Interpreting Control Structures – The Right Way

Mitch Bradley
Bradley Forthware

Abstract

A very simple modification allows the Forth interpreter to execute conditionals and loops in interpret state as well as in compile state. Interpreted loops run at the same speed as compiled loops.

How It Works

The Forth 83 Standard says that control structures (conditionals and loops) are compiled inside colon definitions. There is a easy way to remove this restriction so that control structures work just as well from interpret state.

The idea is very simple: when a control structure is encountered while interpreting, switch to compile state and begin compiling an unnamed temporary colon definition. When that control structure is finished, execute the unnamed colon definition and then forget it.

Nested control structures can be easily handled. Each word which begins a control structure increments a variable, and each word which ends a control structure decrements the variable. When that variable changes from 0 to 1, begin compiling the unnamed colon definition. When the variable changes from 1 to 0, execute the unnamed colon definition and forget it.

This can easily be implemented with 4 words:

SAVED-DP (-- adr)
: The address of a variable which contains the starting address of the temporary colon definition, if one is being compiled.

LEVEL (-- adr)
: The address of a variable which contains the current control structure nesting level.

+LEVEL (--)
: Increments the value contained in the variable **LEVEL** . If **STATE** is interpreting and **LEVEL** was 0 before being incremented, switch to compile state and begin compiling an unnamed temporary colon definition.

-LEVEL (--)
: Decrements the value contained in the variable **LEVEL** . If **LEVEL** is 0 after having been decremented, execute the temporary colon definition, discard it, and return to interpret state.

These words are easy to implement on most systems. A "standard" implementation does not appear to be possible, because of differences in the way that different systems compile colon

definitions. Another implementation dependency arises from different interpreter organizations; in some systems, the interpreter and compiler are separate loops; in other systems, there is only one loop whose behaviour changes according to the value of STATE . Nevertheless, a person familiar with the internals of a particular system should have little difficulty figuring out how to do it for that system.

Implementation for F83

```
variable saved-dp  variable level
: +level  ( -- )
   level @  if
      \ If in compile state, just increment level
      1 level +!
   else  state @ 0=  if
      \ If in interpret state, switch to compile state
      1 level !
      here saved-dp !              \ Remember the start
      r> ['] ] >body >r >r         \ XXX Execute ] after the caller
   then then
;

: -level  ( -- )
   state @ 0= abort" Conditionals not paired"
   level @  if
      -1 level +!  level @ 0=  if
          compile exit                \ Finish the definition
          saved-dp @ here - allot     \ Reclaim the memory
          [compile] [                 \ Enter interpret state
          here >r                     \ YYY Execute the definition
      then
   then
;

: begin    +level  [compile] begin    ; immediate
: do       +level  [compile] do       ; immediate
: ?do      +level  [compile] ?do      ; immediate
: if       +level  [compile] if       ; immediate
: then     [compile] then    -level   ; immediate
: loop     [compile] loop    -level   ; immediate
: +loop    [compile] +loop   -level   ; immediate
: until    [compile] until   -level   ; immediate
: again    [compile] again   -level   ; immediate
: repeat   [compile] repeat  -level   ; immediate
```

The words **SAVED-DP** , **LEVEL** , **+LEVEL** , and **-LEVEL** take up about 200 bytes of dictionary space. If the calls to **+LEVEL** and **-LEVEL** are added to the kernel versions of the control structure words **BEGIN** , **IF** , **LOOP** , etc., instead of redefining them, the total increase in the size of the dictionary is just over 200 bytes.

Implementation Notes:

In the above example, there are two lines of code which are not entirely portable. The line marked XXX executes the word "] " (right-bracket) after the caller of +level . This is necessary in F83, because in F83 "] " is the compiler loop. If "] " were executed directly within **+LEVEL** , the rest of the control structure would be compiled before the beginning run-time word. For instance, in the case of **IF** , the rest of the control structure would be compiled before " **[compile] if**". For systems in which "] " simply sets the variable **STATE** (as in FIG Forth and MVP-Forth), the phrase:

```
r> ['] ] >body >r >r    \ XXX Execute ] after the caller
```

may be replaced by:

```
]
```

The line marked YYY causes the unnamed temporary colon definition to be executed when **-LEVEL** returns. For most threaded code Forth implementations, pushing the parameter field address on the return stack is a convenient (but not "standard") way to execute an unnamed colon definition. Other systems might need to use a different technique.

Compiler Extension Words

It is possible to add the **+LEVEL** function to the control structure defining words **<MARK** and **>MARK** (see Forth-83 Standard, Chapter 15), and to add the **-LEVEL** function to **<RESOLVE** and **>RESOLVE** , thus making the interpreted control structure behavior automatic for any words which use **xMARK** and **xRESOLVE** . Here is an implementation of these system extension words, with the additional features of compiler security and automatic compilation of the run-time word.

```
: +>mark    ( acf -- adr )  +level , >mark  ;
: +<mark    ( -- adr )      +level <mark  ;
: ->resolve ( adr chk2 chk1 -- )     pairs  >resolve -level  ;
: -<resolve ( adr chk2 acf chk1 -- ) rot ?pairs  ,  <resolve -level  ;

: begin  +<mark 1                                              ; immediate
: do     ['] (do)   +>mark 3                                   ; immediate
: ?do    ['] (?do)  +>mark 3                                   ; immediate
: if     ['] ?branch +>mark 2                                  ; immediate
: else   ['] branch  +>mark 2   2swap 2 ->resolve              ; immediate
: then   2 ->resolve                                           ; immediate
: loop   compile   (loop)  over 2+ <resolve  3 ->resolve       ; immediate
: +loop  compile  (+loop)  over 2+ <resolve  3 ->resolve       ; immediate
: until  ['] ?branch 1 -<resolve                               ; immediate
: again  ['] branch 1 -<resolve                                ; immediate
: repeat 2swap [compile] again  [compile] then                 ; immediate
: while  [compile] if                                          ; immediate
```

Conditional Compilation

Conditional compilation is the use of control structures (e.g. **IFTRUE** , **OTHERWISE** , **IFEND** , Forth-83 Standard Appendix B) to control the sequence of words compiled, rather

than the sequence of words executed. The temporary compilation technique does not address the issue of conditional compilation, it simply extends the utility of existing operators to the interpret state.

The names **IF** , **ELSE** , **THEN** are not suitable for conditional compilation anyway, because during conditional compilation you may very will wish to exclude code which already contains **IF** , **ELSE** , **THEN** , so it is necessary to distinguish between the compilation conditionals and the the execution conditionals.

Compiling Words in Control Structures

The use of compiling words, such as comma (,), colon (:), etc, within interpreted control structures will result in an error. The problem is that compiling words modify the contents of the dictionary just above HERE , which overwrites the temporary definition being executed.

This problem can easily be alleviated by compiling the temporary definition into a separate area away from **HERE** . That separate area should probably be called **COMPILE-BUFFER** . This modification is left to the reader as an exercise.

Error Recovery

In case an error occurs during the compilation of a temporary definition, the following code should be added to the **QUIT** routine:

```
level @  if
   level off   saved-dp @ here -  0 max allot
then
```

Prompting

The current nesting level can easily be shown at the Forth prompt. This code prompts with "ok" in interpret state, "]" in compile state, and " n]" during temporary compilation, where n is the current nesting level.

```
state @  if
   level @  ?dup if  1 .r  else  ."  "  then  ." ] "
else
   ." ok "
then
```

Acknowledgements

The notion of a compile buffer, a separate area where temporary definitions are compiled before they are executed, appeared in STOIC , a Forth-like language for 8080 systems [Sachs83]. The GRAFORTH system for Apple II computers is reputed to based on a compile-buffer technique.

[Baden85] describes a technique for accomplishing the effect of interpreted control structures. Baden's technique involves re-reading the input stream on each iteration of the loop. Consequently, interpreted loops run much slower than compiled loops.

Bibliography

[Sach83] Sachs, J.M. and Burns, S.K., "STOIC, and interactive programming system for dedicated computing", Software – Practice and Experience, Vol 13, 1983, pp 1–16.

[Baden85] Baden, Wil, "Interpretive Logic", 1985 Asilomar FORML Conference.