NFAs and DFAs can be hard-coded using this pattern:

```python
state := start state
c := first char
while (true) {
case state of {
  1: case c of {
      char1 : {
        c := nextChar();
        state := new state;
      }
  2: case c of {
      char2 : {
        c := nextChar();
        state := new state;
      }
      char3 : {
        return; /* accept */
      }
  }
}
```
We can also encode the transitions directly into a **transition table**:

<table>
<thead>
<tr>
<th>state</th>
<th>char(_1)</th>
<th>char(_2)</th>
<th>other</th>
<th>Accepting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>[3]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

States in brackets don’t consume their inputs. Accepting states are indicated by a √. Empty entries represent error states.
Given the table, we can write an interpreter to perform lexical analysis of any DFA:

```plaintext
state := 1
c := first char
while not ACCEPT[state] do {
    newstate := NEXTSTATE[state, c]
    if ADVANCE[state, c] then
        c := nextChar()
    state := newstate
}
if ACCEPT[state] then accept;
```
Table-driven C Comments
Table-driven C Comments...

class Comments {
    public static final int SLASH = 0;
    public static final int STAR = 1;
    public static final int OTHER = 2;
    public static final int END = 3;

    static int[][] NEXTSTATE = {
        // /*/ */ */ other
        { 1, -1, -1 },
        {-1, 2, -1 },
        { 2, 3, 2 },
        { 4, 3, 2 },
        {-1, -1, -1 }
    };
}
static boolean[] ACCEPT =
    {false,false,false,false,true};

static boolean[][] ADVANCE = {
    //  "/"    "*"    other
    {true,   true,   true},
    {true,   true,   true},
    {true,   true,   true},
    {true,   true,   true},
    {true,   true,   true}};


static String input;
static int current = -1;

static int nextChar() {
    int ch;
    current++;
    if (current >= input.length()) return END;
    switch (input.charAt(current)) {
        case '/': { ch = SLASH; break; }
        case '*': { ch = STAR; break; }
        default: { ch = OTHER; break; }
    }
    return ch;
}
public static boolean interpret () {
    int state = 0;
    int c = nextChar();
    while ((c != END) && (state>=0) && !ACCEPT[state]) {
        int newstate = NEXTSTATE[state][c];
        if (ADVANCE[state][c]) {
            c = nextChar();
            state = newstate;
        }
    }
    return (state>=0) && ACCEPT[state];
}

public static void main (String[] args) {
    input = args[0];
    boolean result = interpret();
}
Let’s do the same thing again, but this time we will hard-code the interpreter using switch-statements.

nextChar and the constant declarations are the same as for the previous program.
class Comments {
    // Declarations of SLASH,STAR,OTHER,END, and nextChar().
    public static boolean interpret() {
        int state = 0;
        int ch = nextChar();
        while(true) {
            switch (state) {
                case -1 :
                    return false;
                case 0 :
                    switch (ch) {
                        case SLASH:ch=nextChar();state=1;break;
                        default :return false;
                    }
                    break;
            }
        }
    }
case 1:
    switch (ch) {
        case STAR: ch=nextChar(); state=2; break;
        default : return false;
    }
    break;
case 2:
    switch (ch) {
        case SLASH: ch=nextChar(); state=2; break;
        case STAR : ch=nextChar(); state=3; break;
        case OTHER: ch=nextChar(); state=2; break;
        default : return false;
    }
    break;
case 3 :
    switch (ch) {
        case SLASH: ch=nextChar(); state=4; break;
        case STAR : ch=nextChar(); state=3; break;
        case OTHER: ch=nextChar(); state=2; break;
        default    : return false;
    }
    break;

case 4 :
    return (ch == END);
}
From REs to NFAs
We will describe our tokens using REs, convert these to an NFA, convert this to a DFA, and finally code this into a program or a table to be interpreted:

We will next show how to construct an NFA from a regular expression. This algorithm is called Thompson’s Construction (after Ken Thompson of Bell Labs).
Thompson’s Construction

- Each piece of a regular expression is turned into a part of an NFA.
- Each part is glued together (using $\epsilon$-transitions) into a complete automaton.
- An RE matching the character a translates into
  
  ![Diagram](image)

- An RE matching $\epsilon$ translates into
  
  ![Diagram](image)
We represent an RE component $r$ by the figure:

An RE matching the regular expression $r$ followed by the regular expression $s$ ($rs$) translates into
The regular expression $r | s$ translates into

![Diagram of Thompson's Construction for Alternation]
Thompson’s Construction – Repetition

The regular expression $r^*$ translates into

![Diagram of Thompson's Construction](image-url)
The regular expression \texttt{ab|a} translates into

![Diagram showing the construction of a DFA for the regular expression \texttt{ab|a}]

- First state with \( \epsilon \) transition to itself
- Transition on \( a \) from to first state to second state
- Transition on \( \epsilon \) from second state to third state
- Transition on \( b \) from third state to fourth state
- Transition on \( \epsilon \) from fourth state to fifth state
- Loop on \( \epsilon \) in fifth state

This diagram represents the automaton constructed using Thompson’s Construction for the given regular expression.
The regular expression `letter(letter|digit)*` translates into
From NFA to DFA
We now know how to translate a regular expression into an NFA, and how to translate a DFA into code. The missing piece is how to translate an NFA into a DFA.
From NFA to DFA...

- Each state in the DFA corresponds to a set of states in the NFA.
- The DFA will be in state \(2, 3, 4\) if the NFA could have been in any of the states \(\{2, 3, 4\}\).
- After reading \(a_1a_2\cdots a_n\) the DFA is in a state that represents the states the NFA could be in after seeing the input \(a_1a_2\cdots a_n\).
From NFA to DFA...

- A in the DFA represents the set of states \{1, 2, 4\} in the NFA. These are the states the FAs could be in before any input is consumed (the start states).

- B in the DFA represents the set of states \{2, 3, 4\} in the NFA. These are the states we can get to on the symbol a from A.
We need three functions:

1. \texttt{\textit{$\epsilon$-closure}(T)} is the set of NFA states reachable from some NFA state \( s \) in \( T \) on \( \epsilon \)-transitions alone. This is essentially a graph exploration algorithm that finds the nodes in a graph reachable from a given node.

2. \texttt{\textit{move}(T,a)} is the set of NFA states to which there is a transition on input symbol \( a \) from some NFA state \( s \in T \).

3. \texttt{\textit{SubsetConstruction}(N)} returns a DFA \( D=\text{\textit{Dstates}},\text{\textit{Dtrans}} \) corresponding to NFA \( N \).
procedure $\epsilon$-closure($T$)
  push all states in $T$ onto stack
  $C := T$
  while stack is not empty do
    $t := \text{pop}(\text{stack})$
    for each edge $t \xrightarrow{\epsilon} u$ do
      if $u$ is not in $C$ then
        $C := C \cup u$
        push(stack, $u$)
  return $C$
\( \varepsilon\)-closure(\(T\)) – Example

\( \varepsilon\)-closure(1) = \{1, 2, 4\}

\( \varepsilon\)-closure(2) = \{2\}

\( \varepsilon\)-closure(4) = \{2, 4\}

\( \varepsilon\)-closure(\{3, 4\}) = \{2, 3, 4\}
move\((T,a)\) – Example

\[
\begin{align*}
\text{move}(&\{1\}, a) = \{2, 3\} \\
\text{move}(&\{2, 3\}, b) = \{4\}
\end{align*}
\]
procedure SubsetConstruction(NFA N)
    Dstates := \{ \epsilon\text{-closure}(s_0) \}
    Dtrans := \{
    repeat
        T := an unexplored state in Dstates
        for each input symbol \( a \) do
            U := \epsilon\text{-closure}(\text{move}(T,a))
            if \( U \) is not in Dstates then
                Dstates := Dstates \cup U
                Dtrans := Dtrans \cup (T \xrightarrow{a} U)
        until all states have been explored
    return (Dstates,Dtrans)
NFA $\Rightarrow$ DFA

The diagram illustrates the conversion of an NFA to a DFA. The NFA has start states 1 and 3, with transitions on symbols a, b, and c. The DFA has a single start state A, with transitions on symbols a, b, and c. The unexplored state is marked as state C.
SubsetConstruction(N) – Example

1 \( \varepsilon\)-closure(1) = \{1, 2, 4\} = \text{A} \\
   \text{A} will be the DFA's start state.
Example...

\[ \varepsilon\text{-closure}(\text{move}(\{A\}, a)) = \varepsilon\text{-closure}(\text{move}(\{1, 2, 4\}, a)) = \varepsilon\text{-closure}(\{2, 3\}) = \{2, 3, 4\} = B \]

- We add the transition \(A \xrightarrow{a} B\)
Example...

\[ \epsilon\text{-closure}(\text{move}(A, b)) = \epsilon\text{-closure}(\text{move}([1, 2, 4], b)) = \epsilon\text{-closure}([4]) = [2, 4] = C \]

- We add the transition \( A \xrightarrow{b} C \)
\[ \epsilon - \text{closure}(\text{move}(B, b)) = \epsilon - \text{closure}(\text{move}([2, 3, 4], b)) = \epsilon - \text{closure}([4]) = [2, 4] = C \]

We add the transition \( B \xrightarrow{b} C \).
Example...

\[ \varepsilon\text{-closure}(\text{move}(C, b)) = \varepsilon\text{-closure}(\text{move}(\{2, 4\}, b)) = \varepsilon\text{-closure}(\{2, 4\}) = \{2, 4\} = C \]

- We add the transition \( C \xrightarrow{b} C \)
A slightly different approach is to generate the power-set of the set of NFA states, and then add all the edges we get from $\epsilon$-closure().
On $\epsilon$ we can go to states 1, 2, 4 which becomes our start state, A.
From states $1, 2, 4$ we can go to states $2, 3, 4$ on an $a$. 
Example, Take 2...

From states 1, 2, 4 we can go to states 2, 4 on a b.
Example, Take 2...

- From states 2, 3, 4 we can go to states 2, 4 on a b.
Example, Take 2...

- From states ②, ④ we can go to states ②, ④ on a b.
Finally, removing unreachable states gives us our DFA.
Keywords
Keywords revisited

- For a language with many keywords (Ada-95 has 98, COBOL has hundreds), the transition table can be large.

- We can remove all keywords from the transition table and instead analyze them as IDENTs.

- When an IDENT is found we look it up in a special table to see if it is, in fact, a reserved word.

- We can use a regular hash-table, of course, but if we’re concerned about speed we can use a minimal perfect hash-table. This is a static table and related lookup routines that have been optimized for a particular static set of words.
Keywords revisited...

For example, we could build this perfect hash-table for the words LUCA, MODULA-2, OBERON:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LUCA</td>
</tr>
<tr>
<td>1</td>
<td>MODULA-2</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>OBERON</td>
</tr>
</tbody>
</table>

```java
int hash(String s) { return s[0] - 'L'; }
boolean member(String s) { return table[hash(s)] = s; }
```

In this case we use the first character of the string as the hash-value.

This is not a **minimal** table, there’s one wasted entry.
Using Unix gperf

- gperf (http://www.gnu.org/manual/gperf-2.7) is a Unix program that takes a list of keywords as input and returns a perfect hash-table (and related search routines) as output.

- From the gperf manual:

  The perfect hash function generator gperf reads a set of "keywords" from a keyfile. It attempts to derive a perfect hashing function that recognizes a member of the static keyword set with at most a single probe into the lookup table. If gperf succeeds in generating such a function it produces a pair of C source code routines that perform hashing and table lookup recognition.
The following command

```
> echo "BEGIN\nEND" | gperf -L ANSI-C
```

generates the C program below.

```c
/* ANSI-C code produced by gperf version 2.7 */
#define TOTAL_KEYWORDS 2
#define MIN_WORD_LENGTH 3
#define MAX_WORD_LENGTH 5
#define MIN_HASH_VALUE 3
#define MAX_HASH_VALUE 5
```
static unsigned int hash (register const char *str, register unsigned int len) {
    static unsigned char asso_values[] = {
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        <--- Lots more stuff like this --->
    };
    return len + asso_values[(unsigned char)str[len - 1]] +
           asso_values[(unsigned char)str[0]];
}
const char * in_word_set (  
    register const char *str,  
    register unsigned int len) {  
    static const char * wordlist[] = {  
        "", "", "", "END", "", "BEGIN"};  
    
    if (len<=MAX_WORD_LENGTH && len>=MIN_WORD_LENGTH) {  
        register int key = hash (str, len);  
        if (key <= MAX_HASH_VALUE && key >= 0) {  
            register const char *s = wordlist[key];  
            if (*str == *s && !strcmp (str + 1, s + 1)) return  
        }  
    }  
    return 0;  
}

In this particular case, the hash function only looks at the first and last characters of the string, as well as the string length.
Summary
The problem with table-driven methods is that the tables can easily get huge. Much work has gone into constructing table-compression algorithms, and data structures for sparse tables. See the Dragon book for details.

There are also many algorithms for minimizing the number of states in a DFA. See Louden, pp. 72–74.
Read Louden, pp. 31–80.

Or, read the Dragon book, pp. 83–140.

An interview with Ken Thompson:


His Turing award lecture (*Reflections on Trusting Trust)*:

http://www.acm.org/classics/sep95/.

The next slide shows how you insert a Trojan Horse in the C compiler.
compile (String S)
   if (we’re compiling "login.c")
      GENERATE_CODE(
         if (user=="collberg" && passwd="D. Troi")
            login_ok = true
      )
   if (we’re compiling "gcc.c")
      GENERATE_CODE(
         if (we’re compiling "login.c")
            GENERATE_CODE(
               if (user=="collberg" && passwd="D. Troi")
                  login_ok = true
            )
      )