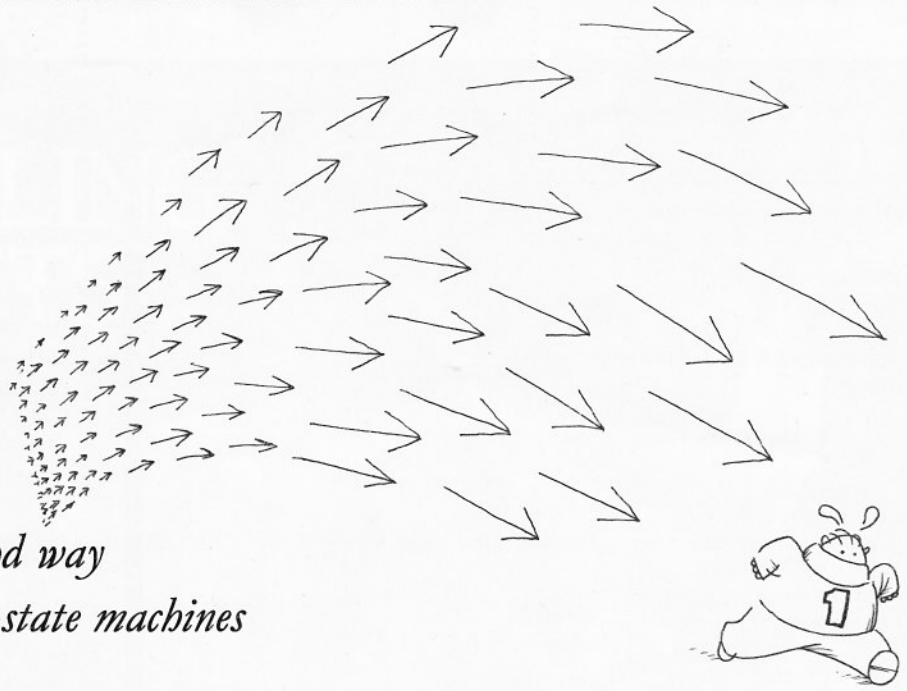
Goto? Yes, goto!



There's only one good way to implement finite-state machines

Tim Cooper

I often read about finite-state machines and how to implement them. I can't help but violently disagree with what some people say about FSMs—for example, to hear someone suggest an array of function pointers (of all things) or say that the *goto* method compromises good structured programming. After seeing my example code and benchmarks, maybe you'll be convinced that there is only one way to implement FSMs that wins on both counts of efficiency and good programming.

Overview of FSMs

For the uninitiated, an FSM is an algorithm that is often used to process some kind of incoming data. It can be in any one of many states, and the way it processes the input depends on its state. For example, a C compiler that uses an FSM to process a C program will process a word differently if it is inside a comment or string constant.

FSMs are frequently used for text processing because they are good for implementing syntax diagrams or rules. The Turing machine is built on the concept of an FSM. Many other things can be considered FSMs: a word processor that jumps between *Opening, Editing, Printing,* and *Special Functions* modes following a strict set of rules but not necessarily a particular hierarchy or cycle.

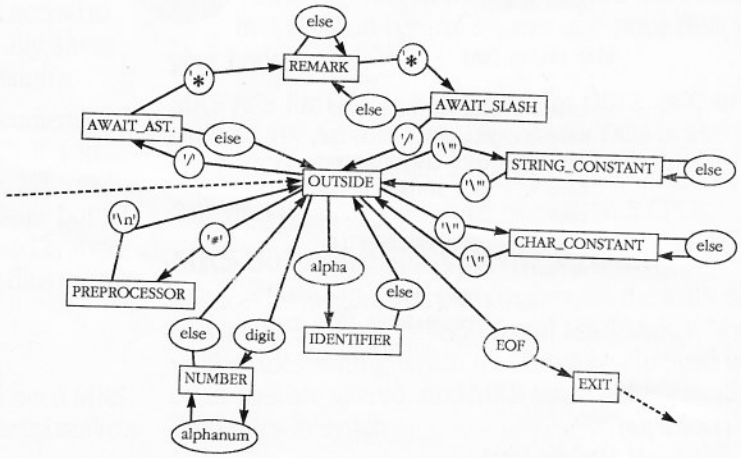
FSMs are often visualized with the help of a graph diagram. The parser implemented in Listing 1 processes C programs according to the FSM shown in Figure 1.

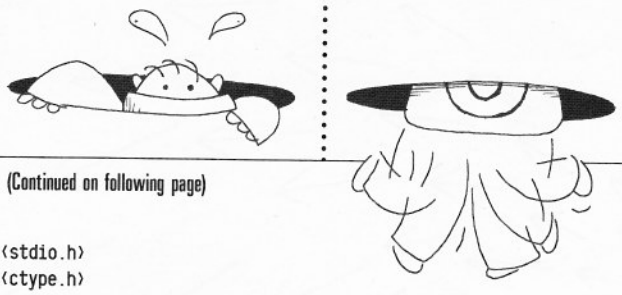
After learning about FSMs, I designed an extension to C, a new keyword that would allow easy and efficient implementation of FSMs. To my surprise, I discovered I could implement this keyword myself using one simple *#define* statement: *#define state(s) s: ch = fgetc (text); keep##s.*

ch is a global *char* variable, and *text* is the *FILE* * for the input text. (If you want to implement something other than a text processor, it is easy to modify this macro.)

FIGURE 1.

Finite-state machine





LISTING 1. (Continued on following page)

```
include <stdio.h>
include <ctype.h>

#define state(s) s: ch = fgetc(prog); keep##s
#define or | |

typedef enum {identifier, number operator} TokenType;

FILE * prog;
char ch, id[32];
int idx;          /* Identifier buffer and buffer index */

void ProcessToken (TokenType t);
void parse(void);

main(int argc, char * argv[])
  {
   (_ctype+1)['_'] = 1;              /* This is an efficient way to get */
   if (prog = fopen (argv[1], "r"))  /* isalpha to recognize underscores */
      parse ();                      /* as alphas (Microsoft C). */
  }

void parse(void)
  {
   state(OUTSIDE):if (isspace (ch))     /* Starting state */
                    goto OUTSIDE;
                  if (isalpha (ch))
                    {
                     id[0] = ch;
                     idx = 1;
                     goto IDENTIFIER;
                    }
                  else if (isdigit (ch))
                    {
                     id[0] = ch;
                     idx = 1;
                     goto NUMBER;
                    }
                  else switch (ch)
                    {
                     case '#': goto PREPROCESSOR;
                     case '/': goto AWAIT_ASTERISK;
                     case '\'': goto CHAR_CONSTANT;
                     case '\"': goto STRING_CONSTANT;
                     case '\n': goto OUTSIDE;
                     case EOF:  goto END_OF-FILE;
                     default:   id[0] = ch;
                                idx = 1;
                                ProcessToken (operator);
                                goto OUTSIDE;
                    }

state (IDENTIFIER):
              if (isalnum (ch))
                {
                 id[idx++] = ch;
                 goto IDENTIFIER;
                }
```

Note the double hatch symbol—it is the ANSI operator for merging. It concatenates *keep* onto the front of whatever argument you give it, thereby constructing a single identifier.

How to use the macro

The macro is used to mark the beginning of each state. It is used in the form: *state (STATE_NAME): ...code....* To write an FSM, you begin with the statement: *state (STATE_NAME): ...etc....* The code follows the colon.

To transfer to another state, just write *goto STATE_NAME* and you will go to that new state. The next character will be read automatically. To exit, you need to create an exit state that is the last state: *state (EXIT):*. If this is the last state, processing will continue on to the code following the FSM.

Of course, there are alternative ways to exit, such as setting your own label at the end of the code (this avoids the problem of *EXIT* being reached on an EOF, then trying to read the next character) or using *return* statements to return from the middle of the FSM (if the error state is entered).

Sometimes the FSMs are not completely straightforward, and you find yourself in a situation where you need to jump to another state without gobbling up another character. For example, you have to see the next character before you know that you know whether a slash (/) is an comment delimiter or an operator.

One popular but inefficient method has been to use *unget* statements to push the character back onto the stream. This macro method avoids this by using the *keepSTATE_NAME* label. This label is set in every state by the macro, just after the *read character* statement. (The name comes from the macro and the colon comes from the code.) To jump to another

state in this way, simply write: *goto keepSTATE_NAME:* and you will transfer to the new state without reading the next character. (You *keep* the current character.)

How the macro works

It is quite easy to understand how this macro works. The *s:* statement sets up a label with the name of the macro argument. The next statement reads a character from text (using the function *version*) to do it. The final part is used to set the *keepSTATE_NAME* label. The colon is omitted because for aesthetic purposes I prefer to include it in the code. (It then resembles a *switch* statement.)

Comparing other methods

It may seem funny to claim a *goto* method makes for good structured programming. Here, the *goto*s are being used to move around an FSM, not to create spaghetti code. Their use is strictly proscribed by the FSM's specification. FSMs are typically envisaged as a network of states or a syntax diagram; they cannot generally be arranged into a linear flow or hierarchy. The traditional branching and looping constructs are inconvenient and the *goto* statement is completely natural.

I don't believe the *goto* statement could be more efficient. The *switch* method involves, for each new character, a jump to the correct case statement, the code, assignment of the new state number, a *break* to get to the bottom of the *switch,* and a third jump back to the top of the loop. The macro method accomplishes all this with a single jump.

I tested my method against the *switch* method and found it to be about 1.7 times quicker. (I tested it on pure code that did nothing but move around the FSM and worked out of memory.) It was also more compact.

Listing 1 illustrates this method using a program that processes C files. Although it is intended as an example, it is tight and readable code and could easily be adapted to any use. It is the core of a program I recently wrote that draws a tree of function calls, an index, and a

LISTING 1. (Continued on following page)

```
            else
               {
               ProcessToken (identifier);
               goto keepOUTSIDE;
               }

state (NUMBER):if (isalnum (ch))
               {
               id[idx++] = ch;
               goto NUMBER;
               }
            else
               {
               ProcessToken (number);
               goto keepOUTSIDE;
               }

state (CHAR_CONSTANT):
            if (ch == '\' ')
               goto OUTSIDE;
            else if (ch == '\\')
               fgetc(prog);
            goto CHAR_CONSTANT;

state (STRING_CONSTANT):
            if (ch == '\" ')
               goto OUTSIDE;
            else if (ch == '\\')
               fgetc(prog);
            goto STRING_CONSTANT;

state (AWAIT_ASTERISK):
            if (ch == '*')
               goto REMARK;
            else
               {
               id[idx=0] = '/';
               ProcessToken (operator);
               goto keepOUTSIDE;
               }

state (REMARK):if (ch == '*')
               goto AWAIT_SLASH;
            else goto REMARK;

state (AWAIT_SLASH):
            if (ch == '/')
               goto OUTSIDE;
            else goto REMARK;
```

boxed style of printout.

The program uses an FSM to examine a C program. All preprocessor statements, remarks, and white-space characters are stripped out. All character and string constants are also left unprocessed. All identifiers are found and sent to *ProcessToken()* with a parameter denoting that they are identifiers. All numerical constants are similarly processed, and everything else is sent as single characters with the parameter labeling them operators. (The FSM in Figure 1 is a simplification and does not deal with the "operator states" of C.) The *ProcessToken()* function simply prints the token and its type.

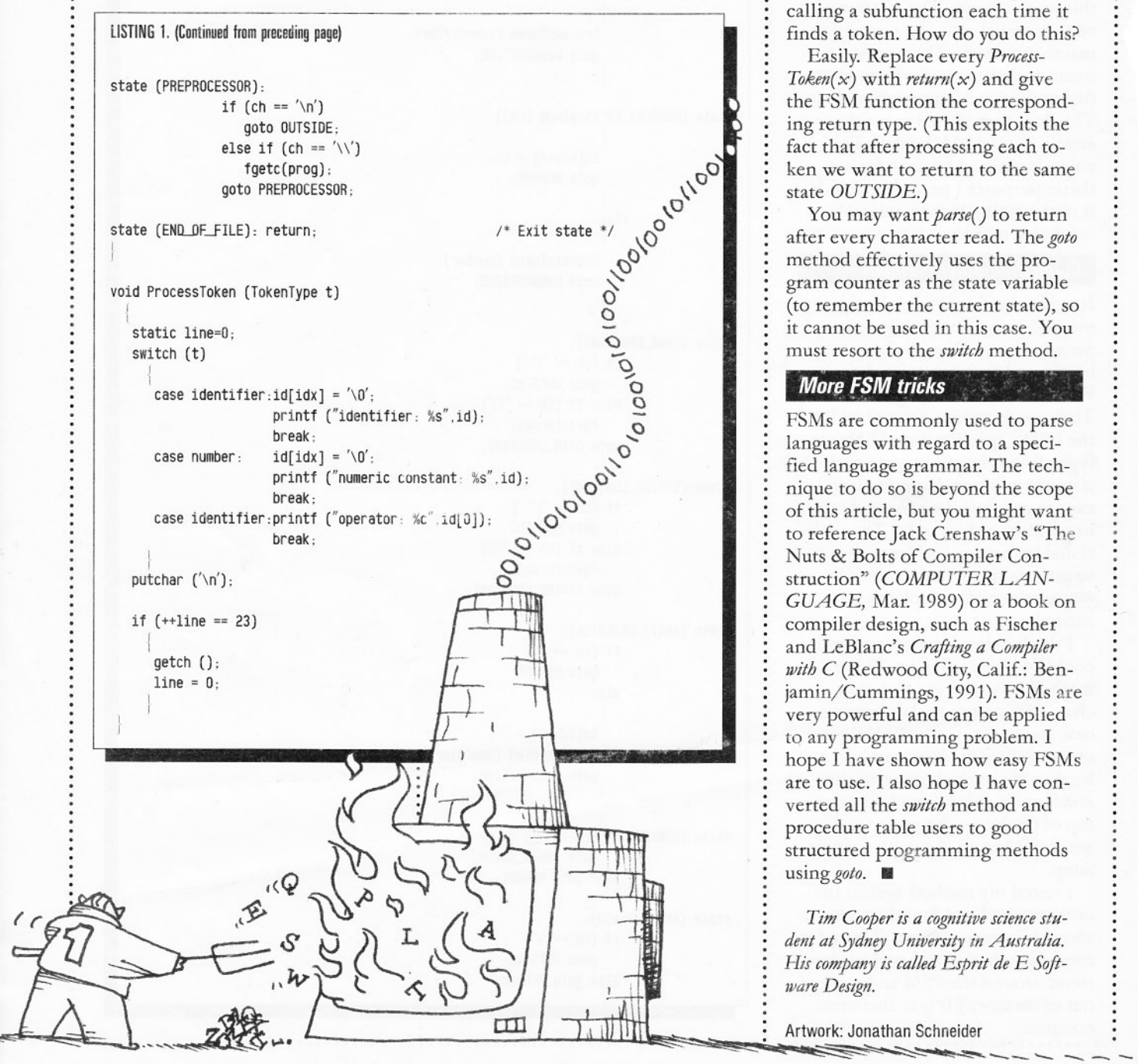
LISTING 1. (Continued from preceding page)

```
state (PREPROCESSOR):
              if (ch == '\n')
                 goto OUTSIDE;
              else if (ch == '\\')
                 fgetc(prog);
              goto PREPROCESSOR;

state (END_OF_FILE): return;                    /* Exit state */


void ProcessToken (TokenType t)
  {
   static line=0;
   switch (t)
     {
      case identifier:id[idx] = '\0';
                      printf ("identifier: %s",id);
                      break;
      case number:    id[idx] = '\0';
                      printf ("numeric constant: %s",id);
                      break;
      case identifier:printf ("operator: %c",id[0]);
                      break;
     }
   putchar ('\n');

   if (++line == 23)
     {
      getch ();
      line = 0;
     }
  }
```



Using parse()

Often you will want the main program to call *parse()* each time it wants a token, rather than *parse()* calling a subfunction each time it finds a token. How do you do this?

Easily. Replace every *ProcessToken(x)* with *return(x)* and give the FSM function the corresponding return type. (This exploits the fact that after processing each token we want to return to the same state *OUTSIDE*.)

You may want *parse()* to return after every character read. The *goto* method effectively uses the program counter as the state variable (to remember the current state), so it cannot be used in this case. You must resort to the *switch* method.

More FSM tricks

FSMs are commonly used to parse languages with regard to a specified language grammar. The technique to do so is beyond the scope of this article, but you might want to reference Jack Crenshaw's "The Nuts & Bolts of Compiler Construction" (*COMPUTER LANGUAGE*, Mar. 1989) or a book on compiler design, such as Fischer and LeBlanc's *Crafting a Compiler with C* (Redwood City, Calif.: Benjamin/Cummings, 1991). FSMs are very powerful and can be applied to any programming problem. I hope I have shown how easy FSMs are to use. I also hope I have converted all the *switch* method and procedure table users to good structured programming methods using *goto*. ■

Tim Cooper is a cognitive science student at Sydney University in Australia. His company is called Esprit de E Software Design.

Artwork: Jonathan Schneider