



Fermi National Accelerator Laboratory

TM-1508

**Finite State Tables for
General Computer Programming Applications**

Mark Leininger
Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

January 1988



Operated by Universities Research Association Inc. under contract with the United States Department of Energy

Finite State Tables for General Computer Programming Applications

Mark Leininger
January, 1988

Abstract

The Finite State Table is a computer programming technique which offers a faster and more compact alternative to traditional logical control structures such as the IF-THEN-ELSE statement. A basic description of this technique is presented. The application example is the creation of plot output from engineering analysis and design models generated by I-DEAS*, a commercial software package used for solid modeling, finite element analysis, design and drafting.

* I-DEAS is a registered trademark of Structural Dynamics Research Corporation.

1. Introduction

The Finite State Table (FST) has found uses in the areas of language compilers and artificial intelligence. Although the application areas which most effectively exploit the power of the FST and rely most heavily on its use are themselves quite complex, the FST itself is a very simple and straightforward concept. Because the FST depends heavily on the use of pointers, it is most effectively implemented in languages like C or Pascal. However, the concept could be applied in Fortran applications as well. The application examples in this paper will be in C. The specific comparison will be between the FST and IF-THEN-ELSE structure.

2. Conceptual Model of the FST

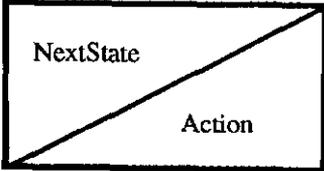
The easiest way to explain the FST is by a simple example. Figure 1 is the FST for a coffee vending machine. Coffee costs fifteen cents and can be purchased using a dime and a nickel or three nickels. No change is given so quarters are not allowed. The two columns are labeled DepositFive and DepositTen, representing input actions to the FST of depositing five cents and depositing ten cents respectively. The three rows are labeled StartState, FiveState, and TenState; they represent the three states the machine can assume. StartState represents the starting state, the state in which a customer finds the machine when no money has been deposited. The FiveState and TenState represent the state of the machine after five or ten cents have been deposited respectively. Each entry in the FST is described by its row and column number. For consistency with later examples, row and column numbers will begin at zero. So, StartState identifies row zero, FiveState identifies row one; DepositFive identifies column zero, etc.

To explain the entries in the FST, assume a nickel is deposited in the machine while it is in StartState. Looking at the entry at row zero and column zero, you will see FiveState above a diagonal line and DoNull below the diagonal line. FiveState represents the next state the machine will assume when a nickel is inserted while the machine is in StartState. DoNull represents the action the machine takes as a result of the customer depositing a nickel. As the name implies, DoNull means the machine does nothing, it simply moves to the next state and waits for another coin to be deposited. Each entry in the FST consists of the next state above the diagonal line and the resulting action below the diagonal line. Returning to the customer who has deposited a nickel, the machine is in FiveState. If the customer deposits a dime, the corresponding FST entry is found in row 1 (FiveState), column 1 (DepositTen). The entry indicates that the next state of the machine is StartState. This is correct because the customer has now deposited a total of fifteen cents so the machine is ready to serve the next customer. The entry also indicates that the action the machine takes is DoCoffee. As the name implies, DoCoffee represents the

Figure 1

Finite State Table for Coffee Vending Machine

	DepositFive	DepositTen
StartState	FiveState DoNull	TenState DoNull
FiveState	TenState DoNull	StartState DoCoffee
TenState	StartState DoCoffee	StartState DoReturn



physical action of dispensing a cup of coffee. This completes one cycle through the FST resulting from a customer depositing a nickel followed by a dime.

The only other entry in the FST which requires explanation is row two, column one. This entry in the FST would be reached after a customer accidentally deposited two dimes. Since this machine can make no change, the FST entry indicates that the machine is to take the action DoReturn, which returns both dimes to the customer. The next state is StartState.

3. Comparison of the FST to the IF Statement

A typical situation confronting the programmer is to read some input, make a logical decision based on that input and take some appropriate action. For example, reading the integer 0 from a plot file may cause the pen on the plotter to be raised; reading the integer 1 causes the pen to be lowered. The typical way a program makes such a logical decision is with an IF-THEN-ELSE structure. Pseudocode for such a situation would be:

```

read Command from plot file
if ( Command = 0 ) then
    raise pen
else if ( Command = 1 ) then
    lower pen
end if

```

Doing the same thing using an FST requires knowing a little about C. C allows the definition of complex data types. In the case of finite state tables, the data type needed is an array of pointers to functions (a function in this context is equivalent to a subroutine in Fortran). Fortran programmers are familiar with arrays of integer or real values, but C allows arrays of mixed, complex data types. So, for example, an array element in C can be a pointer to a function. The significance of this is that the logical decisions are made ONCE at compile time rather than every time input is read at execution time. For example, the above IF statement could be replaced by the following pseudocode employing an FST:

```

read Command from input
execute TableArray [CurrentState][Command]

```

TableArray is a two dimensional array of pointers to functions. Assuming the CurrentState is equal to 0, if Command is 0 the array element invoked will be TableArray [0][0]. This array element contains a pointer to the function which raises the pen. If Command is 1, the array element invoked will be TableArray [0][1]. This element points to the function which lowers the pen. The actual syntax is more complex because the pointers have to be

dereferenced but the important point is that no logical decision needs to be made at execution time when using the FST; the logic is built into the table itself at compile time. The IF statement requires a logical decision at execution time every time an input command is read. Obviously, one of the benefits one expects from using an FST is a significant time improvement. A second and probably more important advantage of the FST is the compact and concise notation which allows easy debugging. The logical flow can be followed by stepping through the table itself rather than trying to understand IF statements.

To compare the syntax of the FST versus an IF statement, the coffee machine used as a conceptual model will be implemented in C using both the FST and IF statement. Figures 2 and 3 included at the end of this document are the source code which implements the operation of the coffee machine. Figure 2 uses the FST, figure 3 uses the IF statement. Note the portion of the code in each case which controls the logical flow is in bold face and has been included below for convenient reference. First the FST:

```
while ( Coin != -1 ) {
    TabPtr = &FstTable [CurrentState][Coin/5-1];
    (*TabPtr->FuncPtr)( FpOutput );
    CurrentState = (TabPtr->NextState);
    fscanf ( FpSource, "%d", &Coin ); /*
end_while */ }
```

Now the IF statement:

```
while ( NextCoin != -1 ) {
    if ( TotalCoin == 5 )
        DoNull ( FpOutput );
    else if ( TotalCoin == 10 )
        DoNull ( FpOutput );
    else if ( TotalCoin == 15 ) {
        DoCoffee ( FpOutput );
        TotalCoin = 0; }
    else if ( TotalCoin > 15 ) {
        DoReturn ( FpOutput );
        TotalCoin = 0; } /*
end_if */

    fscanf ( FpSource, "%d", &NextCoin );
    TotalCoin += NextCoin; /*
end_while */ }
```

For this simple problem the two pieces of code above are relatively equal in size but if there were a thousand decisions to make rather than just three, the IF statement would need to be approximately 300 times longer while the FST

would remain identical. The only change in the FST would be in the declaration of the table itself at compile time.

For each of these examples, input is read from a file TEST.DAT and the output, consisting simply of printing the action to be taken, is written to a file ACTIONS.DAT. The end of the input is signaled by a -1 as data. For example, the following two lines of input:

```
5 5 5
-1
```

would result in the following three lines of output:

```
*** DoNull ***
*** DoNull ***
*** DoCoffee ***
```

4. Application Example

I-DEAS is an engineering software package which produces a generic plot file called a structured picture file. To generate hardcopy plots, a variety of devices is used, including laser printers and pen plotters. A package of graphics subroutines is available which allows the creation of a device specific plot file simply by linking different device drivers. The application for the FST is to read the structured picture file and invoke the appropriate graphics subroutines to generate a device specific plot file which can be used to produce hardcopy. A detailed description of the structured picture file can be found in the I-DEAS User's Guide, page 513-531. A simple example of the commands in the structured picture file is:

```
0 2 4 6 0
0 0 1 1
1.000000 0.772727 0.000000 0.000000 1.000000
0.772727
```

The above command contains information on the size of the graphics and text areas. The first two integers, 0 and 2, indicate the command class and command number respectively. These uniquely define what action needs to be taken and are the basis for the construction of the finite state table. The next three integers, 4 6 0, indicate that the following data consists of 4 integers, 6 reals, and 0 characters. The data itself then follows. The actual commands involved will not be discussed further. Figure 4 is a portion of the FST required for this application. Figure 5 contains the portion of the code which actually implements the FST. The functions themselves have not been included in Figure 5 because they are simply graphics routines.

The point to notice is that although this example is quite complex, the FST in Figure 4 presents the logic in an easy to understand way. The portion of the code required to process a structured picture file is bold faced in Figure 5; it is reproduced here for convenience:

```

while ( CurrentState != EOFState ) {
    fscanf ( FpSource, "%d", &CurrentState );
    fscanf ( FpSource, "%d", &CommandNumber );
    TabPtr = &FstTable [CurrentState][CommandNumber];
    (*TabPtr->FuncPtr)( FpSource, FpOutput );
    CurrentState = (TabPtr->NextState); /*
end_while */ }

```

This same logical control would require several pages of IF statements. Note that the declarations in the source code in Figure 5 needed to initialize the finite state table are quite long. These declarations can be incorporated into an include statement to reduce the apparant size of the actual code.

5. Summary

The finite state table is an alternative way to provide logical control within a program and, as the complexity of the logic increases, offers the following advantages over other structures like the IF statement:

- 5.1 Compact notation;
- 5.2 Easy for programmer to follow program flow and debug;
- 5.3 Logic is incorporated at compile time rather than execution time.

Figure 2

Source code for the FST implementation of coffee machine

```

#include <stdio.h>

#define NumRow    3
#define NumCol    2

#define StartState    0
#define FiveState     1
#define TenState      2

main()
/*=====*/
{
FILE *FpSource;
FILE *FpOutput;

int CurrentState=0;
int Coin;

    /** Declare functions used by finite state table ***/

int
    DoCoffee(),
    DoReturn(),
    DoNull();

struct Table
{
    int NextState;
    int ( *FuncPtr )();
} FstTable[NumRow][NumCol] =

{
    /* Start structure initialization */
    {
        /* begin row 1, StartState */
        { FiveState, DoNull },
        { TenState, DoNull }
    },
    /* end row 1 */

    {
        /* begin row 2, FiveState */
        { TenState, DoNull },
        { StartState, DoCoffee }
    },
    /* end row 2 */

    {
        /* begin row 3, TenState */
        { StartState, DoCoffee },
        { StartState, DoReturn }
    },
    /* end row 3 */
};
/* end structure initialization */

```

(continued on next page)

```

struct Table *TabPtr;

FpOutput = fopen ( "ACTIONS.DAT", "w" );
FpSource = fopen ( "TEST.DAT", "r" );

fscanf ( FpSource, "%d", &Coin );
while ( Coin != -1 ) {
    TabPtr = &FstTable [CurrentState][Coin/5-1];
    (*TabPtr->FuncPtr)( FpOutput );
    CurrentState = (TabPtr->NextState);
    fscanf ( FpSource, "%d", &Coin ); /*
end_while */ }

fclose ( FpSource ); /*
fclose ( FpOutput ); /*

end main */ }

DoNull ( Fp1 )
/*=====*/

FILE *Fp1;
{
fprintf ( Fp1, "\t*** DoNull ***\n" );
}

DoCoffee ( Fp1 )
/*=====*/

FILE *Fp1;
{
fprintf ( Fp1, "\t*** DoCoffee ***\n" );
}

DoReturn ( Fp1 )
/*=====*/

FILE *Fp1;
{
fprintf ( Fp1, "\t*** DoReturn ***\n" );
}

```

Figure 3

Source code for IF implementation of coffee machine.

```

#include <stdio.h>

main()
/*=====*/
{
FILE *FpSource;
FILE *FpOutput;

int TotalCoin, NextCoin;

FpOutput = fopen ( "ACTIONS.DAT", "w" );
FpSource = fopen ( "TEST.DAT", "r" );

fscanf ( FpSource, "%d", &NextCoin );
TotalCoin = NextCoin;

while ( NextCoin != -1 ) {

    if ( TotalCoin == 5 )
        DoNull ( FpOutput );
    else if ( TotalCoin == 10 )
        DoNull ( FpOutput );
    else if ( TotalCoin == 15 ) {
        DoCoffee ( FpOutput );
        TotalCoin = 0; }
    else if ( TotalCoin > 15 ) {
        DoReturn ( FpOutput );
        TotalCoin = 0; } /*
end_if */

    fscanf ( FpSource, "%d", &NextCoin );
    TotalCoin += NextCoin; /*

end_while */ }

fclose ( FpSource );
fclose ( FpOutput ); /*

end main */ }

DoNull ( Fp1 )
/*=====*/
FILE *Fp1; {
fprintf ( Fp1, "\t*** DoNull ***\n" ); }

DoCoffee ( Fp1 )
/*=====*/
FILE *Fp1; {
fprintf ( Fp1, "\t*** DoCoffee ***\n" ); }

DoReturn ( Fp1 )
/*=====*/
FILE *Fp1; {
fprintf ( Fp1, "\t*** DoReturn ***\n" ); }

```

Figure 4

FST for application example

	0	1	2	3	4
StartState	ControlState DoNull	GraphPrimState DoNull	GraphAttrState DoNull	RasterState DoNull	AnimateState DoNull
ControlState	EOFState DoError	StartState DoDisplyInfo	StartState DoSourceDev	StartState DoNull	StartState DoViewPort
GraphPrimState	EOFState DoError	StartState DoPolyLine	StartState DoPolyMark	StartState DoPolyGon	StartState DoTextOutput
GraphAttrState	EOFState DoError	StartState DoResetAttr	StartState DoTextSize	StartState DoCharGap	StartState DoLineStyle
RasterState	EOFState DoError	StartState DoNull	StartState DoNull	StartState DoNull	StartState DoNull
AnimateState	EOFState DoError	StartState DoNull	StartState DoNull	StartState DoNull	StartState DoNull

Figure 5
Source code for application example.

```

#include <stdio.h>

#define NumRow    6
#define NumCol 15

#define StartState      0
#define ControlState    1
#define GraphPrimState  2
#define GraphAttrState  3
#define RasterState     4
#define AnimateState    5
#define EOFState        6

main() /*
=====

begin main */ {

FILE *FpSource;
FILE *FpOutput;

int CurrentState;
int CommandNumber;

    /** Declare functions used by finite state table ***/

int
    /** Control Commands ***/
    DoDisplayInfo(),
    DoSourceDevInfo(),
    DoViewPortDef(),
    DoGraphContext(),
    DoClipPort(),
    DoEOF(),

    /** Graphic Primitive Commands ***/
    DoPolyLine(),
    DoPolyMark(),
    DoPolyGon(),
    DoTextOutput(),
    DoArc(),

(continued on next page)

```



```

{ /* begin row 2, ControlState */
  { EOFState, DoError },
  { StartState, DoDisplayInfo },
  { StartState, DoSourceDevInfo },
  { StartState, DoNull },
  { StartState, DoViewPortDef },
  { StartState, DoGraphContext },
  { StartState, DoClipPort },
  { EOFState, DoEOF },
  { EOFState, DoError },
}, /* end row 2 */

{ /* begin row 3, GraphPrimState */
  { EOFState, DoError },
  { StartState, DoPolyLine },
  { StartState, DoPolyMark },
  { StartState, DoPolyGon },
  { StartState, DoTextOutput },
  { StartState, DoArc },
  { EOFState, DoError },
}, /* end row 3 */

{ /* begin row 4, GraphAttrState */
  { EOFState, DoError },
  { StartState, DoResetAttr },
  { StartState, DoTextSize },
  { StartState, DoTextPrec },
  { StartState, DoCharGap },
  { StartState, DoCharPath },
  { StartState, DoLineStyle },
  { StartState, DoMarkSymbol },
  { StartState, DoMarkSize },
  { StartState, DoIntStyle },
  { StartState, DoExtStyle },
  { StartState, DoLineColor },
  { StartState, DoFillColor },
  { StartState, DoMarkColor },
  { StartState, DoTextColor }
}

```

