Formal Definition of FORTH

Focus on FORTH

by C.H. Ting

C. Ting, Offete Enterprises, Inc., 1306 S. J., San Mateo, CA 94402.

FORTH was developed as a programming tool to solve real-time control problems.[1] While it has never been formally defined as a programming language, I think FORTH is mature enough now that it can be defined rigorously. The wide-spread use of this powerful tool requires that a common base be established to facilitate the exchange of programs and ideas in a standardized language form. The publication of FORTH-79 Standard[2] clearly reflects this necessity. To define FORTH as a programming language also helps us to focus our attention on the basic characteristics of FORTH, and to understand it more fully.

This article will present the definition of FORTH in the Backus Normal Form (BNF) notation. The basic syntax is presented in Table I, in which the focal point is the definition of "word". Some detailed clarifications on colon definitions and defining words are worked out in Tables II and III. Explanatory notes are arranged by section to highlight some problems not clearly expressed in the formal definitions.

Programming Language

A programming language is a set of symbols with rules (syntax) of combining them to specify execution procedures to a computer. A programming language is used primarily to instruct a computer to perform specific functions. However, it can also be used by programmers to document and to communicate problem solving procedures. The most essential ingredients of a programming language are therefore the symbols it employs for expressions and the syntax rules of combining the symbols for man-machine or man-man communications.

FORTH uses the full set of ASCII characters as symbols. Most programming languages use subsets of ASCII characters, including only numerals, upper-case alphabets, and some punctuation characters. Use of punctuation characters differs significantly from language to language. Non-printable characters are generally reserved exclusively for the system and are not available for language usage. In employing the full ASCII set of characters, FORTH thus allows the programmer a much wider range of usable symbols to name objects. On the other hand, the prolific use of punctuation characters in FORTH makes comprehension very difficult by uninitiated programmers.

Only four of the ASCII characters are used by FORTH for special system functions and are not for programming usage: NUL (ASCII 0), RUB (ASCII 127), CR (ASCII 13), and SP (ASCII 32). RUB is used to delete the previously entered character. It is used at the keyboard interactively to correct typing errors. NUL, CR, and SP are delimiting characters to separate groups of characters to form words. All other characters are used to form words and are used the same way. Non-printable characters are treated the same as printable characters. Because non-printable characters are difficult to document and communicate, their usage is discouraged in normal programming practice. However, the non-printable characters are very useful in maintaining a secured system. For terminals that have function keys, their control sequences may be used as the names of the operations each performs.

Words

Words are the basic syntactical units in FORTH. A word is a group of characters separated from other words by delimiting characters. With the exception of NUL, CR, SP and RUB, any ASCII character may be part of a word. Certain words for string processing may specify a regular character as the delimiting character for the word immediately following it, in order to override the delimiting effect of SP. However, the delimiting effect of CR and NUL cannot be overridden.

The usage of "word" in FORTH literature is very confusing because many quite different concepts are associated with it. Without sorting out these different aspects of "word" into independently identifiable entities, it is impossible to arrive at a satisfactory description of this language. Here the word is defined as a syntactical unit in the language, simply a group of characters separated from other words by delimiting characters. Semantically (concerning the meaning of a word), a word in FORTH can be only one of three things: a string, an instruction, or a number.

A FORTH program is thus simply a list of words. When this list of words is given to a computer with a FORTH operating system loaded in, the computer will be able to execute or interpret this list of words and perform functions as specified by this list. The functions may include compilation of new instructions into the system to perform complicated functions not implemented in the original operating system.

A string is merely a group of characters to be processed by the FORTH computer. To be processed correctly, a string must be preceded by an instruction which specifies exactly how this string is to be processed. The string instruction may even specify a regular character as the delimiting character for the following string to override the effect of SP. It is often appropriate to consider the string to be an integral part of the preceding instruction. This would disturb the uniform and simple syntax rule in FORTH and it is better to consider strings as independent objects in the language.

String processing is a major component in the FORTH operating system because FORTH is an interpretive language. Strings are needed to supply names for new instructions, to insert comments into source text for documentation, and to produce messages at run-time to facilitate human interface. The resident FORTH instructions for string processing are all available to programmers for string manipulations.

A number is a string which causes the FORTH computer to push a piece of data onto the data stack. Characters used in a number must belong to a subset of ASCII characters. The total number of characters in this subset is equal to a "base" value specified by the programmer. This subset starts from 0 and goes up to 9. If the base value is larger than 10, the upper-case alphabets are used in their natural sequence. Any reasonable base value can be specified and modified at run-time by the programmer. However, a very large base value causes excessive overlapping between numbers and instructions, and a "reasonable base value" must avoid this conflict in semantical interpretation.

A number can have a leading "-" sign to designate data of negative value. Certain punctuation characters such as "." are also allowed in numbers depending upon the particular FORTH operating system.

The internal representation of numbers inside the FORTH computer depends upon implementation. The most common format is a 16-bit integer num-

ber. Numbers are put on the data stack to be processed. The interpretation of a number depends entirely on the instruction which uses the number. A number may be used to represent a true-or-false flag, a 7-bit ASCII character, an 8-bit byte, a 16-bit signed or unsigned integer, a 16-bit address, etc. Two consecutive numbers may be used as a 32-bit signed or unsigned double integer, or a floating point number.

FORTH is not a typed language in which numerical data types must be declared and checked during compilation. Numbers are loaded on the data stack (hereafter called the "stack") where all numbers are represented and treated identically. Instructions using the numbers on the stack will take whatever they need for processing and push their results back on the stack. It is the responsibility of the programmer to put the correct data on the stack and use the correct instructions to retrieve them. Non-discriminating use of numbers on the stack might seem to be a major source of errors in using FORTH for programming. In practice, the use of the stack greatly eases the debugging process in which individual instructions can be thoroughly exercised to spot any discrepancies in stack manipulations. The most important advantage gained in the uniform usage of data stored on the stack is that the instructions built this way can be context-free, allowing them to be repeatedly called in different environments to perform the same task.

Numbers and strings are objects or nouns in a programming language. Typed and named numbers in a program provide vital clues to the functions and the structures in a program. The explicitly defined objects or nouns make statements in a program easy to comprehend. The implicit use of data objects stored on the data stack makes FORTH programs very tight and efficient. However, statements in a program deprived of nouns are difficult to understand. For this reason, the most important task in documenting a FORTH program is to specify the stack effects of the instructions, indicating what types of data are retrieved from the stack and what types of data are left on the stack upon exit.

Standard Instructions

In a FORTH computer, an instruction is best defined as "a named, linked, memory resident, and executable entity which can be called and executed interactively." The entire linked list of instructions in the computer memory is called a "dictionary." Instructions are known to the programmer by their ASCII names. The names of the instructions in a FORTH computer are words that a programmer can use either to execute the instruction interactively or to build (compile) new instructions to solve his programming problems.

In FORTH literature, instructions are called "words," "definitions," or "word definitions." The reason that I chose to call them "instructions" is to emphasize the fact that an instruction given to the FORTH computer causes immediate actions performed by the computer. The instructions in the dictionary are an instruction set of the FORTH virtual computer, in the same sense as the instruction set of a real CPU. The difference is that the FORTH instructions can be executed directly and the FORTH instructions are accessed by their ASCII names. Therefore, FORTH can be considered as a high-level assembly language with an open instruction set for interactive programming and testing. The name "instruction" conveys more precisely the characteristics of a FORTH instruction than "word" or "definition," and leaves "word" to mean exclusively a syntactical unit in the language definition.

Instruction set is the heart of a computer as well as of a language. In all conventional programming languages, the instruction set is immutable and limited in number and in scope. Programmers can circumvent the shortcomings of a language by writing programs to perform tasks that the native instruction set is not capable of. The instruction set in a FORTH computer provides a basis or a skeleton from which a more sophisticated instruction set can be built and optimized to solve a particular problem.

Because the instruction set in FORTH can be easily extended by the user, it is rather difficult to define precisely the minimum instruction set a FORTH computer ought to have. The general requirement is that the minimum set should provide an environment in which typical programming problems can be solved conveniently. FORTH-79 Standard suggested a convenient instruction set as summarized in Table I. These instructions provided by the operating system are called "standard instructions," and are divided into nucleus instructions, interpreter instructions, compiler instructions, and device instructions.

User Instructions

Instructions created by a user are called "user instructions." There are several classes of user instructions depending upon how they are created. High level instructions are called "colon instructions" because they are generated by the special instruction ":". Low level instructions containing machine codes of the host CPU are called "code instructions" because they are generated by the instruction CODE. Other user instructions include constants, variables, and vocabularies.

Instructions are verbs in FORTH language. They are commands given to the computer for execution. Instructions cause the computer to modify memory cells, to move data from one location to the other. Some instructions modify the size and the contents of the data stack. Implicitly using objects on the data stack eliminates nouns in FORTH programs. It

is not uncommon to have lines of FORTH text without a single noun. The verbs-only FORTH text earns it the reputation of a "write-only" language.

FORTH is an interpretive language. Instructions given to the computer are generally executed immediately by the interpreter, which can be thought of as the operating system in the FORTH computer. This interpreter is called "text interpreter" or "outer interpreter." A word given to the FORTH computer is first parsed out of the input stream, and the text interpreter searches the dictionary for an instruction with the same name as the word given. If an instruction with matching name is found, it is executed by the text interpreter. The text intepreter also performs the tasks of compiling new user instructions into the dictionary. The process of compiling new instructions is very different from interpreting existing instructions. The text interpreter switches its mode of operation from interpretation to compilation by a group of special instruction called "defining instructions," which perform the functions of language compilers in conventional computers.

Syntax of these defining instructions is more complicated than the normal FORTH syntax because of the special conditions required of the compilation of different types of user instructions. The syntax of the defining instructions provided by a standard FORTH operating system is summarized in Table II. The most important defining instruction is the ":" or colon instruction. To define colon instructions satisfactorily, a new entity, "structure," must be introduced. This concept and many other aspects involving defining instructions are discussed in the following subsections.

Structures and Colon Instructions

Words are the basic syntactical units in FORTH language. During run-time execution, each word has only one entry point and one exit point. After a word is processed by the interpreter, control returns to the text interpreter to process the next consecutive word. Compilation allows certain words to be executed repeatedly or to be skipped selectively at run-time. A set of instructions, equivalent to compiler directives in conventional programming languages, are used to build small modules to take care of these optional cases. These modules are called structures.

A structure is a list of words bounded by a pair of special compiler instructions, such as IF-THEN, BEGIN-UNTIL, or DO-LOOP. A structure, similar to an instruction, has only one entry point and one exit point. Within a structure, however, instruction or word sequence can be conditionally skipped or selectively repeated at runtime. Structures do not have names and they cannot be executed outside of the colon instruction in which it is defined. However, a structure can be given a name and defined as a new user instruction. Structures can be nested, but two structures cannot overlap each other. This would violate the one-entry-one-exit rule for a structure.

Structure is an extension of a word. A structure should be considered as an integral entity like a word inside a colon instruction. Words and structures are the building blocks to create new user instructions from low level to high level. All the instructions created at low levels are available to build new instructions. The resulting instruction set then becomes the solution to the programming problem. This programming process contains naturally all the ingredients of the much touted structure programming and software engineering.

Using the definition of structures, the precise definition of a colon instruction is: *"a named, executable entity equivalent to a list of structures."* When a colon instruction is invoked by the interpreter, the list of structures is executed in the order the structures were laid out in the colon instruction.

When a colon instruction is being compiled, words appearing on the list of structures are compiled into the body of the colon instruction as execution addresses. Thus, a colon instruction is similar to a list of subroutine calls in conventional programming languages. However, only the addresses of the called subroutines are needed in the colon instruction because the CALL statement is implicit. Parameters are passed on the data stack and the argument list is eliminated also. Therefore, the memory overhead for a subroutine call is reduced to a bare minimum of two bytes in FORTH. This justifies the claim that equivalent programs written in FORTH are shorter than those written in assembly language.

Compiler instructions setting up the structures are not directly compiled into the body of colon instructions. Instead, they set up various mechanisms such as conditional tests and branch addresses in the compiled codes so that execution sequence can be directed correctly at runtime. The detailed codes that are compiled are implementation dependent.

Code Instructions

Colon instruction allows a user to extend the FORTH system at a high level. Programs developed using only colon instructions are very tight and memory efficient. These programs are also transportable between different host computers because of the buffering of the FORTH virtual computer. Nevertheless, there is an overhead in execution speed in using colon instructions. Colon instructions are often nested for many levels and the interpreter must go through these nested levels to find executable codes which are defined as code instructions. Typically, the nesting and unnesting of colon instructions (calling and returning) cost about 20% to 30% of execution time. If this execution overhead is too much to be tolerated in a time-critical situation, instructions can be coded in machine codes which will then be executed at the full machine speed. Instructions of this type are created by the CODE instruction, which is equivalent to a machine code assembler in conventional computer systems.

Machine code representation depends on the host computer. Each CPU has its own machine instruction set with its particular code format. The only universal machine code representation is by numbers. To define code instructions in a generalized form suitable for any host computer, only two special compiler instructions, "," (comma), and "C," are needed. "C," takes a byte number and compiles it to the body of the code instruction under construction, and "," takes a 16-bit integer from the data stack and compiles it to the body of the code instruction. An assembly code is thus a number followed by "C," or ",". The body of a code instruction is a list of numbers representing a sequence of machine codes. As the code instruction is invoked by the interpreter, this sequence of machine codes will be executed by the host CPU.

Advanced assemblers have been developed for almost all computers commercially available based on this simple syntax. Most assemblers use names of assembly mnemonics to define a set of assembler instructions which facilitates

coding and documenting of the code instructions. The detailed discussion of these advanced instructions is outside the scope of this article.

Constants, Variables, and Vocabulary

The defining instructions CONSTANT and VARIABLE are used to introduce named numbers and named memory addresses to the FORTH system, respectively. After a constant is defined, when the text interpreter encounters its name, the assigned value of this constant is pushed to the data stack. When the interpreter finds the name of a predefined variable, the address of this variable is pushed to the data stack. Actually, the constants defined by CONSTANT and the variables defined by VARIABLE are still verbs in the FORTH language. They instruct the FORTH computer to introduce new data items to the data stack. However, their usage is equivalent to that of numbers, and they are best described as "pseudo-nouns."

Semantically, a constant is equivalent to its preassigned number, and a variable is equivalent to an address in the RAM memory, as shown in Table II.

VOCABULARY creates subgroups of instructions in the dictionary as "vocabularies." When the name of a vocabulary is called, the vocabulary is made the "context vocabulary," which means it is searched first by the interpreter. Normally the dictionary in a FORTH computer is a linearly linked list of instructions. VOCABULARY creates branches to this trunk dictionary so that the user can specify partial searches of the dictionary. Each branch is characterized by the end of the linked list as a link address. To execute an instruction defined by VOCABULARY is to store this link address into a memory location named CONTEXT. Hereafter, the text interpreter will first search the dictionary starting at this link address in CONTEXT when it receives an instruction from the input stream (e.g., the console terminal).

Instructions defined by VOCABULARY are used to switch context in FORTH. If all instructions were given unique names, the text interpeter would be able to locate them without any ambiguity. The problem arises because the user might want to use the same names for different instructions. This problem is especially acute for single character instructions, which are favored for instructions used very often to reduce the typing chore or to reduce the size of source text. The usable ASCII characters are the limit of choices. Instructions of related functions can be grouped into vocabularies using vocabulary instructions. Context will then be switched conveniently from one vocabulary to another. Instructions with identical names can be used unambiguously if they are placed in different vocabularies.

Creating Defining Instructions

FORTH is an interpretive language with a multitude of interpreters. This is the reason why FORTH can afford to have such a simple syntax structure. An instruction is known to a user only by its name. The interpreter which interprets the instruction is specified by the instruction itself, in its code field which points to an executable machine instruction routine. This executable routine is executed at run-time and it interprets the information contained in the body of the instruction. Instructions created by one defining instruction share the same interpreter. The interpreter which interprets high level colon instructions is called "address interpreter," because a colon instruction is equivalent to a list of addresses. Constants and variables also have their respective interpreters.

A defining instruction must perform two different tasks when it is used to define a new user instruction. To create a new instruction, the defining instruction must compile the new instruction

into the dictionary, constructing the name field, link field, and code field – which point to the appropriate interpreter – and the parameter field, which contains pertinent data making up the body of this new instruction. The defining instruction must also contain an interpreter, which will execute the new instruction at runtime. The address of this interpreter is inserted into the code field of all user instructions created by this defining instruction. The defining instruction is a combination of a compiler and an interpreter in conventional programming terminology. A defining instruction constructs new user instructions during compilation and executes the instructions it created at runtime. Because a user instruction uses the code field to point to its interpreter, no explicit syntax rule is necessary for different types of instructions. Each instruction can be called directly by its name. The user does not have to supply any more information except the names, separated by delimiters.

The most exciting feature of FORTH as a programming language is that it not only provides many resident defining instructions as compiler-interpreters, but also supplies the mechanism for the user to define new defining instructions, to generate new classes of instructions or new data structures tailored to specific applications. This unique feature in FORTH amounts to the capability of extending the language by constructing new compilers and new interpreters. Normal programming activity in FORTH is to build new instructions, which is similar to writing programs and program modules in conventional languages. The capability to define new defining instructions is extensibility at a high level in the FORTH language.[4] This unique feature cannot be found in any popular programming language.

There are two methods to define a new defining instruction as shown in Table III. The :–<CREATE–DOES>–; construct creates a defining instruction with an interpreter defined by high level instructions very similar to a structure list in a regular colon definition. The interpreter structure list is put between DOES> and ";". The compilation procedure is contained between <CREATE and DOES>. Since the interpreter will be used to execute all the instructions created by this defining instruction, the interpreter is preferably coded in machine codes to increase execution speed. This is accomplished by the

:–<CREATE–;CODE–

construct. The compilation procedure is specified by instructions between <CREATE and ;CODE. Data following ;CODE are compiled as machine code which will be used as an interpreter when the new instruction defined by this defining instruction is executed at run-time.

Conclusion

Computer programming is a form of art, far from being a discipline of science or engineering. For a specified programming problem, there are essentially an infinite number of solutions, entirely dependent upon the programmer as an artisan. However, we can rate a solution by its correctness, its memory requirement, its execution speed, and other qualities. A solution must be correct. For some applications, the best solution has to be the shortest and fastest. The only way to achieve this goal is to use the computer with an instruction set optimized for the problem. Optimization of the computer hardware is clearly impractical because of the excessive costs. Thus, one would have to compromise by using a fixed, general purpose instruction set offered by a real computer or a language compiler. To solve a problem with a fixed instruction set, one has to write programs to circumvent the shortcomings of the instruction set.

The solution in FORTH is not achieved by writing programs, but by creating a new instruction set in the FORTH virtual computer. The new instruction set in essence becomes "the" solution to the programming problem. This new instruction set can be optimized at various levels for memory space and for execution speed, including hardware optimization. FORTH allows us to surpass the fundamental limitation of any computer, which is the limited and fixed instruction set. This limitation is also shared by conventional programming languages, though at a higher and more abstract level.

FORTH as a programming language allows programmers to be more creative and productive because it enables them to mold a virtual computer with an instruction set best suited for the problems at hand. In this sense, FORTH is a revolutionary development in computer science and technology.

References

1. Moore, C. H., "FORTH: A New Way to Program in Minicomputer," *Astron. Astrophys. Suppl. 15*, pp. 497-511, (1974).

2. "FORTH-79: A Publication of the FORTH Standards Team," FORTH Interest Group, P. O. Box 1105, San Carlos, CA (1980).

3. Main, R. B., "FORTH vs. Assembly," *FORTH Dimensions 1*, 33 (1978).

4. Harris, K., "FORTH Extensibility," *BYTE, 5*, , (1980).

Formal Definition of FORTH as a Programming Language

(Tables I, II, and III)

Table I

```
LANGUAGE DEFINITION OF FORTH

<character> ::= <ASCII code>
<delimiting character> ::= NUL | CR | SP | <designated character>
<delimiter> ::= <delimiting character> |
        <delimiting character><delimiter>

<word> ::= <instruction> | <number> | <string>
<string> ::= <character> | <character><string>
<number> ::= <integer> | -<integer>
<integer> ::= <digit> | <digit><integer>
<digit> ::= 0 | 1 | 2 | ... | 9 | A | B | ... | <base-1>

<instruction> ::= <standard instruction> | <user instruction>

<standard instruction> ::= <nucleus instruction> |
        <interpreter instruction> |
        <compiler instruction> | <device instruction>

<nucleus instruction> ::= ! | * | */ | */MOD | + | +! | - | -DUP | / |
        /MOD | 0< | 0= | 0> | 1+ | 1- | 2+ | 2- | < | = | > | >R | @ |
        ABS | AND | C! | C@ | CMOVE | D+ | D< | DNEGATE | DROP | DUP |
        EXECUTE | EXIT | FILL | MAX | MIN | MOD | MOVE | NEGATE | NOT | OR |
        OVER | R> | R@ | ROT | SWAP | U* | U/MOD | U< | XOR

<interpreter instruction> ::= # | #> | #S | ' | ( | -TRAILING | . | <# |
        >IN | ? | ABORT | BASE | BLK | CONTEXT | CONVERT | COUNT | CURRENT |
        DECIMAL | EXPECT | FIND | FORTH | HERE | HOLD | NUMBER | PAD |
        QUERY | QUIT | SIGN | SPACE | SPACES | TYPE | U. | WORD

<compiler instruction> ::= +LOOP | , | ." | : | ; | ALLOT | BEGIN |
        COMPILE | CONSTANT | CREATE | DEFINITIONS | DO | DOES> | ELSE |
        FORGET | I | IF | IMMEDIATE | J | LEAVE | LITERAL |
        LOOP | REPEAT | THEN | UNTIL | VARIABLE | VOCABULARY | WHILE |
        [ | [COMPILE] | ]

<device instruction> ::= BLOCK | BUFFER | CR | EMIT | EMPTY-BUFFERS |
        FLUSH | KEY | LIST | LOAD | SCR | UPDATE

<user instruction> ::= <colon instruction> | <code instruction> |
        <constant> | <variable> | <vocabulary>
```

Table II

```
USER INSTRUCTIONS

The statement in parenthesis is according to the FORTH syntax.

COLON INSTRUCTION
<colon instruction> ::= <structure list>
   : <colon instruction>  <structure list>  ;      )

<structure list> ::= <structure><delimiter> |
        <structure><delimiter><structure list>
<structure> ::= <word> | <if-else-then> | <begin-until> |
        <begin-while-repeat> | <do-loop>

<if-else-then> ::= IF<delimiter><structure list>THEN
        IF<delimiter><structure list>ELSE<delimiter><structure list>THEN
<begin-until> ::= BEGIN<delimiter><structure list>UNTIL
<begin-while-repeat> ::=
    BEGIN<delimiter><structure list>WHILE<delimiter><structure list>REPEAT

<do-loop structure> ::= <structure> | I | J | LEAVE
<do-loop structure list> ::= <do-loop structure><delimiter> |
        <do-loop structure><delimiter><do-loop structure list>
<do-loop> ::= DO<delimiter><do-loop structure list>LOOP |
        DO<delimiter><do-loop structure list>+LOOP

CODE INSTRUCTION
<code instruction> ::= <assembly code list>
   CODE  <code instruction>  <assembly code list>     )
<assembly code list> ::= <assembly code><delimiter> |
        <assembly code><delimiter><assembly code list>
<assembly code> ::= <number><delimiter>, | <number><delimiter>C,

CONSTANT INSTRUCTION
<constant> ::= <number>
    <number>  CONSTANT  <constant>          )

VARIABLE INSTRUCTION
<variable> ::= <address>
    VARIABLE  <variable>        )
<address>  ::= <integer>

VOCABULARY INSTRUCTION
<context vocabulary> ::= <vocabulary>
    VOCABULARY  <vocabulary>        )
```

(Table III on next page)

Table III

CREATING NEW DEFINING INSTRUCTIONS

```
<high-level defining instruction> ::=
        CREATE<delimiter><compiler structure list>{DOES>}<delimiter>
        <interpreter structure list>;
 : <high-level defining instruction> CREATE <structure list DOES>
        <structure list> ;    )
<low-level defining instruction> ::=
        CREATE<delimiter><compiler structure list>;CODE<delimiter>
        <interpreter assembly code list>
 : <low-level defining instruction> CREATE <structure list> ;CODE
        <interpreter assembly code list>    )
<compiler structure list> ::= <structure list>
<interpreter structure list> ::= <structure list>
<interpreter assembly code list> ::= <assembly code list>
```