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

```plaintext
state := start state
c := first char
while (true) {
  case state of {
    1: case c of {
      char1 : {
        char1 := nextChar();
        state := new state;
      }
    2: case c of {
      char2 : {
        char2 := nextChar();
        state := new state;
      }
    3: case c of {
      char3 : {
        return; /* accept */
      }
    }
    if ACCEPT[state] then accept;
  }
  }
}
```
static boolean[] ACCEPT =
{false,false,false,false,true};

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

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.
Thompson’s Construction

- Each piece of a regular expression is turned into a part of an NFA.
- Each part is glued together (using ε-transitions) into a complete automaton.
- An RE matching the character a translates into

\[ \epsilon \]

- An RE matching ε translates into

\[ \epsilon \]

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:

```
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);
```

- 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 – Concatenation

- We represent an RE component r by the figure:

\[ \epsilon \]

- An RE matching the character a translates into

\[ \epsilon \]

- An RE matching the regular expression r followed by the regular expression s (rs) translates into

\[ \epsilon \]
Thompson’s Construction – Example I

- The regular expression \( ab|a \) translates into

\[
\begin{array}{c}
\text{a} \\
\varepsilon \\
a \\
\varepsilon \\
\end{array} \quad \begin{array}{c}
b \\
\varepsilon \\
b \\
\varepsilon \\
\end{array} \quad \begin{array}{c}
\varepsilon \\
\end{array}
\]

Thompson’s Construction – Alternation

- The regular expression \( r|s \) translates into

\[
\begin{array}{c}
\varepsilon \\
r \\
\varepsilon \\
\varepsilon \\
\end{array} \quad \begin{array}{c}
s \\
\varepsilon \\
s \\
\varepsilon \\
\end{array}
\]

Thompson’s Construction – Example II

- The regular expression \( letter(letter|digit)^* \) translates into

\[
\begin{array}{c}
\text{letter} \\
\varepsilon \\
\text{digit} \\
\varepsilon \\
\text{letter} \\
\varepsilon \\
\varepsilon \\
\end{array} \quad \begin{array}{c}
\varepsilon \\
\text{letter} \\
\varepsilon \\
\varepsilon \\
\varepsilon \\
\varepsilon \\
\end{array} \quad \begin{array}{c}
\text{digit} \\
\varepsilon \\
\varepsilon \\
\end{array}
\]

Thompson’s Construction – Repetition

- The regular expression \( r^* \) translates into

\[
\begin{array}{c}
\varepsilon \\
r \\
\varepsilon \\
\varepsilon \\
\end{array} \quad \begin{array}{c}
r \\
\varepsilon \\
\varepsilon \\
\end{array}
\]
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.

We need three functions:

1. $\epsilon$-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. 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. SubsetConstruction($N$) returns a DFA $D = (D\text{states}, D\text{trans})$ corresponding to NFA $N$. 

• $\mathbb{A}$ 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).

• $\mathbb{B}$ 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 $\mathbb{A}$.

From NFA to DFA...

From NFA to DFA...
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(\text{stack}, $u$)
    return $C$

move($T, a$) – Example

- move($\{1\}, a$) = $\{2, 3\}$
- move($\{2, 3\}, b$) = $\{4\}$

procedure SubsetConstruction(NFA $N$)

Dstates := $\{\epsilon$-closure($s_0$)$\}$
Dtrans := $\{\}$
repeat
    $T := \text{an unexplored state in Dstates}$
    for each input symbol $a$ do
        $U := \epsilon$-closure(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)

$\epsilon$-closure($T$) – Example

- $\epsilon$-closure($1$) = $\{1, 2, 4\}$
- $\epsilon$-closure($2$) = $\{2\}$
- $\epsilon$-closure($3$) = $\{2, 4\}$
- $\epsilon$-closure($\{3, 4\}$) = $\{2, 3, 4\}$
2. $\epsilon$-closure(move(A, a)) =
$\epsilon$-closure(move({1, 2, 4}, a)) =
$\epsilon$-closure({2, 3}) = {2, 3, 4} = B
- We add the transition $A \xrightarrow{a} B$

3. $\epsilon$-closure(move(A, b)) =
$\epsilon$-closure(move({1, 2, 4}, b)) = $\epsilon$-closure({4}) = {2, 4} = C
- We add the transition $A \xrightarrow{b} C$

1. $\epsilon$-closure(1) = {1, 2, 4} = A
- A will be the DFA’s start state.
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.

- We add the transition A $\rightarrow$ C

$\epsilon$-closure(move(A, $b$)) =
$\epsilon$-closure(move({2, 3, 4}, $b$)) = $\epsilon$-closure({4}) =
{2, 4} = C

- We add the transition A $\rightarrow$ C

$\epsilon$-closure(move(C, $b$)) =
$\epsilon$-closure(move({2, 4}, $b$)) = $\epsilon$-closure({2, 4}) =
{2, 4} = C

- We add the transition C $\rightarrow$ C
Example, Take 2...

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

Example, Take 2...

From states 1, 2, 4 we can go to states 2, 3, 4 on an a.

Example, Take 2...

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

Example, Take 2...

From states 1, 2, 4 we can go to states 2, 4 on a b.
Keywords revisited...

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

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

```
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.

Example, Take 2...

- Finally, removing unreachable states gives us our DFA.

```
1 —> 2
2 —> 3 —> 4
```

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.
Using Unix gperf...

- 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,
    }
    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 s;
        }
    }
    return 0;
    }
```

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.
Readings and References

- Read Louden, pp. 31–80.
- Or, read the Dragon book, pp. 83–140.
- 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.

```java
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
                )
        )
```