**State Transition Diagrams**

- Finite directed graph
- Edges (transitions) labeled with symbols from an alphabet
- Nodes (states) labeled only for convenience
- One initial state
- Several accepting states

A string $c_1c_2c_3\ldots c_n$ is accepted by a state transition diagram if there exists a path from the starting state to an accepting state such that the sequence of labels along the path is $c_1, c_2, \ldots, c_n$

- Same state might be visited more than once
- Intermediate states might be final

The set of exactly all the strings accepted by a state transition diagram is the language accepted (or recognized) by the state transition diagram

**Deterministic Finite Automata**

A state diagram describes graphically a deterministic finite automaton (DFA), a machine that at any given time is in one of finitely many states, and whose state changes according to a predetermined way in response to a sequence of input symbols

Formal definition: a deterministic finite automaton is a tuple $M = (K, \Sigma, \delta, s, F)$

- $K$ ⇒ finite set of states
- $\Sigma$ ⇒ input alphabet
- $F \subseteq K$ ⇒ set of accepting states
- $s \in K$ ⇒ initial state
- $\delta : K \times \Sigma \rightarrow K$ ⇒ transition function

$$
\begin{align*}
K &= \{q_1, q_2, q_3, q_4\} \\
\delta(q_1, 0) &= q_1 \\
\delta(q_1, 1) &= q_2 \\
\delta(q_2, 0) &= q_3 \\
\delta(q_2, 1) &= q_4 \\
\delta(q_3, 0) &= q_4 \\
\delta(q_3, 1) &= q_1 \\
\delta(q_4, 0) &= q_4 \\
\delta(q_4, 1) &= q_3
\end{align*}
$$

**Software Realization**

Big practical advantages of DFA: very easy to implement:

- Interface to define a vocabulary and a function to obtain the input tokens
  ```cpp
typename vocab; /* alphabet + end-of-string */
const vocab EOS; /* end-of-string pseudo-token */
vocab gettoken(void); /* returns next token */
```

- Variable (state) changed by a simple switch statement as we go along

```cpp
int main(void) {
typedef enum {S0, S1, ... } state;
state s = S0; vocab t = gettoken();
while ( t != EOS ) {
    switch (s) {
        case S0: if (t == ...) s = ...; break;
        case S1: ...}
    t = gettoken();
} /* while */
/* accept iff the current state s is final */
```
SOFTWARE REALIZATION: EXAMPLE

typedef enum {ZERO, ONE, EOS} vocab;

vocab gettoken(void) {
    int c = getc(stdin);
    if (c == '0') return ZERO;
    if (c == '1') return ONE;
    if (c == '
') return EOS;
    perror("illegal character");
}

int main (void) {
    typedef enum {S0, S1 } state;
    state s = S0; vocab t = gettoken();
    while ( t != EOS ) {
        switch (s) {
            case S0: if (t == ONE) s = S1; break;
            /* if (t == ZERO) s = S0; break */
            case S1: if (t == ONE) s = S0; break;
            /* if (t == ZERO) s = S1; break */
            } /* switch */
        t = gettoken(); } /* while */
    if (s != S0) printf("String not accepted.
");
}

NONDETERMINISM

So far the state diagrams are deterministic = for any pair (state, input
symbol) there can be at most one outgoing transition

A nondeterministic diagram allows for the following situation:

- The acceptance condition remains unchanged:
  - A string $c_1c_2c_3\ldots c_n$ is accepted by a state transition diagram if there exists some path from the starting state to an accepting state such that the sequence of labels along the path is $c_1, c_2, \ldots, c_n$

- Why nondeterminism?
  - Simplifies the construction of the diagram
  
- A nondeterministic diagram can be much smaller than the smallest possible deterministic state diagram that recognizes the same language

- Also known as nondeterministic finite automata (NFA)

SOFTWARE REALIZATION (CONT’D)

As above, except that we have to keep track of a set of states at any
given time
typedef enum { Q0, Q1, Q2, Q3 } state;

int main (void) {
    typedef enum {S0, S1 } state;
    state s = S0; vocab t = gettoken();
    while ( t != EOS ) {
        switch (s) {
            case Q0: if (t == ONE) s = S1; break;
            /* if (t == ZERO) s = S0; break */
            case Q1: if (t == ONE) s = S0; break;
            /* if (t == ZERO) s = S1; break */
            } /* switch */
        t = gettoken(); } /* while */
    if (t != EOS) printf("String not accepted.
");
}

SOFTWARE REALIZATION

This kind of implementation is fine for “throw-away” automata
- Text editor search function searches for a pattern in the text
- The next search is likely to be different so a brand new automaton needs to be created

Some times the automaton is created once and then used multiple times
- The lexical structure of a programming language is well established
- Lexical analysis in a compiler is accomplished by an automaton that never changes
- In such a case it is more efficient to precalculate the set of states
  - Exactly as in the previous program
  - Except that we no longer have an input to guide us, so we calculate the sets NewA for all possible inputs
  - We obtain a DFA that is equivalent to the given NFA (i.e., recognizes the same language)
- **ε-TRANSITIONS**
  - Useful at times to have “spontaneous” transitions = transitions that change the state without any input being read = ε-transitions
  - Only available for nondeterministic state transition diagrams!
  - Example of usefulness: Construct the state transition diagram for the language
    \[ \{0, 1\}^*01\{0, 1\}^* + \{w \in \{0, 1\}^*: w \text{ has an even number of 1's}\} \]
  - Even better ε-transitions can be eliminated afterward

---

**ELIMINATING ε-TRANSITIONS**

For every diagram \( M \) with ε-transitions a new diagram without ε-transitions can be constructed as follows:

- Make a copy \( M' \) of \( M \) where the ε-transitions have been removed.
  - Remove states that have only ε-transitions coming in except for the starting state
- Add transitions to \( M' \) as follows: whenever \( M \) has a chain of ε-transitions followed by a “real” transition on \( x \):
  
  \[
  \text{add to } M' \text{ a transition from state } q \text{ to state } p \text{ labeled by } x:
  \]

\[
\begin{align*}
  &\text{@} \\
  &\quad \longrightarrow \\
  &\quad \quad \cdots \\
  &\quad \longrightarrow \\
  &\quad \longrightarrow \\
  &\quad \longrightarrow \\
\end{align*}
\]

- Note that \( q \) and \( p \) may be any states
  - In particular this step is also used in the case where \( q = p \)
- If \( M \) has a chain of ε-transitions from a state \( r \) to an accepting state, then \( r \) is made to be an accepting state of \( M' \).