# How (Not) to Code a Finite State Machine

Douglas W. Jones*

March 30, 1988

## Abstract

The standard advice for those coding a finite state machine is to use a while loop, a case statement, and a state variable. This is usually bad advice! The reasons for this are explored here, and better advice is formulated.

## 1 Introduction

In a recent examination of compiler construction texts, I discovered that many newer texts advocate the same awkward approach to constructing a lexical analyzer [2, Section 3.6.4], [1, Sections 3.6 through 3.9], and [7, Section 6.2.3]. This approach results from the formal translation of a finite state model of lexical analysis to a program.

In contrast, older texts such as [4, Section 3.3] and [5, Section 8.8] may use finite state automata, but the reduction to code is largely intuitive and the resulting code is sometimes hard to follow, although the latter complaint may be largely because of the primitive programming languages used in the examples. The clearly written but intuitively derived lexical analyzer in [8, Section 5.8] illustrates this. A similar approach has been used in [6, Section 4.2]; notably, this text presents the lexical analyzer before any discussion of automata theory.

It is unfortunate that the majority of the material written on reducing finite state automata to code is in texts on compiler construction. Finite state machines play a sufficiently important role in a variety of computing applications to suggest that their use in programs should be introduced in lower level or even introductory programming texts.

## 2 A Finite State Machine

An example (Moore model) finite state transducer is shown in Figure 1. Here, the output in each state is the state name itself. When presented with the input string 000111011100 this machine would output ABBBCCCDCCCDA. This assumes an input alphabet limited to 2 characters.

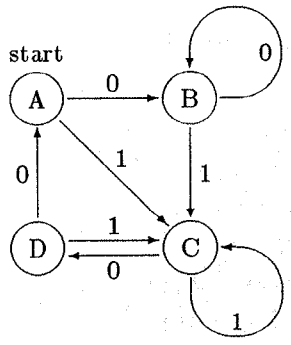


Figure 1: A finite state transducer.

*Author's current address: Department of Computer Science, University of Iowa, Iowa City, Iowa 52242; phone (319) 335-0740; CSNET jones@cs.uiowa.edu

# 3 The Wrong Way

When the finite state transducer in Figure 1 is converted into a program as advocated in [1,2], the following code results:

```
type
    states = (A, B, C, D)
var
    ch: char { the input symbol };
    state: states;
begin { FSM-1 }
    state := A;
    while true do case state of
        A: begin
               write('A');
               read(ch);
               if ch = '0'
               then state := B
               else state := C
           end;
        B: begin
               write('B');
               read(ch);
               if ch = '0'
               then state := B
               else state := C
           end;
        C: begin
               write('C');
               read(ch);
               if ch = '0'
               then state := D
               else state := C
           end;
        D: begin
               write('D');
               read(ch);
               if ch = '0'
               then state := A
               else state := C
           end;
    end { while case };
end { FSM-1 }
```

This code appears to be well structured, but this is misleading. The original finite state machine was not presented with any pretense of good structure, and the derived code faithfully reproduces the structure of the machine. The unstructured control transfers in the machine have been modeled in the code with assignments to the variable *state*. These assignments serve essentially as **goto** statements! Unfortunately the **while** and **case** statements that enclose the body of the program serve only to obscure this underlying control structure, not to clarify it.

# 4 A Better Way

Except in very special circumstances, it would seem preferable to admit that the original finite state machine was unstructured and make no attempt to hide this lack of structure. The following example shows the result of eliminating the cosmetic control structures used in FSM-1 and replacing them with **goto** statements:

```
label { entries for machine states }
    A, B, C, D;
var
    ch: char { the input symbol };
begin { FSM-2 }
A: write('A');
    read(ch);
    if ch = '0' then goto B else goto C;
B: write('B');
    read(ch);
    if ch = '0' then goto B else goto C;
C: write('C');
    read(ch);
    if ch = '0' then goto D else goto C;
D: write('D');
    read(ch);
    if ch = '0' then goto A else goto C;
end { FSM-2 }
```

This example illustrates the fact that any sequential procedure can be viewed as a finite state machine, with the program counter serving as the state variable. This relationship is well known to those who reduce algorithms to hardware, but it appears to be underutilized by those who write programs to model finite-state processes.

# 5 The Best Way

The finite state machine in Figure 1 actually had a reasonable control structure! This can be emphasized by redrawing it as shown in Figure 2. The style used here is similar to that used in recursive transition network (syntax chart) descriptions of languages (see [3], for example). Here, all loop entries come from above, iteration connections are drawn to the side, and loop exits are drawn to the bottom. The correspondence between the redrawn version and the regular expression $(0^*(1^+0)^+0)^+$ should be fairly clear (Note: * implies zero or more repetitions, + implies one or more).

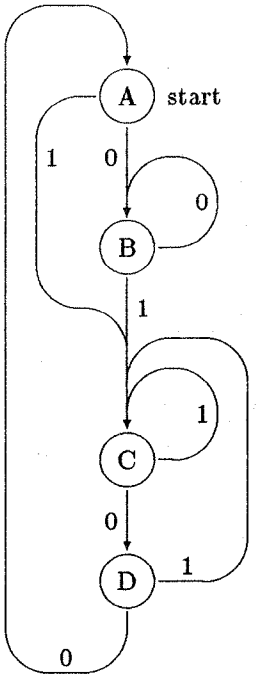


Figure 2: A structured version

The following well structured code can be derived from either the structured machine in Figure 2 or from the regular expression. It is interesting to note that this code is similar in style to many recursive descent parsers.

```
var
    ch: char { the input symbol };
begin { FSM-3 }
    repeat
        { state A }
        write('A');
        read(ch);
        while ch = '0' do begin
            { state B }
            write('B');
            read(ch);
        end { while };
        repeat
            repeat
                { state C }
                write('C');
                read(ch);
            until ch = '0';
            { state D }
            write('D');
            read(ch);
        until ch = '0';
    until false
end { FSM-3 }
```

A decent compiler should be able to generate the same object code from FSM-2 and FSM-3, and a program restructuring engine applied to FSM-2 should generate code similar to FSM-3. The code of FSM-3 strongly resembles the code of the intuitively written lexical analyzers referenced in Section 1, but it has a clear relationship to the finite state model from which it was derived.

# 6 Discussion

There is no excuse for using the nested **case** statement and **while** loop of FSM-1 in code that is automatically generated from finite state machines or regular expressions, unless the target language has no **goto** statement. When automatic code generation tools are adequate, adequately documented, and available throughout the lifetime of the programs they are used to create, programmers should never have to examine or modify the generated code. This implies that there should be no barriers to the use of **goto** statements, as in FSM-2, in such code.

Well structured code such as presented in FSM-3 seems preferable when a finite state machine must be reduced to code by hand, or when the code must be maintained without the aid of the tools used to produce it. Of course, this does not say that large amounts of code must be written to get rid of a **goto** statement; it is almost always better to use a **goto** than to assign to a state or condition variable that is used later to cause a control transfer.

The nested **case** statement and **while** loop of FSM-1 are well justified when the solution to a problem is best expressed by a system of multiple concurrent finite state machines. In such cases, the best approach is usually to place each finite state machine in a coroutine, using structured code such as that in FSM-3 for the body of each machine. Unfortunately, coroutine-based languages are not universally available, so simulations using a state variable and case statement for each coroutine are frequently the only practical approach.

If a state transition table is used to determine the next state, indexed by current state and input character, then a **case** statement and **while** loop are appropriate. Here again, the result is hard to maintain unless the tools used to produce the state table remain available throughout the life of the program. The difference in maintainability between an antique FORTRAN **block data** statement and a modern Turing **init** construct is not very large when dealing with a state table of hundreds of entries.

The examples presented here are an interesting test of software complexity metrics. All have the same *deep control structure* but they have different *shallow control structures.* Which complexity metrics are mislead by the apparent simplicity of the shallow control structure of FSM-1? Which metrics report that FSM-3 is more complex because of the deep nesting?

# References


[1] Aho, Alfred V., Sethi, R., and Ullman, J. *Compilers, Principles Techniques, and Tools.* Addison Wesley, Reading Mass., 1987.

[2] Barrett, W. A., and Couch, J. *Compiler Construction, Theory and Practice.* SRA, 1979.

[3] Cooper, D. *Standard Pascal User Reference Manual.* Norton, New York, 1983.

[4] Gries, David. *Compiler Construction for Digital Computers* Wiley, New York, 1971.

[5] McKeeman, W. M., Horning, J. J., and Wortman, D. B. *A Compiler Generator.* Prentice Hall, Englewood Cliffs, NJ, 1970.

[6] Tremblay, J., and Sorenson, P. G. *The Theory and Practice of Compiler Writing.* McGraw-Hill, New York, 1985.

[7] Waite, William M., and Goos, G. *Compiler Construction.* Springer Verlag, New York., 1984.

[8] Wirth, Niklaus. *Algorithms + Data Structures = Programs.* Prentice Hall, Englewood Cliffs, NJ, 1976.