Implementing Automata

- NFAs and DFAs can be hard-coded using the pattern

```java
state := start state
c := first char
while (true) {
    case state of {
        1: case c of {
            char1 := nextChar();
            state := new state;
        }
        2: case c of {
            char2 := nextChar();
            state := new state;
        }
        char3 := {
            return; /* accept */
        }
    }
}
```

Slide 13–1

Implementing Automata...

- We can also encode the transitions directly into a transition table.

```
<table>
<thead>
<tr>
<th>state</th>
<th>char1</th>
<th>char2</th>
<th>other</th>
<th>Accepting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>[3]</td>
</tr>
</tbody>
</table>
```

- States in brackets don’t consume their inputs. Accepting states are indicated by √. Empty entries represent error states.

Slide 13–2

Implementing Automata...

- Given the table, we can write an interpreter to perform lexical analysis of any DFA:

```java
state := 1
c := first char
while not ACCEPT[state, c] do {
    nextstate := nextState[state, c]
    state := nextstate
    c := nextChar()
}
if state = ACCEPT then accept;
```

Slide 13–3
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;
}

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}
    };
}

Table-driven C Comments

Table-driven C Comments...
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 :
                    return (state>=0) && ACCEPT[state];
            }
        }
    }

    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();
    }
}

Hard-coded C Comments

- 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

  \[ a \]

- An RE matching ε translates into

  \[ \epsilon \]

Thompson’s Construction – Concatenation

- We represent an RE component r by the figure:

  ![Concatenation Diagram]

  - Start state for r
  - Accepting state for r

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

  \[ \epsilon \]

  \[ r \]

  \[ s \]

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:

  ![From REs to NFAs Diagram]

- 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 – Alternation

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

Slide 13–16

Thompson’s Construction – Repetition

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

Slide 13–17

Thompson’s Construction – Example I

- The regular expression \( abla \) translates into

Slide 13–18

Thompson’s Construction – Example II

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

Slide 13–19
From NFA to DFA...

We need three functions:

1. \( \epsilon\text{-}closure(T) \) 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. \( \text{move}(T,a) \) is the set of NFA states to which there is a transition on input symbol \( a \) from some NFA state \( s \in T \).

3. \( \text{SubsetConstruction}(N) \) returns a DFA \( D=(\text{Dstates,Dtrans}) \) corresponding to NFA \( N \).

\( \epsilon\text{-}closure(T) \)

Procedure \( \epsilon\text{-}closure(T) \)

1. Push all states in \( T \) onto stack \( C := T \)
2. While stack is not empty do
   1. \( t := \text{pop(stack)} \)
   2. For each edge \( t \xrightarrow{\epsilon} u \) do
      1. If \( u \) is not in \( C \) then
         1. \( C := C \cup u \)
         2. Push(\text{stack, } u) \)
3. Return \( C \)

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.
- 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 \( a_1 a_2 \cdots a_n \) the DFA is in a state that represents the states the NFA could be in after seeing the input \( a_1 a_2 \cdots a_n \).

From NFA to DFA...

- 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).
- 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 \( 1 \).
\begin{enumerate}
\item $\varepsilon$-closure(\{1\}) = \{1,2,4\}
\item $\varepsilon$-closure(\{2\}) = \{2\}
\item $\varepsilon$-closure(\{4\}) = \{2,3,4\}
\item $\varepsilon$-closure([2,4]) = \{2,3,4\}
\end{enumerate}

\begin{enumerate}
\item move(\{1\},a) = \{2,3\}
\item move(\{2\},b) = \{4\}
\item move([2,3],a) = \{3\}
\end{enumerate}

\bigskip

\textbf{SubsetConstruction(\textit{NFA})}

\begin{align*}
\text{procedure \textit{SubsetConstruction}(NFA \textit{N})} \\
\text{Dstates} := \{\varepsilon\text{-closure}(s_0)\} \\
\text{Dtrans} := \{\} \\
\text{\textbf{repeat}} \\
\text{\hspace{1em}T := \{\text{an unexplored state in Dstates}\}} \\
\text{\hspace{1em}U := \varepsilon\text{-closure(\textit{move}(T,a))}} \\
\text{\hspace{1em}if } U \text{ \is not in Dstates \textbf{then}} \\
\text{\hspace{2em}Dstates := Dstates \cup Dtrans \cup \{T \rightarrow U\}} \\
\text{\hspace{1em}Dtrans := \{Dtrans \cup \{T \rightarrow U\}\}} \\
\text{\textbf{until all states have been explored}} \\
\text{\textbf{return}} (\text{Dstates, Dtrans})
\end{align*}

\bigskip

1. $\varepsilon$-closure(\{4\}) = \{1,2,4\} = A
2. $\varepsilon$-closure(\{\{2,4\}\}) = \varepsilon$-closure(\{2,3,4\}) = B

\bigskip

\textbf{Slide 13–24}

\textbf{Slide 13–25}

\textbf{Slide 13–26}

\textbf{Slide 13–27}

\bigskip

\textbf{We add the transition} A \overset{a}{\rightarrow} B
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:

<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.

SubsetConstruction(N) - Example...

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

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

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

The resulting DFA:
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.

Using Unix gperf...

- The following command

```bash
> echo "BEGIN\n\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
```
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.