ok

Interpret word? Yes EXECUTE number? Yes NUMBER ERROR fig. 2-1 Overview of FORTH outer interpreter

In general, because FORTH is interpretive as well as compiled, the best way to study something new is in front of a computer running FORTH. Therefore we explain with illustrations, expecting the reader to try them out.

In what follows, anything the user types in will be set in `Helvetica`, such as `DECIMAL` below.

Machine responses appear in ordinary type.

We now give a trivial illustration:

```
DECIMAL <cr> ok
```

**Notes**:

- **<cr>** means "the user pushes the ENTER or ⇐ button".
- **ok** is what FORTH says in response to an input line, if nothing has gone wrong.
- `DECIMAL` is an instruction to use base 10 arithmetic. FORTH will use any base on tell it, within reason, but usually only `DECIMAL` and `HEX` (hexadecimal) are predefined.

When the outer interpreter (see fig. 2.1 on p. 13) encounters text with no dictionary entry, it tries to interpret it as a **NUMBER**.

It places the number in a special memory location called "the top of the stack" (TOS)[5]

```
2 17 +. <cr> 19 ok
```

**Notes**:

[4] Successive words in the input stream are separated from each other by blank spaces. ASCII "at, the standard FORTH delimiter.

[5] We will explain about the stack in 2.3.

- FORTH interprets 2 and 17 as numbers, and pushes them onto the stack. "**+**" is a word and so is "**.**" so they are **EXECUTE**d.
- + adds 2 to 17 and leaves 19 on the stack.
- The word . (called "emit") removes 19 from the stack and displays it on the screen.

We might also have said [6]

```
HEX 0A 14 * . <cr> CR ok
```

(Do you understand this? Hint: **HEX** stands for "switch to hexadecimal arithmetic")

If the incoming text can neither be located in the dictionary nor interpreted as a number, FORTH issues an error message.

## 2.2 Extending the dictionary

THE compiler is one of FORTl-Is most endearing features. It is elegant, simple, and mostly written in FORTH. Although the technical details of the FORTH compiler are generally more interesting to systems developers than to scientists, its components can often be used to solve programming problems. When this is the case, we necessarily discuss details of the compiler. In this section we discuss how the compiler extends the dictionary. In 28 below we examine the parts of the compiler in greater detail.

FORTH has special words that allow the creation of new dictionary entries, i.e., new words. The most important are " : " ("start a new definition") and " ; " ("end the new definition").

Consider the phrase

```
: NEW-WORD WORD1 17 WORDZ . . . WORDn ; ok
```

The initial "**:**" is **EXECUTE**d because it is already in the dictionary. Upon execution, "**:**" does the following:

- Creates a new dictionary entry, **NEW-WORD**, and switches from **interpret-** to **compile** mode.
- In compile mode, the interpreter looks up words and rather than executin them installs ointers to their code. If the text is a number (**17** above), FORTH builds the literal number into the dictionary space allotted for **NEW-WORD**.

[6] since FORTH uses words, when we enter an input line we say the corresponding phrase.

- The action of **NEW-WORD** will be to **EXECUTE** sequentially the previously-defined words **WORD1**, **WORD2**, ...**WORDn**, placing any built-in numbers on the stack as they occur.
- The FORTH compiler **EXECUTE**s the last word " ; " of the definition, by installing code (to return control to the next outer level of the interpreter[7]) then switching back from compile to interpret mode. Most other languages treat tokens like "**;**" as ags (in the input stream) that *trigger* actions, rather than actions in their own right FORTH lets components execute themselves.

In FORTH *all* subroutines are words that are invoked when they are named. No explicit CALL or GOSUB statement is required.

The above definition of **NEW-WORD** is extremely structured compared with FORTRAN or BASIC. Its definition is just a series of subroutine calls.

We now illustrate how to define and use a new word using the previously defined words "**:**"and "**;**". Enter the phrase (this new word ***+** expects 3 numbers, a, b, and c on the stack)

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

Notes:

- ***** multiplies b with c, leaving b*c.
- **+** then adds b*c to a, leaving a + b*c behind.

Now we actually try out a + :

```
DECIMAL 5 6 7 *+ . 47 ok
```

**Notes:**

- The period **.** is not a typo, it EMlTs the result.
- FORTHs response to **a b c *+ .** is a + b*c ok.

What if we were to enter *+ with nothing on the stack? Let's try it and see (**.S** is a word that displays the stack without changing its contents):

```
.S empty stack ok

*+ empty stack ok
```

.....

**Exercise:**

Suppose you entered the input line

[7] This level could be either the outer interpreter or a word that invokes **NEW-WORD**.

```
HEX 5 6 7 *+ . <cr> xxx ok
```

What would you expect the response **xxx** to be?

*Answer:* **2F**

. . . . . .

## 2.3 Stacks and reverse Polish notation (RPN)

We now discuss the stack and the "reverse Polish" or "postfix" arithmetic based on it. (Anyone who has used one of the Hewlett-Packard calculators should already be familiar with the basic concepts.)

A Polish mathematician (J .Lukasewcleia) showed that numerical calculations require an irreducible minimum of elementary operations (f*etc*hing and storing numbers as well as addition, subtraction, multiplication and division). The minimum is obtained when the calculation is organized by "stack" arithmetic.

Thus virtually all central processors (CPUs) intended for arithmetic operations are designed around stacks. FORTH makes efficient use of CPU's by reflecting this underlying stack architecture in its syntax, rather than translating algebraic-looking program statements ("infix" notation) into RPN-based machine operations as FORTRAN, BASIC, C and Pascal do.

But what is a stack? As the name implies, a stack is the machine analog of a pile of cards with numbers written on them. Numbers are always added to, and removed from, the top of the pile. (That is, a stack resembles a job where layoffs follow seniority: last in, first out.) Thus, the FORTH input line

```
DECIMAL 2 5 73 -16 ok
```

followed by the line

```
+ - * . yyy ok
```

leaves the stack in the successive states shown in Table 2-1 below

Cell# Initial Ops-¿ + - * . 0 -16 Result 57 -52 104 ... 1 73 -¿ 5 2 ... ... 2 5 2 ... ... 3 2 ... ...

Table 2-1 *Picture of the stack during operations*

We usually employ zero-based relative numbering in FORTH data structures stacks, arrays, tables, *etc*. so TOS ("top of stack") is given relative #0, NOS ("next on stack") #1, *etc*.

The operation "**.**" ("emit") displays -104 to the screen, leaving the stack empty. That is, **yyy** above is **-104**.

## 2.4 Manipulating the parameter stack

FORTH system incorporate (at least) two stacks: the **parameter** stack which we now discuss, and the **return stack** which we defer to 2.3.2.

In order to use a stack-based system, we must be able to put numbers on the stack, remove them, and rearrange their order. FORTH includes standard words for this purpose.

Putting numbers on the stack is easy: one simply types the number (or it appears in the definition of a FORTH word).

To remove a number we have the word **DROP** that drops the number from TOS and moves up all the other numbers.

Tb exchange the top 2 numbers we have .

**DUP** duplicates the TOS into NOS, pushing down all the other numbers.

**ROT** rotates the top 3 numbers.

Cell # initial Ops-¿ DROP SWAP ROT DUP 0 -16 Result 73 73 5 16 1 73 -¿ 5 -16 -16 2 5 2 5 73 3 2 ... 2 2 4 ... ... ... ...

Table 2-2 Stack manipulation operators

These actions are shown on page 19 above in Thble 22 (we show what each word does to the initial stack).

In addition the words **OVER**, **UNDER**, **PICK** and **ROLL** act as shown in Table 2-3 below (note **PICK** and **ROLL** must be preceded by an integer that says where on the stack an element gets **PICK**ed or **ROLL**ed).

Cell # initial Ops-¿ OVER UNDER 4 PICK 4 ROLL 0 -16 Result 73 -16 2 1 73 -¿ -16 73 -16 2 5 73 -16 73 3 2 5 5 5 4 ... 2 2 2 ...

Table 2-3 More stack manipulation operators

Clearly, **1 PICK** is the same as **DUP**, **2 PICK** is a synonym for **OVER**, **2 ROLL** means **SWAP**, and **3 ROLL** means **ROT**.

As Brodie has noted (**TF**), it is rarely advisable to have aword use a stack so deep that **PICK** or **ROLL** is needed. It is generally better to keep word definitions short, using only a small number of arguments on the stack and consuming them to the extent possible. On the other hand, **ROT** and its opposite, **-ROT**[8], are often useful.

[8]defined as `: -ROT ROT ROT ;`

### 2.4.1 The return stack and Its uses

We have remarked above in 22 that compilation establishes links from the calling word to the previously- defined word being invoked. Part of the linkage mechanism during actual execution- is the **return stack** (rstack): the address of the next word to be invoked after the currently executing word is placed on the rstack, so that when the current word is done, the system jumps to the next word. Although it might seem logical to call the address on the rstack the **next** address, it is actually called the **return** address for historical reasons.

In addition to serving as a reservoir of return addresses (since words can be nested, the return addresses need a stack to be put on) the rstack is where the limits of a **DO ... LOOP** construct are placed[9]

The user can also store/retrieve to/from the rstack This is an example of using a component for a purpose other than the one it was designed for. Such use is not encouraged by every FORTH text, needless to say, since it introduces the spice of danger. To store to the rstack we say **>R**, and to retrieve we say **R>**. **DUP>R** is a speedup of the phrase **DUP >R**. The words **D>R DR>** , for maving double-length integers, also exist on many systems. The word **R@** copies the top of the rstack to the TOS.

**The danger is this:** anything put on the rstack during a words execution must be removed before the word terminates. If the **>R** and the **R>** do not balance, then a **wrong next address** will be jumped to and **EXECUTE**d. Since this could be the address of data, and since it is being interpreted as machine instructions, the results will be **always unpredictable**, but seldom amusing.

Why would we want to use the rstack for storage when we have a perfectly good parameter stack to play with? Sometimes it becomes simply impossible to read code that performs complex gymnastics on the parameter stack, even though FORTH permits such gymnastics.

Consider a problem say, drawing a line on a bit- mapped graphics output device from (x,y) to (x',y') that requires 4 arguments. We have to turn on the appropriate pixels in the memory area representing the display, in the ranges from the origin to the end coordinates of the line. Suppose we want to work with x and y first, but they are 3rd and 4th on the stack. So we have to **ROLL** or **PICK** to get them to TOS where they can be worked with conveniently. We probably need them again, so we use

```
4PICK 4PICK ( -- x y x' y' x y)
```

Now 6 arguments are on the stack! See what I mean? A better way stores temporarily the arguments x and y', leaving only 2 on the stack. If we need to duplicate them, we can do it with an already existing word, **DDUP**.

[9] We discuss looping in 2.7 below.

Complex stack manipulations can be avoided by defining **VARIABLE**s named locations to store numbers. Since FORTH, variables are typically *global* any word can access them their use can lead to unfortunate and unexpected interactions among parts of a large program. Variables should be used sparingly.

While FORTH permits us to make variables local to the sub- if routines that use themlo, for many purposes the rstack can advantageously replace local variables:

- The rstack already exists, so it need not be defined anew.
- When the numbers placed on it are removed, the rstack shrinks, thereby reclaiming some memory.

Suppose, in the previous example, we had put x and y on the rstack via the phrase

```
>R >R DDUP .
```

Then we could duplicate and access x and y with no trouble.

10. See FTR, p. 3253 for a description of beheading - a process to make variables local to a small set of subroutines. Another technique is to embed variables within a data structure so they cannot be referenccd inadvertently. Chapters 283-2, 352, 512 and 112 offer examples.

**A note of caution**: since the rstack is a critical component of the execution mechanism, we mess with it at our peril. If we want to use it, we must clean up when we are done, so it is in the same state as when we found it. A word that places a number on the rstack must get it off again using **R>** or **RDROP** before exiting that word[10]. Similarly, since no LOOP uses the rstack also, for each ¿R in such a loop (after DO) there must be a corresponding R ¿ or RDROP (before LOOP is reached). Otherwise the results will be unpredictable and probably will crash the system.

## 2.5 Fetching and storing

ORDINARY (16-bit) numbers are *fetch*ed from memory to the stack by "**@**" ("fetch"), and stored by "**!**" ("store"). The word @ expects an address on the stack and replaces that address by its contents using, *e.g.*, the phrase **X @**. The word "!" expects a number (NOS) and an address (T OS) to store it in, and places the number in the memory location referred to by the address, consuming both arguments in the process, as in the phrase **32 X !**

Double length (32-bit) numbers can similarly be *fetch*ed and stored, by **D@** and **DI** . (FORTH systems designed for the newer 32-bit machines sometimes use a 32-bit-wide stack and may not distinguish between single- and double-length integers.)

Positive numbers smaller than 255 can be placed in single bytes of memory using **C@** and **C!**. This is convenient for operations with strings of ASCII text, for example screen, file and keyboard I/O.

In Chapters 3, 4, S and 7 we shall extend the lexicon of @ and ! words to include floating point and complex numbers.

## 2.6 Arlthmetlc operations

The 1979 or 1983 standards, not to mention the forthcoming ANSII standard, require that a conforming FORTH system contain a certain minimum set of predefined words. These consist of arithmetic operators + */MOD [MOD '/ for (usually) 16-bit signed-integer (-32767 to +32767) arithmetic, and equivalents for unsigned (0 to 65535), double-length and mixed-mode (16- mixed with 32-bit) arithmetic. The list will be found in the glossary accompanying your system, as well as in SF and F111.

## 2.7 Comparing and testlng

n addition to arithmetic, FORTH lets us compare numbers on the stack, using relational operators ¿ ¡ = .These operators work as follows: the phrase 2 3 ¿ ¡ cr ¿ ok

will leave 0 (”false”) on the stack, because 2 (N05) is not greater than 3 (T OS). Conversely, the phrase

23 ¡ ¡cr¿ ok

will leave-1 (”true”) because 2 is less than 3. Relational operators typically consume their arguments and leave a ”ag” to show what happened[11]. Those listed so far work with signed 16-bit integers. The operator **U¡** tests unsigned 16-bit integers (065535).

FORTH offers unary relational operators 0 = 0 ¿ and 0 ¡ that determine whether the TOS contains a (signed) 16-bit integer that is 0, positive or negative. Most FORTHs offer equivalent relational operators for use with double-length integers.

The relational words are used for branching and control. The usual form is

:MAYBE 0¿ IF WORDl WORDZ WORDn THEN ;

12.

[11]The original FORTH-79 used + 1 for ”true”, 0 for ”false”; many newer system that mostly follow FORTH-79 use -1 for ”true”. HS/FORTH is one such. Both FORTH-83 and ANSII FORTH require -1 for ”true”, 0 for ”false”.

The word **MAYBE** expects a number on the stack, and executes the words between **IF** and **THEN** if the number on the stack is positive, but not otherwise. If the number initially on the stack were negative or zero, **MAYBE** would do nothing.

An alternate form including **ELSE** allows two mutually exclusive actions:

:CHOOSE 0¿ IF WORD1 . . . WOHDn ELSE WORDt' . . . WORDn' THEN ; (n - - )

If the number on the stack is positive, **CHOOSE** executes **WORD1 WORD2... WORD**, whereas if the number is negative or 0, **CHOOSE** executes **WORD1' WORDn'** .

In either example, **THEN** marks the end of the branch, rather than having its usual logical meaning[12].

## 2.8 Looping and structured programming

FORTH contains words for setting up loops that can be definite or indefinite:

BEGIN xxx ag UNTIL

The words represented by xxx are executed, leaving the TOS (ag) set to 0 (F) at which point UNTIL leaves the loop or -1 (T) at which point **UNTIL** makes the loop repeat from **BEGIN**.

A variant is

BEGIN xxx ag WHILE yyy REPEAT

Herc xxx is executed, WHILE tests the ag and if it is 0 (F) leaves the loop; whereas if flag is -1 (T) WHILE executes m and

**REPEAT** then branches back to **BEGIN**. These forms can be used to set up loops that repeat until some external event (pressing a key at the keyboard, *e.g.*) sets the ag to exit the loop. They can also used to make endless loops (like the outer interpreter of FORTH) by forcing flag to be 0 in a definition like

:ENDLESS BEGIN )00( 0 UNTIL ;

FORTH also implements indexed loops using the words **DO LOOP +LOOP /LOOP**. These appear within definitions, *e.g.*

: LOOP-EXAMPLE 100 0 DO )OOt LOOP ;

The words xxx will be executed 100 times as the lower limit, 0, increases in unit steps to 99. To step by -2's, we use the phrase

-2 + LOOP

[12]This has led some FORTH gurus to prefer the synonymous word ENDIF as clearer than THEN.

to replace **LOOP**, as in

: DOWN-BY-2's O 100 DO xxx -2 +LOOP ;

The word **/LOOP** 1s a variant of **+LOOP** for working with unsigned limits[13] and increments (to permit the loop index to go up to 65535 1n 16-bit systems).

## 2.9 The pearl of FORTH

An unusual construct, **CREATE. .DOES¿**, has been called the pearl of FORTH" [14] This is more than poetic license.

**CREATE** is a component of the compiler that makes a new dictionary entry with a given name (the next name in the input stream) and has no other function.

**DOES¿** assigns a specific run-time action to a newly **CREATE**d word (we shall see this in 28-3 below).

### 2.9.1 Dummy words

Sometimes we use **CREATE** to make a dummy entry that we can later assign to some action:

CREATE DUMMY CA' * DefinES DUMMY

The second line translates as "The code address of * defines **DUMMY**". Entry of the above phrase would let **DUMMY** perform the job of * just by saying **DUMMY**. That is, FORTH lets us first define a dummy word, and then give it any other words meaning [15].

Here is one use of this power: Suppose we have to define two words that are alike except for some piece in the middle:

```
: *WORD  WORD1 WORD2 *  WORD3 WORD4 ;
: */WORD WORD1 WORD2 */ WORD3 WORD4 ;
```

we could get away with 1 word, together with **DUMMY** fromabove,

```
: *_or_*/WORD
    WORD1 WORD2
    DUMMY
    WORD3 WORD4 ;
```

by saying

[13] Signed 16-bit integers run from -32768 to +32767, unsigted from 0 to 65535. See FTR.
[14] Michael Ham, "Structured Programming", Dr. Dobbslmmal adelm Tools, October, 1986.
[15] This usage is a non-standard construct of HS/FORTH.

```
CA' * DefinES DUMMY *_or_*/WORD
```

or

```
CA' */ DefinES DUMMY *_or_*/WORD .
```

This technique, a rudimentary example of vector-lug, saves memory and saves programming time by letting us vary something in the middle of a definition *after the definition has been entered in the dictionary*. However, this techniqlue must be used with caution as it is akin to self-modifying code[16]

A similar procedure lets a subroutine call itself recursively, an enormous help in coding certain algorithms.

### 2.9.2 Denlng "defining" words

The title of this section is neither a typo nor a stutter: **CREATE** finds its most important use in extending the powerful class of FORTH words called "defining" words. The colon compiler ":" is such a word, as are VARIABLE and CONSTANT. The definition of VARIABLE is simple

```
:VARIABLE CREATE 2 ALLOT ;
```

Here is how we use it:

```
VARIABLE X <cr> ok
```

The inner workings of VARIABLE are these:

- CREATE makes a dictionary entry with the next name in the input stream in this case, X.
- Then the number 2 is placed on the stack, and the word ALLOT increments the pointer that represents the current location in the dictionary by 2 bytes.
- This leaves a 2- e vacancy to store the value of the variable (that is, the next ictionary header begins 2 bytes above the end of the one just defined).

When the outer interpreter loop encounters a new VARIABLE's name in the input stream, that names address is placed on the stack. But this is also the location where the 2 bytes of storage begins. Hence when we type in X, the TOS will contain the storage address named X.

[16]Self-modifying machine code is considered a serious "no-no" by modern structured programming standards. Although it is sometimes valuable, few modern cpu's are capable of handling it safely. More often, because cpu's tend to use pipelining and parallelism to achieve speed, a piece of code might be modified in memory, but having been pre-f*etc*hed before modification actually execute in unmodified form.

As noted in 2.4 above, the phrase X @ (pronounced "X *fetch*") places the contents of address X on the stack, dropping the address in the process. Conversely, to store a value in the named location X, we use ! ("store"): thus

```
4 X ! <cr> ok
X @ . <cr> ok
```

Double-length variables are defined via DVARIABLE, whose definition is

```
2 DVARIABLE CREATE 4 ALLOT ;
```

FORTH has a method for defining words initialized to contain specific values: for example, we might want to define the number 17 to be a word. CREATE and "," ("comma") let us do this as follows:

```
17 CREATE SEVENTEEN , <cr> ok
```

Now test it *via*

```
SEVENTEEN @ . <cr> 17 ok
```

**Note**: The word "," ("comma") puts TOS into the next 2 bytes of the dictionary and increments the dictionary pointer by 2.

A word C, ("see-comma") puts a byte-value into the next byte of the dictionary and increments the pointer by 1 byte.

### 2.9.3 Run-time vs. compile-time actions

In the preceding example, we were able to initialize the variable SEVENTEEN to 17 when we CREATEd it, but we still have to fetch it to the stack via SEVENTEEN @ whenever we want it. This is not quite what we had in mind: we would like to find 17 in T0S when we say SEVENTEEN. The word DOES¿ gives us precisely the tool to do this.

As noted above, the function of DOES¿ is to specify a run-time action for the "child" words of a defining word. Consider the defining word CONSTANT, defined in high-level[17] FORTH by

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

and used as

```
53 CONSTANT PRIME ok
```

Now test it:

```
PRIME . <cr> 53 ok
```

[17] Of course CONSTANT is usually a machinecode primitive, for speed.

What happened?

- CREATE (hidden in CONSTANT) made an entry (named PRIME , the first word in the input stream following CONSTANT). Then ”,” placed the TOS (the number 53) in the next two bytes of the dictionary.
- DOES¿ (inside CONSTANT) then appended the actions of all words between it and ”;” (the end of the definition of CONSTANT) to the child word(s) defined by CONSTANT.
- In this case, the only word between DOES¿ and ; was @ , so all FORTH constants defined by CONSTANT perform the action of placing their address on the stack (anything made by CREATE does this) and fetching the contents of this address.

**Klingons**

Let us make a more complex example. Suppose we had previously defined a word BOX ( n x y - - ) that draws a small square box of n pixels to a side centered at (x, y) on the graphics display. We could use this to indicate the instantaneous location of a moving object - say a Klingon space-ship in a space-war game.

So we define a defining word that creates (not very realistic looking) space ships as squares n pixels on a side:

```
: SPACE-SHIP CREATE . DOES>
@-ROT (--nxy) BOX;
: SIZE ; \ do-nothing word
```

Now, the usage would be (SIZE is included merely as a reminder of what 5 means it has no function other than to make the definition look like an English phrase)

```
SIZE 5 SPACE-SHIP KLINGON <cr> ok
7135 KLINGON <cr> ok
```

Of course, SPACE-SHIP is a poorly constructed defining word because it does not do what it is intended to do. Its child-word KLINGON simply draws itself at (x, y).

What we really want is for KLINGON to undraw itself from its old location, compute its new position according to a set of rules, and then redraw itself at its new position This sequence of operations would require a definition more like

```
:OLD.POS@ (adr--adr n x y) DUP @ OVER
  2+ D@ :
: SPACE-SHIP CREATE , 4 ALLOT DOES>
   OLD.POS@ UNBOX NEW.POSI
```

```
OLD.POS@ BOX DROP ;
```

where the needed specialized operation UNBOX would be defined previously along with BOX.

**Dimensioned data (with Intrinsic unlts)**

Here is another example of the power of defining words and of the distinction between compile-time and run-time behaviors.

Physical problems generally work with quantities that have dimensions, usually expressed as mass (M), length (L) and time (T) or products of powers of these. Sometimes there is more than one system of units in common use to describe the same phenomena.

For example, traffic police reporting accidents in the United States or the United Kingdom might use inches, feet and yards; whereas Continental police would use the metric system. Rather than write different versions of an accident analysis program it is simpler to write one program and make unit conversions part of the grammar. This is easy in FORTH; impossible in FORTRAN, BASIC Pascal or C; and possible, but exceedingly cumbersomein Ada[18].

We simply keep) all internal lengths in millimeters, say, and con- vert as follow[19]:

```
: INCHES  254 10 */ ;
: FEET  [ 254 12 * ] LITERAL 10 */ ;
: YARDS [ 254 36 * ] LITERAL 10 */ ;
: CENTIMETERS 10 * ;
: METERS 1000 * ;
```

The usage would be

```
10 FEET . <cr> 3048 ok
```

These are more definitions than necessary, of course, and the technique generates unnecessary code. A more compact approach uses a *defining word*, UNITS:

```
: D, SWAP , , ; \ I double length # in next ceIIs
: UNITS CREATE D, DOES> D@ */ ;
```

Then we could make the table

[18] An example (and its justification) of dimensioned data types in Ada is given by Do-While l ones, *Dr. Dobb's Journal*, March 1987. The FORTH solution below is much simpler than the Ada version.

[19] This example is based on 16-bit integer arithmetic. The word */ means "multiply the third number on the stack by NOS, keeping 32 bits of precision, and divide by TOS". That is, the stack comment for */is(a b c –a*b/c).

```
254 10        UNITS INCHES
254 12 * 10   UNITS FEET
254 36 * 10   UNITS YARDS
 10  1        UNITS CENTIMETERS
1000 1        UNITS METERS
\ Usage:
\ 10 FEET . <cr> 3048 ok
\ 3  METERS . <cr> 3000 ok
\ ......
\ \textit{etc}.
```

This is an improvement, but FORTH lets us do even better: here is a simple extension that allows conversion back to the input units, for use in output:

```
VARIABLE <AS>                 \ new variable
0 <AS> !                      \ initialize to "F"
: AS -1 <AS> ! ;              \ set <AS> = "T"
: UNITS CREATE D, DOES>
   D@                         \ get 2 #s
   <AS> @                     \ get current val.
      IF SWAP THEN            \ flip if "true"
  */  0 <AS> ! ;              \ convert. reset <AS>

BEHEAD' < AS >       \ make it local for security[20]
\ unit definitions remain the same
\ Usage:
\ 10 FEET       . <cr> 3048 ok
\ 3048 AS FEET  . <cr> 10 ok
```

### 2.9.4 Advanced methods of controlllng the compiler

FORTH includes a technique for switching from compile mode to interpret mode while compiling or interpreting. This is done using the words ] and [ . (Contrary to intuition, ] turns the compiler on, [ turns it off.)

One use of ] and [ is to create an "action table" that allows us to choose which of several actions we would like to perform[21].

For example, suppose we have a series of push-buttons numbered 1-6, and a word WHAT to read them.

That is, WHAT waits for input from a keypad; when button #3 is pushed, *e.g.*, WHAT leaves 3 on the stack.

[20] Headerless words are described in "R, p. 3251f. The word BEHEAD' is HS/FORTHs method for making a normal word into a headerless one. See Ch. 513 for further details.

[21] Better methods will be described in Chapter 5.

We would like to use the word BUTTON in the following way:

```
WHAT BUTTON
```

BUTTON can be defined to choose its action from a table of actions called BUTTONS . We define the words as follows:

```
CREATE BUTTONS ] RING-BELL OPEN-DOOR
  ENTER LAUGH CRY SELF-DESTRUCT [
: BUTTON 1- 2* BUTTONS + @ EXECUTE ;
```

If, as before, I push #3, then the action ENTER will be executed. Presumably button #7 is a good one to avoid[22].

How does this work?

- CREATE BUTTONS makes a dictionary entry BUTTONS.
- ] turns on the compiler: the previously-defined word-names RING-BELL, *etc*. are looked up in the dictionary and compiled into the table (as though we had begun with :), rather than being executed.
- ]returns to interactive mode (as if it were ;), so that the next colon definition (BUTTON) can be processed.
- The table BUTTONS now contains the code-field addresses (CFAs) of the desired actions of BUTTON.
- BUTTON first uses 1- to subtract 1 from the button number left on the stack by WHAT (so we can use 0based numbering into the table if the first button were # 0, this would be unneeded).
- 2* then multiplies by 2 to get the offset (from the beginning of BUTTONS) of the CFA representing the desired action.
- BUTTONS + then adds the base address of BUTTONS to get the absolute address where the desired CPA is stored.
- @ fetches the CFA for EXECUTE to execute.
- EXECUTE executes the word corresponding to the button pushed. Simple!

You may well ask "Why bother with all this indirection, pointers, pointers to pointers, tables of pointers to tables of pointers, and the like?" Why not just have nested IF...ELSE...THEN constructs, as in Pascal?

There are three excellent reasons for using pointers:

[22]The safety of an execution table can be increased by making the first (that is, the zeroth) action WARNING, and making the first step of BUTTON a word CHECK-DATA that maps any number not in the range 1-6 into 0. Then a wrong button number causes a WARNING to be issued and the system resets.

- Nested IF...THENs uickly become cumbersome and difficult to decipher (TF). ey are also **slow** (see Ch. 11).
- Changing pointers is generally much faster than changing other kinds of data for example reading in code overlays to accomplish a similar task.
- The unlimited depth of indirection possible in FORTH permits arbitrary levels of abstraction. This makes the computer behave more "intelligently" than might be possible with more restrictive languages.

A similar facility with pointers gives the C language its abstractive power, and is a major factor in its popularity.

### 2.9.5 Strings

By now it should be apparent that FORTH can do anything any other language can do. One feature we need in any sort of programming scientific or otherwise is the ability to handle alphanumeric strings. We frequently want to print messages to the console, or to put captions on figures, even if we have no interest in major text processing.

While every FORTH system must include words to handle strings(see, e.g., FTIL Ch. 9) the very functioning of the outer interpreter, compiler, *etc*., demands this there is little unanimity in defining extensions. BASIC has particularly good string-handling features, so HS/FORTH and others provide extensions designed to mimic BASICs string functions.

Typical FORTH strings are Iimite to 255 characters because they contain a count in their first byte . The word COUNT

:COUNT DUP 1+ SWAP C@ ; (adf–nadr+1)

expects the address of a counted string, and places the count and the address of the first character of the string on the stack. TYPE, a required 79 or 83 word, prints the string to the console.

It is straightforward to employ words that are part of the system (such as KEY and EXPECT) to define a word like 3" that takes all characters typed at the keyboard up to a final " (close-quote not a word but a string-terminator), makes a counted string of them, and places the string in a buffer beginning at an address returned by PAD".

The word 3. ("string-emit") could then be defined as

```
: $. COUNT TYPE ; (adr - -)
and would be used with 3" like this:

$' The quick brown fox" < cr> ok
S. The quick brown fox ok
```

Since this book in not an attempt to paraphrase fiR it is strongly recommended that the details of using the system words to devise a string lexicon be studied there.

One might contemplate modifying the fiR lexicon by using a full 16-bit cell for the count. This would permit strings of up to 64k bytes (using unsigned integers"), wasting 1 byte of memory per short ( 255 bytes) string. Although few scientific applications need to manipulate such long strings, the program that generated the index-to this book needed to read a page at a time, and thus to handle strings about 35 kbytes long.

24. A single byte can represent positive numbers 0-255. 25. A system variable that returns the current starting address of the "scratchpad".

26. ma. 3.

38 Chapter 2 Programming In FORTH Scientific Forth

10 FORTH programming style A FORTH program typically looks like this

```
\ Example of FORTH program
:WORD1 .. . ;

:WORD2 OTHER-WORDS ;
:WORDS YET-OTHER-WORDS ;

: LAST-WORD wonon ...WORD3
wonoz wonm ;
LAST-WORD <cr> \ run program
```

Note: The word means "disregard the rest of this line". It is a convenient method for commenting code.

In other words, a FORTH program consists of a series of word definitions, culminating in a definition that invokes the whole shebang. This aspect gives FORTH programming a somewhat different avor from programming in more conventional lan-

guages.

Brodie notes in TF that high-level programming languages are considered good if they require structured, top-down program- ming, and wonderful if they impose information hiding. Languages such as FORTRAN, BASIC and assembler that permit direct jumps and do not impose structure, top-down design and data- hiding are considered primitive or bad. To what extent does FORTH follow the norms of good or wonderful programming practice?

1 Structure The philosophy of "structured programming" entered the general consciousness in the early 1970s. The idea was to make the logic of program control flow immediately apparent,

arms-Warm 39

thereby aiding to produce correct and maintainable programs. The language Pascal was invented to impose by fiat the discipline of structure. To ti? end, direct jumps (GOT Os) were omitted from the language .

FORTH programs are automatically structured because word definitions are nothing but subroutine calls. The language con tains no direct jump statements (that is, no GOT 0's) so it is impossible to write "spaghetti" code.

A second aspect of structure that FORTH imposes (or at least encourages) is short definitions. There is little speed penalty incurred in breaking a long procedure into many small ones, unlike more conventional languages. Each of the short words has one entry and one exit point, and does one job. This is the beaux ideal of structured programming!

"2 "Top-down" design

ost authors of "how to program" books recommend design-

ing the entire program from the general to the particular. This is called "top-down" programming, and embodies these steps:

a Make an outline, ow chart, data-ow diagram or whatever, taking a broad overview of the whole problem.

a Break the problem into small pieces (decompose it). 0 Then code the individual components.

The natural programming mode in FORTH is "bottom-up" rather than "top-down" the most general word appears last, whereas the definitions necessarily progress from the primitive to the complex. It is possible - and sometimes vital to invoke a word before it is defined ("forward referencing ). The dic- tionary and threaded compiler mechanisms make this nontrivial.

27. lronially, most programmers refuse to get along without spaghetti code, so commercial Pascal's now include GOTO. Only FORTH among major languages completely eschews both line labels and 6010;, making it the most structured language available.

28. The FORmula TRANsIator in Ch. 1154 uses this method to implement its recursive structure.



CMptar 2 - Programming In FORTH Scientific Forth

The naturalness of bottom-up programming encourages a some- what different approach from more familiar languages:

0 In FORTH, components are specified roughly, and then 3 they are coded they are immediately tested, debugged, redesigned and improved.

0 The evolution of the components guides the evolution of the outer levels of the program.

We will observe this evolutionary style in later chapters as we design actual programs.

3 Information hiding Information (or data) "hiding" is another doctrine of structured programming. It holds that no subroutine should have access to, or be able to alter (corruptlzg data that it does not absolutely require for its own functioning .

Data hiding is used both to prevent unforeseen interactions between pieces of a large program; and to ease designing and debugging a large program. The program is broken into small, manageable chunks ("black boxes") called modules or objects that communicate by sending messages to each other, but are otherwise mutually impenetrable. Information hiding and modularization are now considered so important that special languages Ada, MODULA-Z, C+ + and Object Pascal have been devised with it in mind.

To illustrate the problem information hiding is intended to solve, consider a FORTRAN program that calls a subroutine

Rather like the "cell" system in revolutionary conspiracies, where members of a cell know only each other but not the members of other cells. Mechanisms for receiving and transmitting mes- sages between cells are double-blind. Hence, if an individual is captured or is a spy, he can betray at most his own cell and the damage is limited.

MZ-PWhFOfiTH 1

PROGRAM MAIN some lines

CALL SUB1(arg1, ar92, , argn. mswer) some lines

END

SUB1(X1, , Xn, Y) some lines Y = something RETURN END

There are two ways to pass the arguments from MAIN to SUBI, and FORTRAN can use both methods.

0 Copy the arguments from where they are stored in MAIN into locations in the address space of SUBl (set aside for them during compilation). If the STATEMENTS change the values X1,...,Xn during execution of SUBl, the original values in the callin program will not be affected (because they are stored elsew ere and were copied during the CALL).

0 Let SUBl have the addresses of the arguments where they are stored in MAIN. This method is dangerous because if the arguments are changed during execution of SUBl, they are changed in MAIN and are forever corrupted. If these changes were unintended, they can produce remarkable bugs.

Although copying arguments rather than addresses seems safer, sometimes this is impossible either because the increased memory overhead may be infeasible in problems with large amounts of data, or because the extra overhead of subroutine calls may unacceptably slow execution.

What has this to do with FORTH?

e FORTH uses linked lists of addresses, compiled into a diction- ary to which all words have equal right of access.

0 Since everything in FORTH is a word constants, variables, numerical operations, I/O procedures- it might seem impos- sible to hide information in the sense described above.

0 Fortunate] , word-names can be erased from the dictionary after their F As have been compiled into words that call them. (This erasure is called "beheadmg".)

emvmMmm.

42 Chapter 2 Programmhg in FORTH Scientific Forth

e Erasing the names of variables arantees they can be neither accessed nor corru ted by unaut orized words (except through a calamity so drea ful the program crashes).

4 Documenting and commenting FORTH code FORTH is sometimes accused of being a "write-only" language. In other words, some complain that FORTH is cryptic. I feel this is basically a complaint against poor documentation and unhelpful word names, as Brodie and others have noted.

Unreadability is equally a aw of poorly written FORTRAN, Pascal, C, *et al*.

FORTH offers a programmer who takes the trouble a goodly array of tools for adequately documenting code.

41 Parenthesized remarks The word ( a left parenthesis followed by a space - says "dis- regard all following text up until the next right parenthesis in the input stream. Thus we can intersperse explanatory remarks within colon definitions. This method was used to comment the Legendre polynomial example program in Ch. 1.

42 Stack comments A particular form of parenthesized remark describes the effect of a word on the stack (or on the oating point fstack in Ch. 3). For example, the stack-effect cement (stack comment, for short)

( adr - - n) would be appropriate for the word @ ("*fetc*h"): it says @ expects I

to find an address (adr) on the stack, and to leave its contents (n) . upon completion.

The corresponding comment for i would be

30. The right parenthesis, ) , is not a word but a delimiter.

(nadr–).

An fstack comment is prefaced by a double colon :2 as

(::x–f[x]).

Note that to replace parentheses within the comment we use brackets [ ] , since parentheses would be misinterpreted. Since the brackets appear to the right of the word ( , they cannot be (mis-) interpreted as the FORTH words ] or [ .

With some standard conventions for names", and standard ab- breviations for different types of numbers, the stack comment may be all the documentation needed, especially for a short word.

### Drop line (\)

The word \(back-slash followed by space) has gained favor as a method for including longer comments. It simply means "drop everything in the input stream until the next carriage return". Instructions to the user, clarifications or usage examples are most naturally expressed in an included block of text with each line set off by \.

### Self-documenting code

By eliminating ungrammatical phrases like CALL or GOSUB, FORTH presents the opportunity via telegraphic names33 for words to make code almost as self-documenting and transparent as a simple English or German sentence. Thus, for example, a robot control program could contain a phrase like

2 TIMES LEFT EYE WINK

which is clear (although it sounds like a stage direction for Brunhilde to vamp Siegfried). It would even be possible without much

See, *e.g.*, L Brodie, 11", Ch. 5, Appendix E. Forthosefamiliarwithassemblylanguage, \is exactly analogous to ; in assembler. But since ; is already used to cleseeoiondefinitionsinkaH, the symbol \has been used in itspiace.

The matter of naming brings to mind Mark Twain's remark that the difference between the almost-right word and the right one is the difference between the lightning-bug and the lightning.

difficulty to define the words in the program so that the sequence could be made English-like:

WINK LEFT EYE 2 TIMES .

5 Salim

Some high level languages perform automatic bounds checking on arrays, or automatic type checking, thereby lending them a spurious air of reliability. FORTH has almost no error checking of any sort, especially at run time. Nevertheless FORTH is a remarkably safe language since it fosters fine-grained decomposition into small, simple subroutines. And each subroutine can be checked as soon as it is defined. This combination of simplicity and immediacy can actually produce safer, more predictable code than languages like Ada, that are ostensibly designed for safety.

Nonetheless, error checking especialiy array bounds-checking can be a good idea during debugging. FORTH lets us include checks in an unobtrusive manner, by placing all the safety mechanisms in a word or words that can be "vectored" in or out as desired".

34. See m for a more thorough discussion of vectoring. Brodie, TF, sugests a nice construct called DOER MAKE that can be used for graceful vectoring.