CHAPTER 13. STRUCTURES IN COLON DEFINITIONS

The source code discussed in this chapter is in KERNEL86.BLK, screens 70-71, and 74-75.

13.1. COMPILER DIRECTIVES

We have discussed in great detail the contents and the functions of the colon definition compiler which compiles colon definitions, and the address interpreter which executes the colon definition as a list of execution addresses. If that were all, the usefulness of colon definitions would be severely limited, as they would not be able to cope with the wide variety of situations a programmer must solve using his computer. Very few problems can be solved by linearly strung procedures or words. We need the capability of altering the execution sequence on the fly, depending upon the results obtained in runtime. We need the capability to compile and use different types of data and data structures, which are used to encode input/output information and to hold intermediate information during processing. Compiler directives are used to allow the user to specify explicitly alternate or repetitive execution sequences and compile special data structures inside a colon definition. Compiler directives are called immediate words in Forth because they have to be executed immediately during compilation so that special structures can be built inside a colon definition. Immediate words can be distinguished from normal words by the fact that a bit, the precedence bit, in the first byte of the name field is set.

The compiler loop] can compile normal, non-immediate words and single- or double-integer literals. However, it incorporates an extremely powerful hook to take care of any special compiling conditions in the form of immediate words. Whenever we have a situation that the compiler] is not able to handle, we will design an immediate word to do whatever is necessary to take care of the situation and then let the compiler] continue its normal compilation.

A few examples were shown in the chapter on the colon compiler. In fact, literals are handled also by the colon compiler. When the compiler fails to locate a word in the dictionary, it converts the word into a number and asks LITERAL or DLITERAL, two immediate words, to compile the numbers into the dictionary in the form of two data types, single integer literal or double integer literal. This way, numbers can be compiled into colon definitions, in-line with the execution addresses which are the default data type in colon definitions.

There are other data types and different methods of interpreting them within the context of a colon definition. F83 is very rich in these special words, for the convenience of you the user. Let's look at them closely.

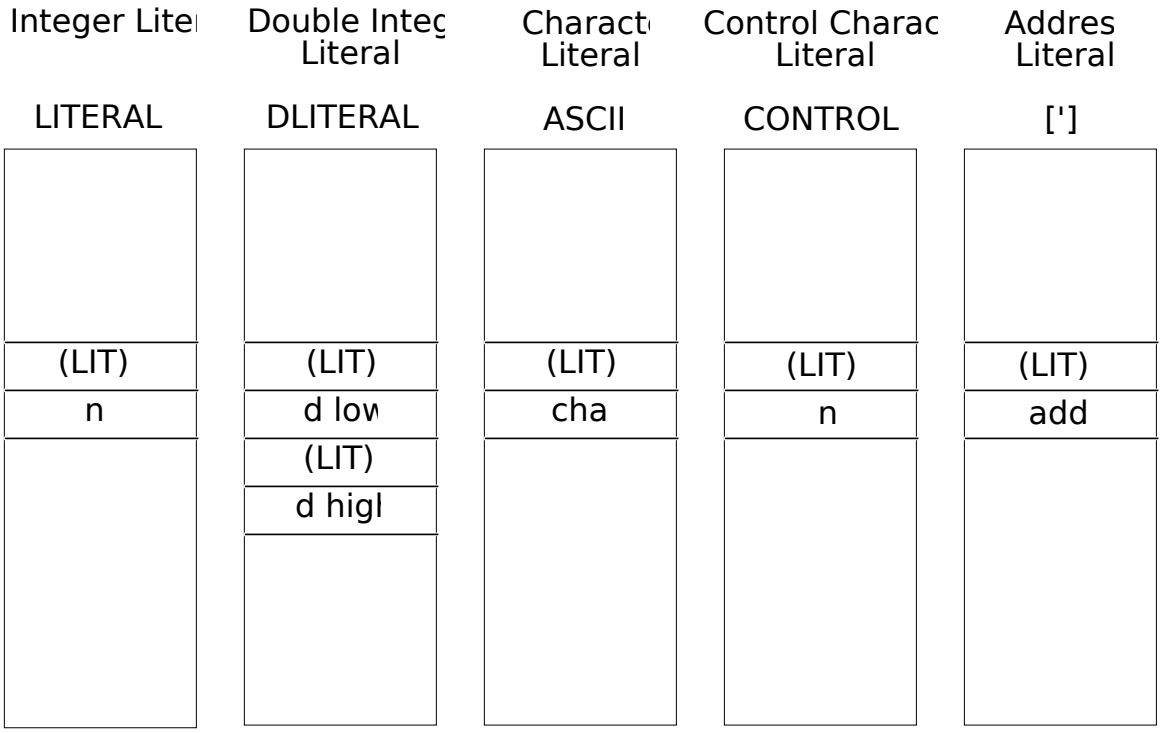
13.2. COMPILING NUMERIC DATA STRUCTURES

Two data types were taken care of: the single integer literal and the double integer literal. The immediate words which compile them are LITERAL and DLITERAL. The runtime word which interprets them, pushing the number on the data stack, is (LIT). The Numeric data structures are shown in Fig. 13.1.

Two immediate words are provided to compile ASCII codes. They also use (LIT) to interpret the compiled character literals:

```
: ASCII         ( --- char )        Compile the next character in the input stream as an ASCII
                                    character literal.
        BL WORD                     Parse out the next character.
        1+ C@                       Get the ASCII code of this character from the word buffer.  STATE       @
                                Are we in the compiling state?
        IF [COMPILE] LITERAL        Yes.  Compile the character as a single integer literal.
                                    However, technically it is a character literal.
        THEN                        If interpreting, just leave the character on stack.
        ;   IMMEDIATE

: CONTROL   ( --- char)             Compile the next character in the input stream as a control
                                    character literal.  The character must be upper case.
        BL WORD                     Get the next character.
        1+ C@                       Get its ASCII code.
        ASCII @                     Offset between the control character and the upper case
                                    character.
        -                           Control ASCII code.
        STATE @                     If compiling,
        IF [COMPILE] LITERAL        compile the control code as a literal.
        THEN                        Leave the character on stack if interpreting.
        ;   IMMEDIATE
```

Figure 13.1 Numeric data structures.


Integer Liter
Double Integ Literal
Charact Literal
Control Charac Literal
Addres Literal
LITERAL
DLITERAL
ASCII
CONTROL
[']
(LIT)
n
(LIT)
d low
(LIT)
d high
(LIT)
cha
(LIT)
n
(LIT)
add

We can always lookup the ASCII table and use the character codes directly in colon definitions. ASCII and CONTROL, however, make very clear documentation to the intention of the programmer. Using these words to invoke ASCII codes explicitly is highly recommended.

Ever heard of address literals? Well, there really are such things. Their usefulness has been demonstrated in many applications in which we want to locate a definition in the dictionary in runtime. An example is to find the address of a colon definition so that we can jump into the middle of it. The reason of doing so is not obvious and certainly it is not orthodox Forth practice. Anyway, if you need the address of another definition inside a colon definition, the word ['] is the one to use, not the plain '.

```
: [']           ( --- )             Compile the address of the next word as a literal.  At
                                    runtime, return that address to the stack.
        ' ( tick )                  Find the execution address of the next word in the input
                                    stream.
        [COMPILE] LITERAL           Compile the address as a literal.
        ;   IMMEDIATE
```

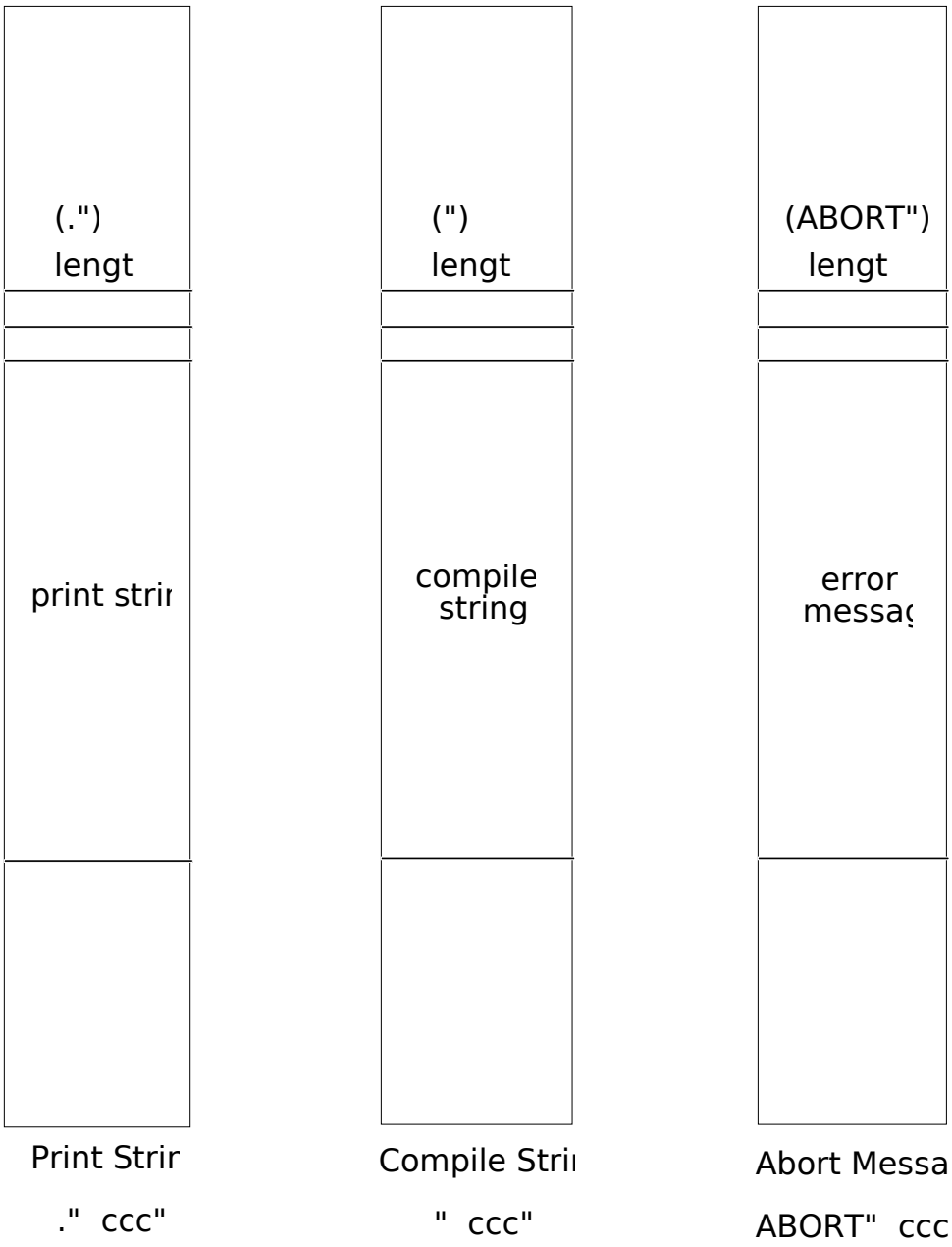
13.3. COMPILING STRING LITERALS

String literals are a very useful data type. They can be used to compile messages in a colon definition. At runtime, the message will be typed out on the console, creating a friendly environment for the end users. String literals are diagrammed in Fig. 13.2.

```
: (")           ( --- addr len )    Return the address and the length of an in-line string.      R>
                                    Address of the in-line string compiled immediately after (").
                                        It is compiled by " and ," .
        COUNT                       Get the addr and len of the string.
        2DUP +                      The address of the executable code after the string.
        EVEN                        Align to cell boundary.
        >R                          Replace it on the return stack to continue the execution
                                    process.
        ;

: (.")          ( --- )             Type out the in-line string and continue executing the word
                                    after the string.  It is compiled by ." .
        R>                          Address of the in-line string.
        COUNT                       Addr and len.
        2DUP + EVEN >R              Replace the address of the next word to be executed.        TYPE
                                    Output the string to console.
        ;

: ,"            ( --- )             Compile the following string to the dictionary.
        ASCII "                     Use " as the delimiter of the string.
        PARSE                       Parse the string out.
        TUCK 'WORD PLACE            Copy the string into the word buffer, just the right place to
                                    compile this string.
        1+ ALLOT ALIGN              All we have to do is to move the DP pointer to include the
```

string in the dictionary.
;

Figure 13.2 The string literals.



```
: ."          ( --- )          Compile the following string to be typed out later.  COMPILE (.")
              Compile the runtime code (.") before the string so that the
        string will be interpreted correctly.
        ,"                     Compile the string into the dictionary.
        ;  IMMEDIATE           This is a compiler directive. Declare it to be immediate.

: "           ( --- )          Compile the string.  At runtime, return its address and
                               length.
        COMPILE (")            Compile the runtime routine (").
        ,"                     Compile the string after (").
        ; IMMEDIATE            Must be immediate.
```

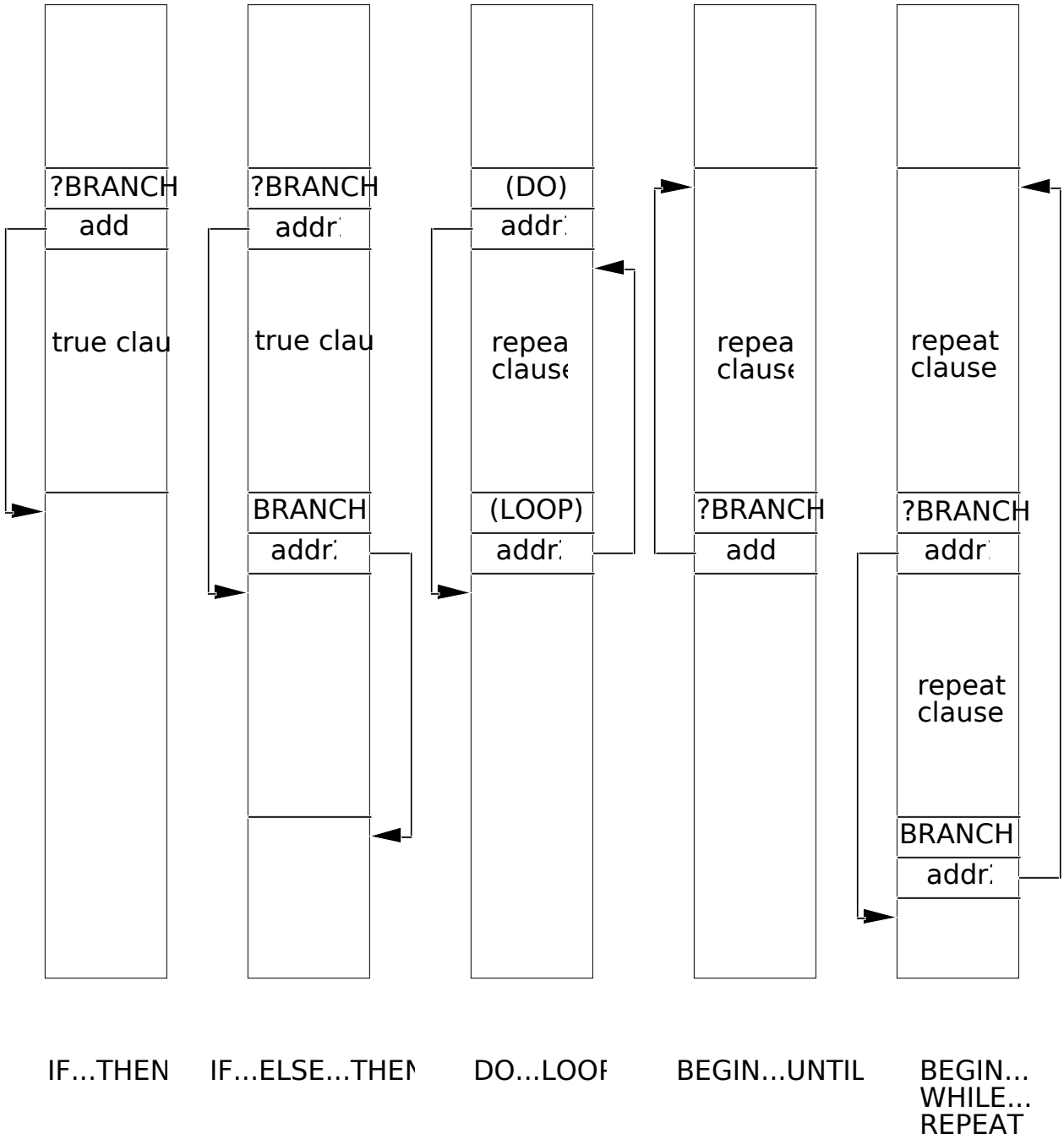
An important word also using string literals is the word ABORT". It forces the Forth system to return to the text interpreter with a clean state to start over again. It can also print out a message explaining why it has to take such a drastic measure to help you figure out what happened in the computer.

```
: (ABORT")   ( f --- ) The runtime routine compiled by ABORT".
        R@ COUNT               Get the addr and len of the following string literal.
        ROT                    Move the flg to the top of stack.
        ?ERROR                 Turn over to ?ERROR to process the error condition.      R>
COUNT + EVEN >R        Move the top of return stack to the word after the string, to
                       resume execution as the error condition was not true.      ;

: ABORT"     ( f --- )         If the flag is true, issue an error message and quit.  COMPILE
(ABORT")     Compile runtime routine.
        ."                     Compile the message.
        ;  IMMEDIATE

DEFER ?ERROR                   Vectored to (?ERROR).

: (?ERROR)  ( addr len f --- )      If the flag is true, execute WHERE to store useful debugging
                                    data, type a message, and quit.
        IF                          If the flag is true, prepare to quit.
                >R >R          Save the string parameters.
                SP0 @ SP!           Initialize the data stack.
                PRINTING OFF        Turn off the printer.
                BLK @ IF            If BLK is not zero, we are processing data from a disk
                               block.
                        >IN @ BLK @ WHERE      Save the character pointer to the input buffer
                                    and the block number and call WHERE to show where
                               error occurred.
                THEN
                R> R>          Restore the string parameters.
                SPACE TYPE SPACE    Print the abort message.
                QUIT                Restart the text interpreter.
        ELSE                        No error condition.
                2DROP               Clear the data stack.
        THEN ;
```

DEFER WHERE — WHERE is vectored to an editor routine (WHERE) to display the block of source with the cursor pointing to the word that causes the abort.

There are other data structures that can be compiled into the colon definitions. However, many of them can be taken care of by variables and arrays derived from variables. Other recurring structures may be handled by the CREATE---DOES> technique.

Figure 13.3 The control structures



13.4. COMPILING CONTROL STRUCTURES

Forth is a structured language. A structured language has provisions for the user to do two things: successive refinement to decompose a problem into smaller parts hierarchically, and building modules with control structures. Control structures, or simply structures, are segments of a program or groups of program statements which have only one entry and one exit. The one- entry-one-exit property of control structures allows the structures to be stacked linearly to form larger segments which can be built into other structures at a higher level. Execution can take alternate paths or repeat a portion of the path only within a structure. Very complicated high level structures can be built on simple structures, enabling programmers to deal with real life problems efficiently.

In a previous chapter, I emphasized that Forth is a truely modular language because the definitions in Forth are true modules, which can be independently executed and compiled, quite different from modules in other languages which can function only within the context of a mainline program. Forth definitions are also structures, with one entry and one exit. There are some exceptions when error conditions are encountered. In these cases, execution is forced to abort to the text interpreter. Forth definitions, as structures, can be stacked linearly together to form higher level structures, which are basically the colon definitions. Besides linearly stacked structures, Forth provides a special set of words which allows us to build other more sophisticated control structures inside colon definitions so that alternate paths can be chosen and segments can be repeated in runtime. These structure building words are all immediate words, because they have to perform extra work to build the desired structures correctly. The control structures are shown in Fig. 13.3. The set of structure-building words in F83 is listed here showing the syntax of their usages:

IF <true clause> THEN
IF <true clause> ELSE <false clause> THEN
BEGIN <repeat clause> UNTIL
BEGIN <repeat clause> AGAIN
BEGIN <repeat clause 1> WHILE <repeat clause 2> REPEAT
DO <repeat clause> LOOP
DO <repeat clause> +LOOP
?DO <repeat clause> LOOP
?DO <repeat clause> +LOOP

Inside the do-loops, the optional words LEAVE and ?LEAVE can be used to force the termination of the loop.

13.5. ADDRESS CALCULATION FOR CONTROL STRUCTURES

In Sections 4.3 and 4.5 we have already discussed the low level words which change the execution sequence in runtime. What the structure building words have to do is to compile these runtime routines into the colon definition with additional branching addresses so that the execution sequence in runtime can be changed according to pre-defined rules. Thus a group of supporting words is needed to calculate the branching addresses during compilation.

```
: ?CONDITION        ( f --- )       Compile time error checking. If the flag is false, abort.
        NOT                         Invert the flag.
        ABORT" Conditionals Wrong"  Abort with a message.
        ;                           This simple error checking is adequate for most situations.

: >MARK             ( --- addr )    Mark the point of a forward branch by saving its address on
                                    the stack.
```

```
        HERE                    Addr in which the forward branching address will be
                                placed.
        0 ,                     Compile a dummy address for the moment.
        ;

: >RESOLVE      ( addr --- )    Resolve a forward branch, by compiling addr at HERE.
        HERE                    This is the address to jump to.
        SWAP !                  Store this address in the memory addr where the forward
                                jump originates.
        ;

: <MARK         ( --- addr )    Set up a backward branch by leaving the current address on
                                stack.
        HERE                    This is the address the backward branch will jump to.   ;

: <RESOLVE ( addr --- )         Resolve a backward branch by compiling addr.
        ,       ;               Compile the backward jump address at this point.

: ?>MARK        ( --- f addr )  Set up a forward branch with error checking.
        TRUE                    Put up a true flag for error checking.
        >MARK   ;               Do the work.

: ?>RESOLVE     ( f addr --- )  Resolve an forward branch with error checking.
        SWAP ?CONDITION Check conditional error first.
        >RESOLVE   ;                    Then resolve the forward branch.

: ?<MARK        ( --- f addr )  Set up a backward branch with error checking.
        TRUE                    The flag for error checking.
        <MARK   ;               Backward jump address.

: ?<RESOLVE     ( f addr --- )  Resolve a backward branch with error checking.
        SWAP ?CONDITION Error checking.
        <RESOLVE    ;           Resolve the backward branching.
```

Error checking is a valuable service to the user to make sure that he has laid down the control structures correctly. Structure words not properly paired are frequent causes of system crashes, because execution can be steered to an unknown address.

13.6. CONTROL STRUCTURE COMPILER DIRECTIVES

Here come the real heroes that compile the control structures in colon definitions. These structure words look very simple and indeed they are. All they have to do is to pick and compile the right runtime routine and resolve the branching addresses. The runtime routines know what to do with the branching addresses and change the execution sequence if necessary. These branching addresses can be considered as special address literals, different from the normal execution addresses compiled by the] compiler.

```
: IF                    ( --- f addr )   Set up the IF-ELSE-THEN structure.
      COMPILE ?BRANCH                    Conditional branch.
      ?>MARK                             Set up forward branch.
      ; IMMEDIATE
```

```
: ELSE          ( f1 addr1 --- f2 addr2 )       Resolve the forward branch from IF and set up
                                                forward branch to THEN.
        COMPILE BRANCH Unconditional branch.
        ?>MARK                          Set up flag and address to jump to THEN.
        2SWAP ?>RESOLVE Resolve the jump address at IF.
        ;  IMMEDIATE

: THEN                  ( f addr --- )  Resolve the forward jump from either IF or ELSE.  ?>RESOLVE
                        Resolve the jump address.
        ;  IMMEDIATE


: BEGIN                 ( --- f addr )  Mark the address for backward branching.
        ?<MARK          ;  IMMEDIATE


: UNTIL                 ( f addr --- )  Compile a conditional branch to BEGIN.
        COMPILE ?BRANCH         Compile the conditional branch runtime routine here.        ?
<RESOLVE                        Put the address of BEGIN here to close the loop.
        ;  IMMEDIATE

: AGAIN                 ( f addr --- )  Compile an unconditional branch to BEGIN.COMPILE  BRANCH
        Unconditional branch.
        ?<RESOLVE                       Address of BEGIN.
        ;  IMMEDIATE

: WHILE                 ( --- f addr )  Compile  a  conditional  exit  in  the  BEGIN-WHILE-REPEAT
                                        loop.
        [COMPILE] IF                    Functionally,  WHILE  is  identical  to  IF.  To  execute  IF  when
                                        WHILE is called, you have to use [COMPILE] to override
                                the immediate effect of IF.
        ;

: REPEAT        ( f1 addr1 f2 addr2 --- )       Compile an unconditional branch to addr1 left by
                                        BEGIN, and resolve the forward branch for WHILE at
                        addr2.
        2SWAP                           Get f1 and addr1 to top of stack.
        [COMPILE] AGAIN                 Use AGAIN to compile the unconditional branch back to
                                BEGIN.
        [COMPILE] THEN                  Since WHILE is identical to IF, we can use THEN to
                                resolve its forward branch.
        ;  IMMEDIATE
```

```
: DO                ( f addr --- )    Compile the header of a do-loop.
        COMPILE (DO)                  Put the runtime (DO) here.
        ?>MARK                        (DO) needs the address after LOOP, making it look like a
                                      forward branching for a real backward branching.
        ;   IMMEDIATE

:  ?DO              ( f addr --- )    Compile the header for ?DO-LOOP.
        COMPILE (?DO)
        ?>MARK      ;   IMMEDIATE
```

```
: LOOP              ( f addr --- )   Complete the do-loop.
        COMPILE (LOOP)               Compile the runtime routine here.
        2DUP 2+ ?<RESOLVE            The backward branch address is 2 bytes after (DO), because
                                     (DO) needs two bytes to store the address after (LOOP), in
                                 case LEAVE needs it.
        ?>RESOLVE                    Store address after (LOOP) to the location just after (DO).   ;
IMMEDIATE

: +LOOP             ( f addr --- )   Compile the ending of the +loop.
        COMPILE (+LOOP)
        2DUP 2+ ?<RESOLVE
        ?>RESOLVE  ;   IMMEDIATE

: LEAVE             ( --- )          Compile (LEAVE).
        COMPILE (LEAVE)  ;   IMMEDIATE

: ?LEAVE            ( --- )          Compile conditional leave.
        COMPILE  (?LEAVE)        ;   IMMEDIATE
```

As it is evident in the definitions of these control structure words, these words must used in pairs, and they can be considered as the delimiters for structures in the colon definition, clearly indicating the entry points and the exit points of the structures. IF must be followed by THEN. DO must be paired with either LOOP or +LOOP. BEGIN must be paired with UNTIL, AGAIN, or REPEAT. Structures can be nested but can not be overlapped. If the structures are overlapping, the system will behave erratically if not crashed.

The error checking in compiling the structures in F83 is not as extensive as that in the fig-Forth model, in which different types of structures are assigned different error checking numbers instead of a true-false flag. Fig-Forth prohibits the compiling of improperly nested structures. Nevertheless, F83 is better than those earlier Forth system without any error checking on the control structures. If you want speed in compilation, you can strip out the error checking in F83 by using >MARK in place of ?>MARK, etc., and change all the 2DUP to DUP. Then you are entirely on your own.