1974, *Astron. Astrophys. Suppl.* **15,** 497-511.

FORTH: A NEW WAY TO PROGRAM A MINI-COMPUTER

C.H. MOORE
National Radio Astronomy Observatory*
Charlottesville, Virginia, U.S.A.

Most programming languages are clumsy or expensive to use on small computers. FORTH is an interactive, high-level language that is neither. It makes extremely efficient use of core and time and makes available facilities usually restricted to low-level languages.

To describe a problem, a programmer defines an application-oriented vocabulary. He composes this vocabulary on disk in source form. A user can compile this into core and use it, as well as the standard facilities of FORTH, to solve his problem with ease and flexibility.

FORTH has been implemented on numerous computers. 8K core memory, a disk (or tape) and a terminal are required. Basic FORTH provides I/0, a macro-assembler and a compiler, resident in only 2K words of core. Another 1K is used for automatic buffers that make disk appear to be an extension of core memory.

1. PHILOSOPHY

The recent development of mini-computers has enhanced their capabilities and decreased their cost to the point where they are displacing larger computers as well as custom hardware. However, a small computer cannot efficiently use a conventional high-level language and is usually programmed in assembler language. This low-level approach requires proficient programmers and considerable effort. Almost universal unhappiness exists as regards the cost, delay and quality of programming. This persists in spite of determined attempts to implement standard techniques effectively.

FORTH is a fresh approach to the problem of effective communication with computers. It is an English-like language whose elements are verbs, nouns and defining words. Verbs cause a sequence of computer operations to occur, nouns are the objects operated on and defining words cause new words to be defined in terms of previously defined ones, or in terms of machine instructions.

FORTH provides a basic vocabulary that a programmer can extend to describe his particular problem. The basic words are those required to compose, rearrange and test a vocabulary simply and conveniently. Thus FORTH provides a single language that covers the range usually requiring assembler language, compiler language, job-control language and an application language.

FORTH is useful at many levels. For the programmer it provides a core-resident assembler and compiler that eliminate loading object programs by re-compiling from source. This capability is usually found only in large computers with very large memories. It provides a text-editor that makes it simple for him to create and modify his program. It also provides a simple way to describe independent (multi-programmed) tasks.

For the user it provides an efficient vocabulary that can handle even demanding real-time applications, and an extensive vocabulary that is particularly valuable in diagnosing puzzling situations. In addition, disk routines that automatically move data blocks from disk to core and back to disk as needed, give him easy access to much more memory than core provides.

*The National Radio Astronomy Observatory is operated by Associated Universities, Inc., under contract with the National Science Foundation.

Finally, although a programmer is required to define the application vocabulary, a user may extend it himself to suit his particular problem. FORTH specifically attempts to avoid conventional jargon and allows stating problems in a simple, natural manner.

2. LANGUAGE

There are 5 key elements of the FORTH system. They are first described briefly, and then their implementation is discussed. But first, since FORTH is basically a vocabulary, it is important to understand what constitutes a 'word' in this vocabulary.

A word is any string of characters bounded by spaces. There are no special characters that cannot be included in a word, or that cannot start a word. Thus characters that represent arithmetic operators, or characters that resemble punctuation, can be words if bounded by spaces. For example, the following are words:

```
FORTH  re-start  +  IF,  ?  */.  2@  3.14  1–0  ′
```

Words will be indicated here by capitals or apostrophies; i.e., the strings FORTH and '?' indicate words.

3. DICTIONARY

The FORTH language is organized into a dictionary that occupies almost all the memory used by the program. Its function is to provide the definition of each word in the vocabulary. This includes the operations to be performed by verbs. The dictionary also contains the most frequently referenced nouns.

FORTH will find words in this dictionary, add words to it and recognize contexts that select limited parts of it.

Words are added to the dictionary by 'defining words' of which three are particularly common. One is ':'. When it is encountered in a message from the terminal, it indicates that the word following it is to be entered into the dictionary. The definition of this new word, in terms of previously defined words, will also be placed in the dictionary. For example,

```
: APPLE  MEDIUM  SIZE  ROUND  RED  FRUIT  ;
```

might be a definition for APPLE.

Another defining word is CODE. It indicates that the word following is to be defined by machine instructions which will also be placed in the dictionary. An example will be given later.

A third defining word is INTEGER which likewise places the following word into the dictionary. However in this case the word is a noun and a location in the dictionary is set aside for its value. For example,

```
100  INTEGER  DATA
```

defines the noun DATA and sets its value to 100.

4. STACK

A push-down stack provides the communication between the user and the words he uses, and between one word and another. He may, for example, place numbers (or other objects) on the stack and then type a word which acts on the stack to produce a desired result. For example, typing

```
12  2400  *  45  /  .
```

places the numbers 12 and 2400 on the stack, multiplies them (*), places 45 on top of the stack, divides the previous result by it (/) and types the result (.).

The recent popularity of push-down stacks in electronic calculators has made this concept a more familiar one.

Another stack, the 'return stack', is used to save certain program parameters.

5. INTERPRETER

FORTH has two interpreters. A high-level interpreter reads a word from the terminal, searches the dictionary for it, and executes the entry it finds.

A low-level interpreter executes a word that was defined by ':' in terms of other words. For example, the definition of APPLE given earlier. When the low level interpreter sees APPLE, it will execute the words MEDIUM, SIZE, ROUND, RED, FRUIT and ';' in sequence. This task is particularly simple, for these words have been compiled. That is, when the word APPLE was defined, the dictionary was searched for each word in its definition and the resulting dictionary address placed into the entry for APPLE. When APPLE is executed, the interpreter need only pick up these addresses and execute the required code. The text that defines APPLE is *not* stored, as is common with interpretative languages.

The low-level interpreter has several important properties. First it is *fast*. Indeed, on some computers it executes only two instructions for each word, in addition to the code implied by the word itself. Second it interprets *compact* definitions. Each word used in a definition is compiled into a single core location. Finally it is machine-independent, for the definition of one word in terms of others does not depend upon the computer that interprets the definition.

As a result, most of the words in a FORTH vocabulary will be defined by ':' and interpreted by the low-level interpreter. The high level interpreter itself, is defined in this way.

6. ASSEMBLER

By using the defining word CODE, the programmer can define words that will cause specified machine instructions to be executed. This type of definition is necessary to perform I/0, implement arithmetic operations, and do other machine-dependent processing.

This is an important feature of FORTH. It permits explicit, computer-dependent code in manageable pieces with specific interfacing conventions. To move an application to a different computer requires re-coding only the CODE words, which will interact with the other words in a computer-independent manner.

Among the words the assembler interprets are instruction mnemonics. Each mnemonic is defined so as to assemble the corresponding machine instruction. For example, the phrase:

```
X LDA,
```

will assemble an LDA instruction with the appropriate address for X and deposit it into the dictionary.

The assembler is extremely compact. Its major cost is the dictionary space occupied by the mnemonics–typically 250 core locations. The push-down stack largely eliminates the symbol table, a large core requirement with conventional assemblers. It does this by eliminating the need to name core locations. Verbs find their parameters on the stack, rather than taking them out of named locations. Likewise, conditional jumps employ the push-down stack at assembly time in a manner described below. Variables and locations, however, may be named, and such names are found in the dictionary as usual.

7. SECONDARY MEMORY

The final key element of FORTH is its blocks–1024 byte sections of secondary memory. Two core buffers are provided into which blocks are read as required. If a block is modified in core, it will be automatically replaced on disk when its buffer must be re-used. For a very small cost, this provides a very great service, for the programmer may presume his data to be in core whenever he wants it.

Blocks are used to store text, the text that defines the vocabulary. These blocks are compiled into core when requested by a user. An editing vocabulary formats a block into 16 lines of 64 characters. It allows the user to modify and re-compile his source code.

Blocks are also used to store data. The programmer can easily combine small records into a block, or spread large records across several blocks. The fact that blocks have the same format, regardless of computer, makes it easy to move an application from one computer to another.

8. EXAMPLE

The FORTH functions described so far are made available through an intrinsic vocabulary of about 100 defined words. This basic dictionary occupies 2K core locations. On a 16 bit computer, secondary memory requires 2 core buffers of 512 locations each. Thus in 3K of core, FORTH provides terminal I/0, disk I/0 in the form of virtual memory, interpreter, number conversion, arithmetic, assembler and compiler.

The vocabulary associated with these is listed in Appendix I. It is extensive enough to make the programmer's job already very simple. To see this, consider a problem:
Suppose that by means of CODE definitions one has defined two words:

READ which reads a 16 bit number from some device and pushes it onto the stack.
SEND which sends a 16 bit number to another device (and removes it from the stack).

The first thing one might do is test them to see that they work:

```
READ .
```

will read a number and type it. To display the number in octal:

```
READ OCTAL .
```

Likewise:

```
100 SEND
```

will send 100 to the output device.

A natural combination of these words might be READ SEND. One might name this combination PASS:

```
: PASS READ SEND ;
```

so that typing PASS will pass a number from one device to the other. To avoid sending negative numbers, one could define:

```
: POSITIVE READ 0 MAX SEND ;
```

which will clip negative numbers at 0.

Once PASS is defined, several numbers may be passed, say 32:

```
: MANY 32 0 DO PASS LOOP ;
```

Typing MANY will execute PASS 32 times. For a variable number of repetitions:

```
: TIMES 0 DO PASS LOOP ;
```

Supplying the loop limit, one has:

```
17 TIMES
```

Now suppose that it is desired to read these numbers and save them on disk. First one defines the block in which the numbers would be saved:

```
100 INTEGER DATA
```

specifies a variable, DATA, that identifies the block and initializes it to 100.
Then one defines a store operator that will use this variable:

```
: STORE DATA @ BLOCK + ! UPDATE ;
```

It expects two numbers on the stack, the sample number (which will be the location in the block) and the value.

DATA @ fetches the block number.
BLOCK produces the core address of the contents of the block, reading it from disk if necessary.
+ adds the sample number to this address.
! stores the value into the data block.
UPDATE advises FORTH that the block must eventually be re-written.

Typing

```
100 3 STORE
```

will store 100 in the fourth (counting from 0) location in the block.

Now one can define

```
: SAMPLE READ SWAP STORE ;
```

and type

```
0 SAMPLE 1 SAMPLE 2 SAMPLE
```

and accomplish pretty much what the words imply. Putting this in a loop:

```
: RECORD 32 0 DO I SAMPLE LOOP ;
```

where 'I' puts the index controlling the loop onto the stack as the parameter for SAMPLE. Every time RECORD is typed 32 numbers are recorded on disk.

Finally, consider establishing a task that will do this every 10 seconds for an hour. Assume the word DELAY has been defined so that

```
1 DELAY
```

will wait 1 millisecond. Then defining

```
: MONITOR 600 0 DO RECORD 1 DATA + ! 10000 DELAY LOOP ;
```

specifies that task. Typing MONITOR begins the hour long task, recording 360 blocks of data.

This example illustrates how FORTH lets one combine simple words in a simple way to form complex operations. This simplicity arises from the convenience of nesting definitions. Note that MONITOR goes through four levels of nesting to get to the basic FORTH words. Although nesting extracts an overhead cost (typically 10–20 μs), when speed is important the innermost loop can always be coded as is usual when coding in a high-level language.

A prudent choice of words can provide definitions that are readable, providing one conceeds the push-down stack convention: that operands must be mentioned, to put them onto the stack, before the operation is given. Note also how easy it is to store data on disk, simply by defining an appropriate operator–with no concern for the actual process of data transfer.

9. PROGRAM

The basic FORTH language can be implemented in many ways. It is presently coded in FORTH, which is as appropriate for this application as for others. The continuing goal is to maximize the capabilities of FORTH while minimizing its size and overhead. As described above, basic FORTH requires about 3K of 16–bit core. On the PDP 11/40 the low-level interpreter overhead is only 4.6 μs per word, actually less than a standard subroutine call (5.9 μs).

The FORTH program has two parts. First is a precompiled object program of about 1K words, including a bootstrap routine to load itself into the computer. This program, as is typical of FORTH, is mostly dictionary. It defines a minimal vocabulary that can define all other words.

The second part is the source description of the basic vocabulary (8 blocks). It is compiled into another 1K words by the object program. As much of the vocabulary is kept in source form as possible, to facilitate changes and additions, and as documentation. This costs nothing, since to re-compile this vocabulary from disk takes only a few seconds. To this basic vocabulary of about 100 words, the vocabulary for the application will be added.

10. DICTIONARY

The dictionary is a list of entires of variable length. An entry can be added to the end of the list, or all the entries after a given one can be forgotten. This provides simple, effective and dynamic core allocation.

When the dictionary must be searched to locate a word, the last entry is examined first. If it doesn't match, a pointer to the previous entry is taken from it and that entry examined. This procedure is repeated until a match is found or the dictionary exhausted. This particular search has the important property that if a word is re-defined, its latest definition will be found first. Moreover, by changing the starting point of the search, and by setting the pointers properly, different parts of the dictionary can be searched and thus several different vocabularies can be made available.

Each dictionary entry has 5 fields. These fields and their implementation on 16 bit computers are shown in figure 1:

10.1. The word being defined requires 2 locations. These 4 bytes contain the character count and first 3 characters of the word. This information can distinguish arbitrarily long words with minimum chance of confusion.

10.2. The pointer used in the dictionary search. This is a 14 bit field containing the address of the previous dictionary entry.

10.3. The precedence assigned the entry. This is a 2 bit field that shares the location containing the search pointer. It contains a code to be described later.

10.4. The address of the code to be executed for the word. The low-level interpreter executes this code whenever it encounters the entry. This field distinguishes different *kinds* of words. For example, all words defined by INTEGER will have the same address, and will cause the same code to be executed.

10.5. Parameters associated with the word. This field may use an arbitrary number of core locations and is responsible for the variable length of the dictionary entries. It contains the definition of verbs, and the space assigned nouns.

11. STACK

FORTH implements a 16 bit push-down stack. Descriptions of such stacks are readily available. Knuth (1968) describes them in *The Art of Computer Programming*, volume 1. When numbers are placed on or taken off the stack, the remaining numbers on the stack are not actually moved. Rather a pointer is adjusted to indicate the last word placed in a static core array.

It is important that the stack extend toward *low* core so that a *positive* address relative to the stack pointer will locate numbers on the stack, and also so that double-word items are placed on the stack in the same relative position that they have in core.

Since the dictionary extends toward *high* core, it is convenient to let the stack and dictionary extend toward each other. Then all core not used by one is available for the other, effectively eliminating any upper limit on the size of the stack.

However there is a lower limit on the stack, since it should never become less than empty. FORTH checks each word *typed* to see that it didn't use more parameters than were actually available on the stack. This provides diagnostics in the error-prone environment of the terminal and permits testing new words. However FORTH doesn't spend time checking words that have been compiled in a definition they are presumed to work correctly.

12. INTERPRETER

FORTH's high-level interpreter is responsible for the appearance of the language. It reads a word from an input string, and thus determines what a word is. The specification of a word is made deliberately simple: a word is a character string bounded by spaces.

A number is a particular kind of word. It may start with a '–' and contain digits, '.' or ':' (to represent time or angle). Numbers without '.' are converted to 16 bit integers and pushed onto the stack:

12345 –0 23:59

Numbers with '.' are converted to 32 bit integers and occupy 2 stack positions, with the high-order part on top:

0. –1.5 3.14159265

The position of the decimal point is saved, but is not used to scale the number. Generally the user is required to type the correct number of decimal places, and the number is treated as an integer with an implied decimal point.

The input string can come from several places. One is the terminal. The terminal input routine will read up to 80 characters into a buffer. It recognizes three control characters:

EOT pass the message to the high-level interpreter.

BACKSPACE discard the last character in the buffer.

CANCEL discard the entire message.

These permit the user to correct typing mistakes.

The message buffer is also used for output. The word TYPE will place a character string into the buffer. The terminal output routine will send it when the buffer is full, or further input is requested. The input string can also come from disk. Typing

3 LOAD

passes the 1024 character string stored in block 3 to the high-level interpreter. In order to let one block load another, the current block number and character pointer are saved on the return stack. The message buffer is identified as block 0. Each block of text must end with the word ';S' which restores the previous block and pointer from the return stack.

FORTH stores all text as 7 bit ASCII characters with even parity. This is an extremely important convention, as it permits effortless compatibility between computers.

The high-level interpreter is defined in the same way as a ':' word and is thus interpreted by the low-level interpreter. It is defined by the phrase

BEGIN WORD FIND NUMBER EXECUTE QUERY END

where BEGIN marks the beginning of an infinite loop.

WORD extracts a word from the input string.

FIND searches the dictionary. If it finds a match for the word, it skips NUMBER. Otherwise

NUMBER attempts to convert it to a binary number. If it fails it types an error message: the word itself, followed by '?'.

Then EXECUTE executes the code for the word (or compiles it).

QUERY waits until more text is available, and

END returns to BEGIN to repeat the procedure.

This interpreter operates in several states. In the lowest state, EXECUTE causes the code specified by the word to be executed. However, the word ':' which defines a new word changes the state so that words will be compiled rather than executed, until the word ';' changes it back. Compiling a word means that the address of the dictionary entry for that word is placed into the entry currently being defined. Similarly, compiling a number means placing the number into the dictionary, preceded by another entry that will place it back on the stack when interpreted.

Figure 2 shows the entry for the word ABS defined by:

: ABS DUP 0 < IF MINUS THEN ;

Its parameter field contains the addresses of the entries for the words that define it. This is analogous to a string of subroutine calls. However, it permits all 16 bits of a location to be used for the address, and eliminates the addressing problem common with 16 bit computers.

The low-level interpreter is *always* interpreting such a definition. It has a pointer (I) that identifies the next word to be interpreted–just as the computer has a register that identified the next instruction to be executed.

The code NEXT:

Puts the address of the entry at I into an index register.

Increments I.

Jumps indirect through the code field of the entry.

The cost of NEXT is a good measure of the match between FORTH and computer. It presents an ideal opportunity for a micro-programmed instruction.

It is important that the address of the *entry* be compiled, not just the address of the code to be executed. Likewise it is important that the entry be placed in an index register. This provides access to the parameter field of the word, as well as the code field.

The actual address assigned an entry depends upon the computer. It is one of the choices available to optimize NEXT. The best choice is the parameter field, but negative relative addressing must be available. Otherwise the code field or the beginning of the entry may be chosen.

13. PRECEDENCE

Since the FORTH compiler is itself written in FORTH, there must be a way to distinguish words being compiled from words directed at the compiler. The precedence field does this. It specifies if a word is to be executed during compilation. Table 1 describes how the precedence of an entry and the value of variable STATE determine when a word is compiled and when it is executed.

Most words have precedence 0 and are compiled normally. The problem arises because some words are difficult to compile, so must be executed while compiling, specifically IF, ELSE, THEN, DO, LOOP and ';'. These words are compiler directives and should not be used outside definitions. Most directives have precedence 1.

However, the nature of FORTH introduces a further complication. In order to be able to *define* compiler directives, it is necessary to define a state in which even compiler directives are compiled. This is done by the directive IMMEDIATE, which sets the precedence of the current entry to 1, and also sets variable STATE to 2. Now directives with precedence 1 will also be compiled (see table 1), but some words must still be executed to stop the compilation process. And so ';' has precedence 2.

14. ASSEMBLER

Each FORTH program has an assembler vocabulary designed for its computer. It uses the manufacturer's mnemonics, with some modifications:

Suffixes are made separate words.

')' is used to indicate indexing (rather than the overworked ','). Since several instructions may share a line of text, commas are appended to the operation mnemonic for readability.

However the greatest change is in the word order. Addresses precede instructions! This is a shock to programmers who have never questioned conventional assembler format. But it is consistent with other FORTH operations, and enormously simplifies the assembler.

The different kinds of instructions are defined by FORTH and in turn used to define the most common mnemonics. Additional mnemonics may be defined whenever needed. They are used mostly after the word CODE which defines an entry that executes the code in its parameter field. For example:

```
CODE + 0S) LDA, 1S) ADD, BINARY JMP,
```

This phrase defines a 16 bit add operation. Its dictionary entry is shown in figure 3. The words act as follows:

CODE + constructs the entry for the word '+'.

O S)	puts onto the stack at assembly time, an address that will reference the number on top of the stack at execute time.
LDA,	assembles a 'load A register' instruction, using the above address from the stack, and places the instruction into the entry for '+'.
1 S)	puts onto the stack the address of the second number on the stack.
ADD,	assembles an 'add to A register' instruction.
BINARY	puts onto the stack the interpreter address appropriate for operations that take two numbers from the stack and leave a single result there.
JMP,	assembles a jump instruction to the address on the stack.

Code ends by returning to one of several points in the interpreter. Each of these has a different effect upon the stack. Of course, the code can also manipulate the stack itself.

Assembler macros are implemented as ':' words since the function of a macro is to define a new instruction in terms of others. For example,

```
: LDB, LDA, IAB, ;
```

might define a 'load B' instruction in terms of a 'load A' instruction and an 'interchange A and B' instruction. Since an instruction mnemonic in FORTH causes that instruction to be assembled, including one in a definition will cause that definition to assemble it as well.

The use of the stack at assembly time also makes possible constructions like:

```
IF ... ELSE ... THEN
```

in the assembler as well as the compiler.

IF	assembles a conditional jump, and leaves the address of the jump instruction on the stack.
ELSE	assembles an unconditional jump, leaves its address on the stack, and inserts the present address in the jump assembled by IF.
THEN	inserts the present address in the jump assembled by ELSE (or IF, if ELSE was omitted).

Similarly:

```
BEGIN ... END
```

will assemble a loop. END assembles a jump to the address left on the stack by BEGIN.

These macros eliminate the need for labels, for they can be nested and even overlapped, by manipulating the stack at assembly time:

```
BEGIN ... IF ... SWAP END THEN ...
```

BEGIN and IF leave addresses on the stack. SWAP exchanges them so that END gets the address from BEGIN, and THEN the address from IF.

15. SECONDARY MEMORY

FORTH's secondary memory technique is basically a form of addressing–relative to the beginning of a block. The block will be brought into core, left there as long as possible and re-written if necessary. Such addressing depends in an important way upon easy access to parameters via the stack. It would be clumsy and expensive to implement in another language.

A FORTH program on a computer with disk has in it the definition

```
: BLOCK CORE BUFFER READ ;
```

which is used to obtain the beginning address, in core, of a disk block:

CORE	checks the core buffers to see if the desired block is there; if so, it skips the next two words.
BUFFER	checks the oldest buffer and writes it back onto disk if it has been changed.
READ	reads the desired block from disk.

Tape can be substituted for disk at some cost in complexity and some delay for tape motion. An initial pass through the tape creates a core map that provides a record number for each block on tape, using a 513th word in each tape record to identify its block number. By knowing where the tape is positioned, FORTH can move it forward or backward to the record for the block being sought. Blocks must be re-written at the *end* of the tape and the map updated to indicate their new positions.

Two core buffers are necessary to allow one block to refer to another without continuous overlaying. More than two buffers are rarely useful. FORTH uses a 513 word buffer with block number in the 513th word, although only 512 words are stored on disk. The word UPDATE sets the sign bit of this status word to indicate that the block has been changed. This permits words to be placed into a block individually, without requiring a disk access for each.

16. STATUS

FORTH has been developed over the past several years. It has recently been chosen for a number of applications in control of instruments, data acquisition and display. Several observatories are using it on IBM 360, Honeywell 316, Varian 620, Data General NOVA, Hewlett Packard 2100, DEC PDP 10 and PDP 11 and Modular MODCOMP II computers.

The success of FORTH in achieving a small system has paid an additional dividend. Basic FORTH contains only about 500 explicitly mentioned machine instructions. To code FORTH for a new computer means replacing these instructions with their functional equivalents. Thus a programmer familiar with FORTH can recode it in several weeks.

The goal of FORTH is to maximize the service it provides the programmer, while minimizing its cost in terms of time and memory use. It seeks to help the programmer provide the user with a system of unprecedented efficiency and versatility.

C.H. Moore

National Radio Astronomy
Observatory
Edgemont Road
Charlottesville, Virginia 22901, USA

Appendix I. Some basic FORTH words. 'l', 'm', and 'n' indicate numbers on the stack; 'a' indicates an address on the stack

Words concerned with the dictionary:

HERE	Address of next available word.
LAST @	Address of last entry.
WHERE	Type name of last entry.
n ,	Compile number into dictionary.
VOCABULARY word	Define the name of a vocabulary.
FORGET word	Forget all entries following 'word'.
n DP +!	Leave n locations for an array.

Words concerned with the stack:

l m n DUP	Leave l m n n on the stack
" OVER	Leave l m n m on the stack
" DROP	Leave l m on the stack
" SWAP	Leave l n m on the stack
" ROT	Leave m n l on the stack
n .	Type (and discard) number.
a ?	Type number at address a.
a COUNT	Fetch the count field of a string.
a n TYPE	Type n characters, starting at address a.

Words concerned with arithmetic:

n CONSTANT word	Define 'word' so that its value (n) is placed onto the stack.
n INTEGER word	Define 'word' so that the address of its parameter field is placed onto the stack.
n a SET word	Define 'word' to store the number into the address.
a @	Fetch the number from address a.
n a !	Store the number into address a.
n a +!	Add the number into address a.
DECIMAL	Specify number base.
OCTAL	Specify number base.
HEX	Specify number base.
n MINUS	Leave −n on the stack
n ABS	Leave |n| on the stack
m n +	Leave m+n on the stack
m n *	Leave m*n on the stack
m n MAX	Leave max (m, n) on the stack
m n MIN	Leave min (m, n) on the stack
m n −	Leave m−n on the stack
m n /	Leave m/n on the stack
m n MOD	Leave m mod n on the stack
l m n */	Leave l*m/n on the stack
n 0=	Leave if n=0 then 1; otherwise 0.
n 0<	Leave n<0 then 1; otherwise 0.
m n <	Leave m<n then 1; otherwise 0.
m n >	Leave m>n then 1; otherwise 0.

Words concerned with the interpreter:

WORD	Read the next word in the input string.
QUESTION	Type an error message–the last word read followed by?
n LOAD	Read text from block n.
;S	End of text.
IMMEDIATE	Set the precedence of the last entry to 1.
: word	Define 'word' and begin compiling its definition.
I	Put the current value of the loop index on the stack.
(the following have precedence 1)	
n IF	Skip to ELSE (or THEN) if number is 0 (false).
ELSE	Skip to THEN.
THEN	Mark end of skip.
m n DO	Begin loop; limit (m) and index (n) are placed on the return stack at execute time.
LOOP	End loop; increment index by 1 and stop at limit.
n +LOOP	End loop; increment index by n and stop at limit.
(the following have precedence 2)	
;	End compilation.
;CODE	End compilation and begin assembling code.
EOT	End of input message from terminal. Await input and continue compilation.

Words concerned with the assembler:

CODE word	Define 'word' to execute the instructions in its parameter field.
NEXT	Execute next word–interpreter return address.
PUT	Replace stack with register return address.
PUSH	Push register onto stack return address.
POP	Discard top of stack return address.
BINARY	'POP' then 'PUT' return address.
S)	Index relative to stack.
E)	Index relative to entry.
X)	Index relative to index register.
I)	Specify indirect address.
R)	Index relative to return stack.
I	Location of next entry for interpreter.
DP	Address of next-available-word in dictionary.
condition IF	Jump if condition false; leave address of jump on stack.
ELSE	Jump; leave address on stack; supply address to IF.
THEN	Supply address to ELSE.
BEGIN	Begin loop; leave address on stack;
condition END	Jump to BEGIN if condition false.

Words concerned with secondary memory:

n BLOCK	Obtain core address of block n.
UPDATE	Mark last block 'changed'.
FLUSH	Re-write changed core buffers onto disk.
ERASE–CORE	Mark buffers empty.

Table 1 Words are compiled if their precedence is less than the value of STATE:

		Precedence		
Situation	STATE	0	1	2
During execution	0	execute	execute	execute
Compiling:	1	compile	execute	execute
after IMMEDIATE	2	compile	compile	execute

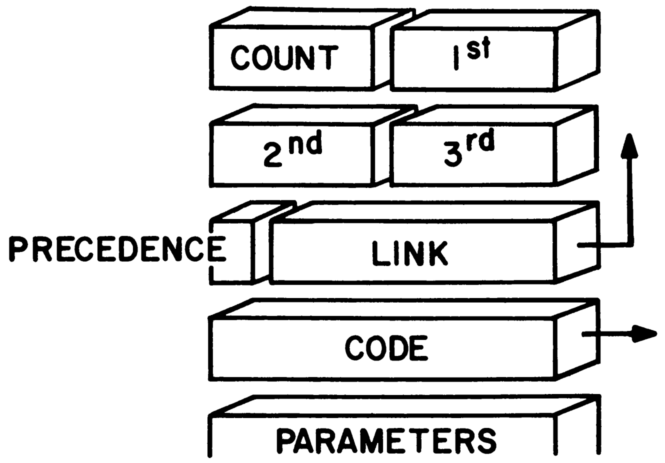

Figure 1 Format of a Dictionary Entry for a 16 bit Computer.

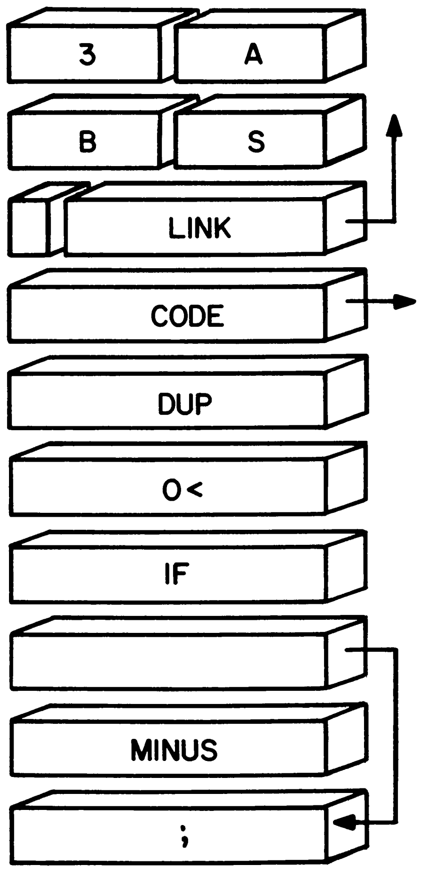

Figure 2 Entry for ';' word ABS. Addresses of entries have been compiled into the parameter field.

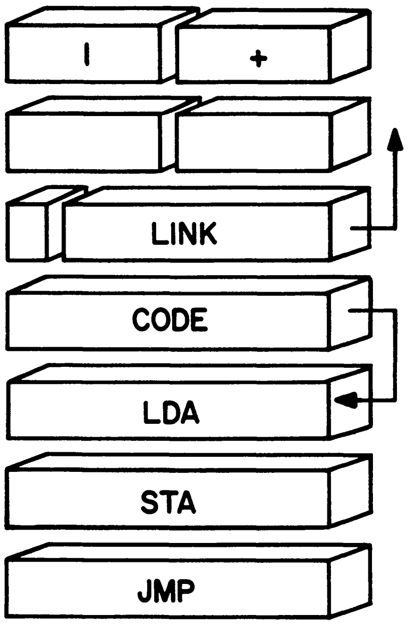

Figure 3 Entry for CODE word '+'. Instructions have been assembled into the parameter field.