CHAPTER 12. COMPILER

The source code discussed here is in the file KERNEL86.BLK, screens 70, and 76 to 78.

12.1. THE COLON DEFINITIONS

Colon definitions are the most prevailing type of word definitions in Forth. A colon definition has a variable-length parameter field where a list of execution addresses is stored. Functionally, a colon definition is the equivalent of the sequence of words whose execution addresses are stored in its parameter field. When the colon definition is invoked, this sequence of words is executed by the address interpreter. Comparing to other high level languages, a Forth colon definition is similar to a procedure or a subroutine, which contains a sequence of procedures or subroutine calls:

```
Forth Colon Definition          FORTRAN Subroutine

: Z                             SUBROUTINE Z
      A                               CALL A
      B                               CALL B
      C                               CALL C
      D                               CALL D
;                               RETURN
```

where A, B, C, and D are other pre-defined words in Forth or subroutines in FORTRAN.

Colon definitions allow us to build higher level functions from existing modules. The building process can continue on until the final colon definition becomes the solution to our programming problem.

What, then, is the advantage of colon definitions over the procedures or subroutines in other languages, since they serve very much the same purposes? The answer is that although a Forth colon definition is the same as a procedure or a subroutine in functionality, it serves the functions more efficiently and it can be debugged more easily. The result is a solution or a program of much higher quality at lower cost. We can summarize the advantages of Forth colon definitions in two words: efficiency and modularity.

EFFICIENCY

Efficiency in computer programming has three aspects: memory utilization, execution speed, and programming productivity. Forth colon definitions excel in all these aspects as compared to other high level languages. Each reference to another pre-compiled definition in Forth costs two bytes in memory, the execution address of the referred definition. The calling of a definition and returning to the caller in Forth is also very fast due to the efficiency in the inner interpreters, especially in hosts of good architecture design. In executing high level language programs, by far the largest overhead is doing subroutine calls and returns, with a host of parameters to be passed between the caller and the callee. Since Forth uses the data stack to pass all the parameters between definitions, the overhead in parameter passing is cut to the minimum. Ease in testing and debugging greatly improve the productivity of programmers using Forth as the software development tool.

MODULARITY

Forth definitions are true modules because they are memory resident and individually executable routines. Once a definition is defined and compiled into the dictionary, it is immediately available for execution and for compiling into other definitions. In other languages, procedures and subroutines are modules only in the abstract sense. They have to be compiled and linked to a mainline program before they can be invoked to do any useful work, within the context of the mainline program. In the example above, the definitions A, B, C, D, and Z are all executable modules in Forth. In FORTRAN, none of the subroutines A, B, C, D, and Z is executable. They are only modules on paper.

Why is true modularity so important? It greatly simplifies the testing and debugging of a program of large size, because individual modules can be thoroughly tested before being integrated into modules at a higher level of construction. In debugging a conventional program, the most valuable tool is the break point facility, allowing the user to stop the program at a selected point to examine the progress of operations. In Forth, each definition can be tested at the interpreter level with a natural break point at its end, eliminating the need of a debugger.

Due to the small overhead in nesting definitions, Forth encourages the breaking of large modules into many small modules which can be tested thoroughly and separately. This modularization gives us a chance to prove the correctness of a large program by proving the correctness of each component and the correctness of their interconnections.

12.2. COLON AND SEMICOLON

So much for propaganda. Let's now look at how the colon compiler itself is defined.

: :	(---)	Define a colon definition. The new definition is hidden until it is completed. The runtime code for : adds a nesting level.
	!CSP	Store the current stack pointer in a variable CSP for error checking at the end of a definition. Normal compilation should not affect the depth of the data stack. If stack depth is changed, it is a potential error condition.
	CURRENT @ CONTEXT !	Select the current vocabulary as the context vocabulary to establish the environment of compilation.
	CREATE	Create a header in the dictionary using the name following:
	HIDE	Smudge the name field of the new header so it is hidden from dictionary searches until compilation is completed.
	]	Enter the colon definition compiler to start constructing the list of execution addresses in the parameter field.
	;USES	Insert the following code routine address into the code field of the new definition. Establish colon inner interpreter.
	NEST ,	Compile the address of the address interpreter NEST here so that it can be put into the code field of new colon definitions.
	;	
: ;	(---)	Terminate a colon definition. It compiles the runtime code of UNNEST to remove a nesting level and changes STATE to

	terminate compilation.
?CSP	Check the current stack pointer with the contents of CSP. If they are not the same, abort.
COMPILE UNNEST	Compile UNNEST at the end of the new colon definition to force execution to return to the caller.
REVEAL	Unsmudge the name field of the new colon definition, making it available for dictionary searches.
[COMPILE] [	Compile [here to terminate the compilation of the new colon definition.
; IMMEDIATE	; must be executed in the compiling state. It must be declared immediate.

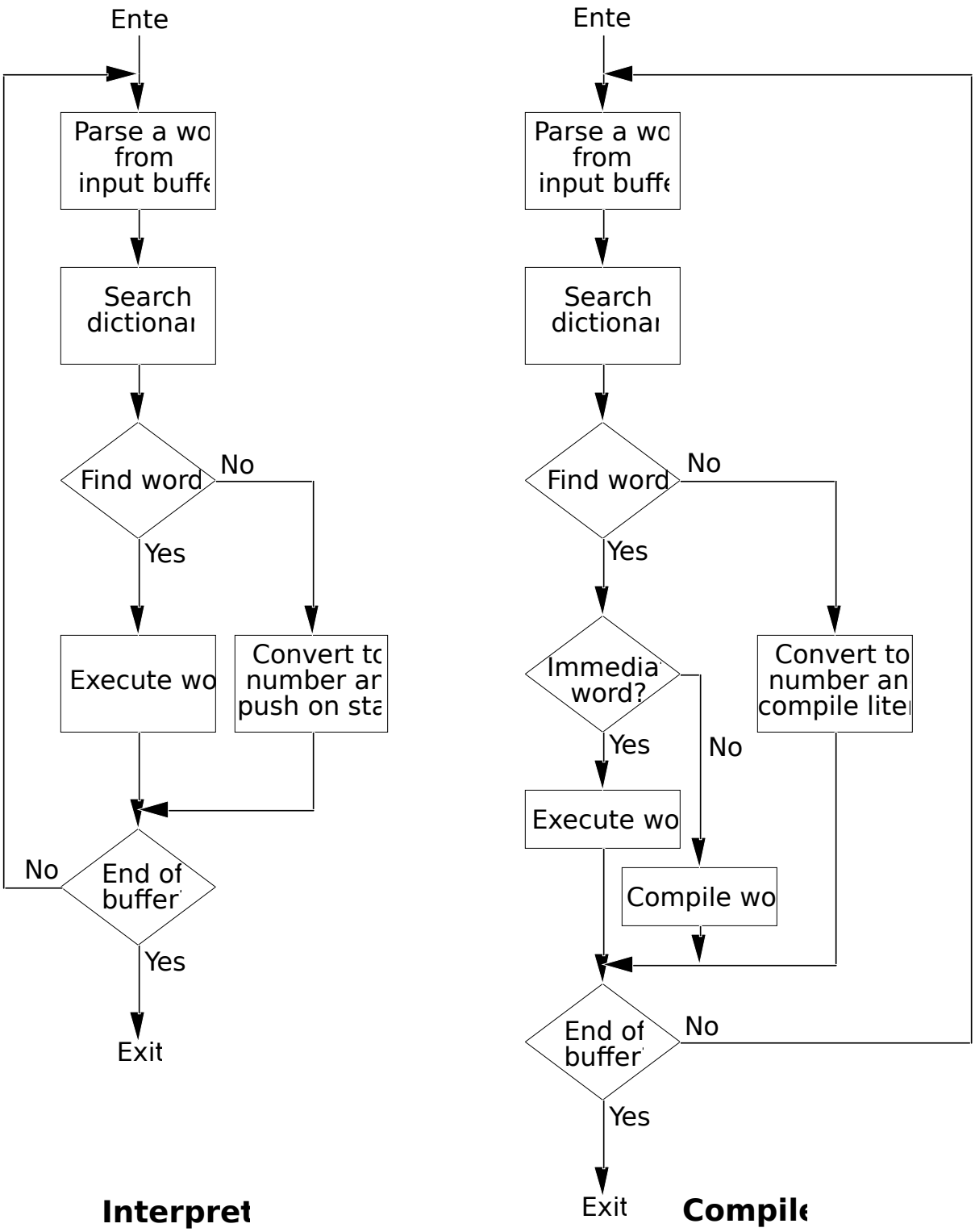
The compilation process is very similar to the interpretation process in Forth. Instead of executing the word parsed out of the input stream, the execution address of the word is added to the top of the dictionary, where we are building the parameter field of a new colon definition. If the word is a number, instead of leaving the number on the data stack, it is compiled into the new definition as a literal so that when the new definition is eventually executed, the same number can be retrieved and put back on the stack. The compiler is embodied in the word], which is the twin brother of INTERPRET, because they share lots of common tasks and structure, as shown in Fig. 12.1. In figForth and many older Forth systems, the compiling functions are actually rolled into INTERPRET as one single piece of definition. The good doctors in the Forth Standard Team decided that it is unsightly that the Siamese twin should share their umbilical cord forever, and cut them loose. The compiling functions are then welded into]. F83 has no choice but to follow the doctor's order.

12.3. THE COMPILER LOOP

```
: ]             ( --- )             The compiling loop.  It sets the compiling flag in STATE,
                                    and parses the next word out of the input stream.  If the
                                    word is found in the dictionary, it is either executed or
                                    compiled depending on whether it is immediate or not.  If it
                                    is a number, it is compiled into the dictionary as either a
                                    single or a double integer literal.  Continue until the input
                                    stream is exhausted. See Fig. 12.1.
    STATE ON                        Set the flag in STATE and enter the compiling mode.
    BEGIN                           Loop to scan the input stream.
        ?STACK                      Check for stack over- or underflow.
        DEFINED                     Parse out the next word and search the dictionary.
            DUP IF                  If it is found in the dictionary,
            0> IF                   and if it is an immediate word,
                    EXECUTE         then execute it.
            ELSE                    If it is not an immediate word, compile its cfa into the dictionary.
            THEN
        ELSE                        It is not a word in the dictionary.
            DROP                    Discard the flag left by DEFINED.
            NUMBER                  Convert the word to a number.
            DOUBLE? IF                  If a punctuation is detected in the string,
            [COMPILE] DLITERAL          compile the double integer literal.
```

```
            ELSE            No punctuation in the string.
                    DROP    Discard the upper half of the converted double integer,
            [COMPILE] LITERAL       and compile the single integer literal.
THEN
        THEN
        TRUE                The compiling flag.
        DONE?               If the input stream is exhausted, leave a true flag to exit the
                            compiling loop.
UNTIL ;                     Otherwise, loop back to compile the next word.
```

Figure 12.1 The interpreter and the compiler



```
: [             ( --- )         Stop compiling and start interpreting.
      STATE OFF                 Turn off the compiling flag in STATE, forcing the Forth
                                system into the interpreting mode.
      ;  IMMEDIATE              It is declared immediate so that its effect can be revealed even
                                during compilation.
```

12.4. LOW LEVEL SUPPORTING INSTRUCTIONS

We have presented the compiler at the highest level. There is a long list of supporting definitions behind these compiler definitions to realize all the functions required in the compilation processes. Let's try to give recognition to all these unsung heroes.

```
VARIABLE DP                     The user variable where the address of the first free memory
                                above the dictionary is stored. It helps the compiler to keep
                                track of its memory.

: HERE          ( --- addr )    Return the address above the dictionary, which is the free
                                memory available for the compiler.
      DP @   ;

: 'WORD         ( --- addr )    Return the address of the word buffer, same as HERE.
      HERE   ;

: ALLOT         ( n --- )       Allocate more space on the dictionary by moving the DP pointer.
      DP +!   ;

: ,             ( n --- )       Copy the top stack item to the top of the dictionary. This is
                                the compiler in its most primitive form.
      HERE !                    Compile n to dictionary.
      2 ALLOT                   Move the DP pointer passing the item just compiled.
      ;

: C,            ( byte --- )    Compile one byte to the dictionary.
      HERE C!                   Compile one byte.
      1 ALLOT                   Move DP.
      ;

: COMPILE   ( --- )             Compile the next word in a colon definition to the dictionary
                                when this definition is executed. It can only be used inside a
                                colon definition.
      R>                        The address of the next word is on the top of the return
                                stack. Retrieve it.
      DUP 2+ >R                 Increment the top of return stack so that the next word will
                                not be executed.
      @                         Get the execution address of the next word.
      ,                         Compile it to the dictionary.
      ;
```

```
: IMMEDIATE         ( --- )     Mark the most recently defined word to make it an immediate
                                word. An immediate word will not be compiled by the ]
                                compiler but will be executed.
      64                        The mask of the precedence bit in the length byte.
      LAST @                    Get the name field address of the most recently defined word
                                from the variable LAST.
      CTOGGLE                   Set the precedence bit in the name field, marking the word
                                immediate.
      ;
```

12.5. IMMEDIATE COMMANDS

Two good examples of immediate words are LITERAL and DLITERAL in]. They are used to compile literal numbers in a colon definition. They are needed because the address interpreter treats the data stored in a colon definition as execution addresses. If we need to put a number on the stack between two addresses, we cannot simply compile the number in-line, because then the number will be interpreted as an address. In-line literal numbers in a colon definition must be preceded by a special runtime word (LIT), which will push the following literal onto the stack when executed. To compile a number into a colon definition, LITERAL is executed immediately to compile first (LIT) and then the number, building the correct literal structure in the colon definition.

```
: LITERAL    ( n --- )      Compile the single integer from the stack as a literal.
      COMPILE (LIT)          First compile the runtime routine (LIT).
      ,                      Then compile the number.
      ;  IMMEDIATE           Make it an immediate word.

: DLITERAL   ( d --- )      Compile the double integer from the stack as a double literal.
      SWAP                   Reverse the order of the double integer so that the right
                             double integer will be pushed onto the stack when executed.
      [COMPILE] LITERAL      Do the literal compilation not now but when DLITERAL is
                             executed. To force the compilation of LITERAL, it must be
                             preceded by [COMPILE], as explained below.
      [COMPILE] LITERAL      Force compilation of the upper half of the double literal.
      ;  IMMEDIATE
```

Words in a colon definition are normally compiled. Immediate words are not compiled but executed immediately. To compile an immediate word like other words in a colon definition, the immediate word must be preceded by [COMPILE]. To compile a word only when the definition is executed, the word is preceded by COMPILE. It is rather confusing for a newcomer to Forth. But, you have to remember, building compiler is not an easy task for everybody. These words encompass activities in the designing of a compiler to build words which will have the correct behavior at runtime. You will have to go to a graduate school of computer sciences to hear these topics discussed only peripherally. To understand these concepts, you have to read more code in the Forth compiler and let it gradually sink in. Or, try to write a few compiler routines yourself and see how they function.

I can offer you one more hint: immediate words are the equivalents of the compiler directives or assembly directives in the conventional programming languages. They may or may not generate

executable code in the program, but they control the process of compilation or assembly and they are executed during the compilation or assembly, but not when the final codes are executed. We will discuss more of these immediate words in the following chapter.

```
: [COMPILE]          ( --- )          Force compilation of the following immediate word.
        ' ( tick )                    Find the execution address of the next word.
        , ( comma )                   Compile it.
        ;   IMMEDIATE                 It must be executed immediately.
```

[COMPILE] cannot wait to let the] compiler find the address it needs, because] might have to execute the next word. [COMPILE] is executed inside the compiler loop and it has to find the address of the next word immediately to compile it. Therefore, [COMPILE] uses a special dictionary searching word ' (tick) to do the dictionary search:

```
: '              ( --- cfa )          Return the execution address of the next word.  If the word
                                      cannot be found, abort.
        DEFINED                       Parse the next word and search the dictionary for it.
        0=                            If the search failed,
        ?MISSING                      Abort with an appropriate message.
        ;                             If the word is found, return its code field address.

: ?MISSING   ( f --- )                Tell the user the word does not exist and abort.
        IF                            The flag is true,
                'WORD COUNT TYPE      Type the word failed to match.
                TRUE ABORT"  ?"       Abort with a very mild message.
        THEN                          Return if the flag is false.
        ;
```