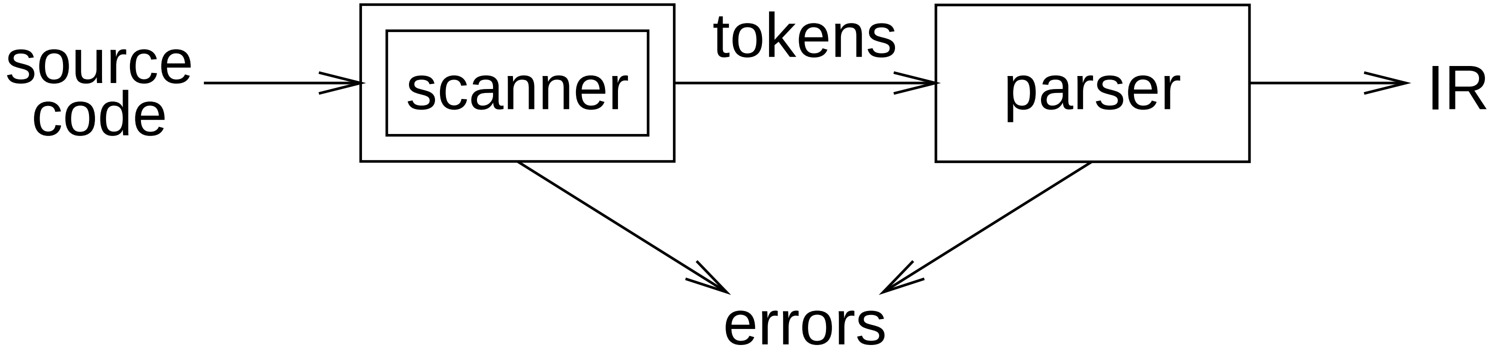
Scanner



- maps characters into *tokens* – the basic unit of syntax

  `x = x + y;`

  becomes

  <id, `x`> `=` <id, `x`> `+` <id, `y`> `;`
- character string value for a *token* is a *lexeme*
- typical tokens: *number*, *id*, `+`, `-`, `*`, `/`, `do`, `end`
- eliminates white space (*tabs, blanks, comments*)
- a key issue is speed

  $\Rightarrow$ use specialized recognizer (as opposed to `lex`)

Specifying patterns

A scanner must recognize the units of syntax

Some parts are easy:

white space

$$
\begin{array}{lcl}
\langle \mathrm{ws} \rangle & ::= & \langle \mathrm{ws} \rangle \ \texttt{' '} \\
 & | & \langle \mathrm{ws} \rangle \ \texttt{'\textbackslash t'} \\
 & | & \texttt{' '} \\
 & | & \texttt{'\textbackslash t'}
\end{array}
$$

keywords and operators

specified as literal patterns: `do`, `end`

comments

opening and closing delimiters: `/* ··· */`

Specifying patterns

A scanner must recognize the units of syntax

Other parts are much harder:

identifiers

alphabetic followed by k alphanumerics (_, $, &, ...)

numbers

integers: 0 or digit from 1-9 followed by digits from 0-9

decimals: integer `'.'` digits from 0-9

reals: (integer or decimal) `'E'` (+ or -) digits from 0-9

complex: `'('` real `','` real `')'`

We need a powerful notation to specify these patterns

Operations on languages

Operation	Definition
union of L and M written $L \cup M$	$L \cup M = \{s \mid s \in L$ or $s \in M\}$
concatenation of L and M written *LM*	$LM = \{st \mid s \in L$ and $t \in M\}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Regular expressions

Patterns are often specified as *regular languages*

Notations used to describe a regular language (or a regular set) include both *regular expressions* and *regular grammars*

Regular expressions (*over an alphabet* Σ):

1. ε is a RE denoting the set $\{\varepsilon\}$
2. if $a \in \Sigma$, then a is a RE denoting $\{a\}$
3. if r and s are REs, denoting $L(r)$ and $L(s)$, then:

   (r) is a RE denoting $L(r)$

   $(r) \mid (s)$ is a RE denoting $L(r) \bigcup L(s)$

   $(r)(s)$ is a RE denoting $L(r)L(s)$

   $(r)^*$ is a RE denoting $L(r)^*$

If we adopt a *precedence* for operators, the extra parentheses can go away. We assume *closure*, then *concatenation*, then *alternation* as the order of precedence.

Examples

identifier

$\textit{letter} \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$

$\textit{digit} \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$

$\textit{id} \rightarrow \textit{letter}\ (\ \textit{letter} \mid \textit{digit}\ )^*$

numbers

$\textit{integer} \rightarrow (+ \mid - \mid \varepsilon)\ (0 \mid (1 \mid 2 \mid 3 \mid \ldots \mid 9)\ \textit{digit}^*)$

$\textit{decimal} \rightarrow \textit{integer}\ \texttt{.}\ (\ \textit{digit}\ )^*$

$\textit{real} \rightarrow (\ \textit{integer} \mid \textit{decimal}\ )\ \texttt{E}\ (+ \mid -)\ \textit{digit}^*$

$\textit{complex} \rightarrow \texttt{'('}\ \textit{real}\ \texttt{,}\ \textit{real}\ \texttt{')'}$

Numbers can get much more complicated

Most programming language tokens can be described with REs

We can use REs to build scanners automatically

Algebraic properties of REs

Axiom	Description
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	concatenation distributes over $\|$
$\varepsilon r = r$ $r\varepsilon = r$	ε is the identity for concatenation
$r^* = (r\|\varepsilon)^*$	relation between $*$ and ε
$r^{**} = r^*$	$*$ is idempotent

Examples

Let $\Sigma = \{a, b\}$

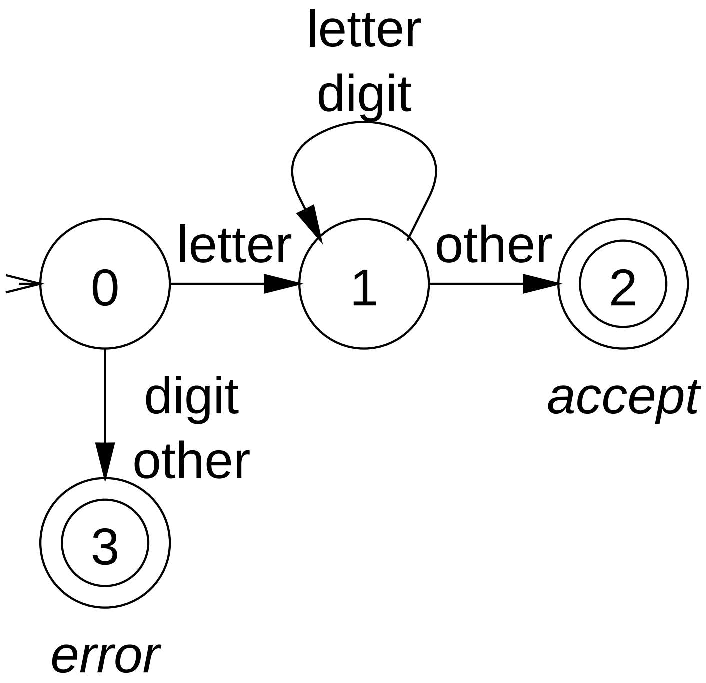
1. $a|b$ denotes $\{a, b\}$
2. $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$
   i.e., $(a|b)(a|b) = aa|ab|ba|bb$
3. a^* denotes $\{\varepsilon, a, aa, aaa, \ldots\}$
4. $(a|b)^*$ denotes the set of all strings of a's and b's (including ε)
   i.e., $(a|b)^* = (a^*b^*)^*$
5. $a|a^*b$ denotes $\{a, b, ab, aab, aaab, aaaab, \ldots\}$

Recognizers

From a regular expression we can construct a

deterministic finite automaton (DFA)

Recognizer for *identifier*:



identifier

letter $\rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$

digit $\rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$

id $\rightarrow$ *letter* (*letter* $\mid$ *digit* $)^*$

Code for the recognizer

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""  /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:     /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:     /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:     /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Tables for the recognizer

Two tables control the recognizer

`char_class`:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

`next_state`:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

To change languages, we can just change tables

Automatic construction

Scanner generators automatically construct code from RE-like descriptions

- construct a DFA
- use state minimization techniques
- emit code for the scanner
  (table driven or direct code)

A key issue in automation is an interface to the parser

`lex` is a scanner generator supplied with UNIX

- emits C code for scanner
- provides macro definitions for each token
  (used in the parser)

Grammars for regular languages

Can we place a restriction on the *form* of a grammar to ensure that it describes a regular language?

Provable fact:

> For any RE r, $\exists$ a grammar g such that $L(r) = L(g)$

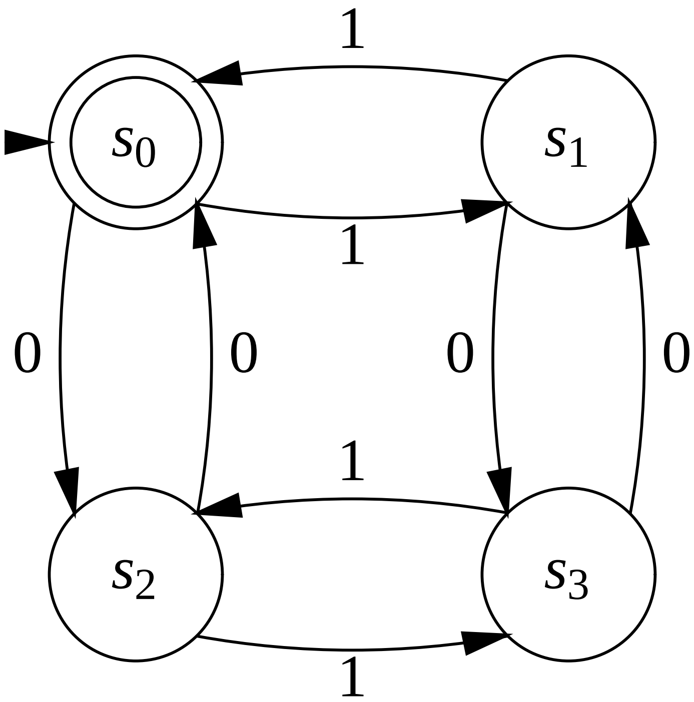
Grammars that generate regular sets are called *regular grammars*:

> They have productions in one of 2 forms:
>
> 1. $A \rightarrow aA$
> 2. $A \rightarrow a$
>
> where A is any non-terminal and a is any terminal symbol

These are also called *type 3* grammars (Chomsky)

More regular languages

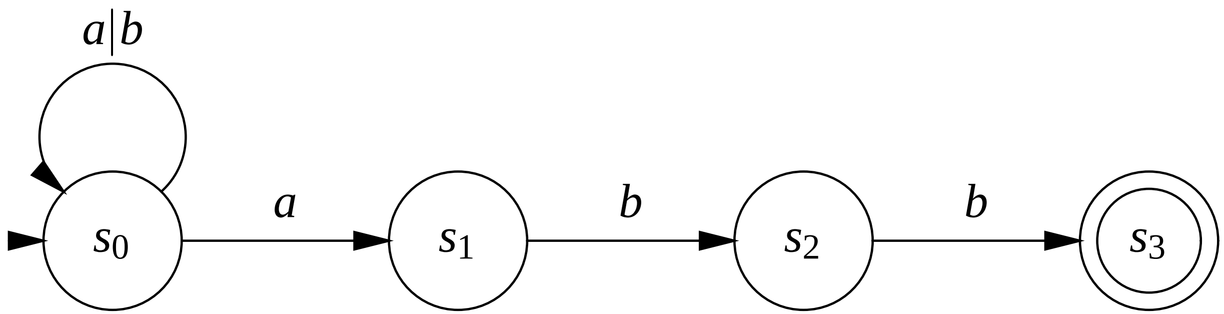
Example: the set of strings containing an even number of zeros and an even number of ones



The RE is $(00 \mid 11)^*((01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*)^*$

More regular expressions

What about the RE $(a \mid b)^*abb$?



State s_0 has multiple transitions on a!

$\Rightarrow$ *nondeterministic finite automaton*

	a	b
s_0	$\{s_0, s_1\}$	$\{s_0\}$
s_1	–	$\{s_2\}$
s_2	–	$\{s_3\}$

Finite automata

A *non-deterministic finite automaton* (NFA) consists of:

1. a set of *states* $S = \{s_0, \ldots, s_n\}$
2. a set of input symbols Σ (the alphabet)
3. a transition function *move* mapping state-symbol pairs to sets of states
4. a distinguished *start state* s_0
5. a set of distinguished *accepting* or *final* states F

A *Deterministic Finite Automaton* (DFA) is a special case of an NFA:

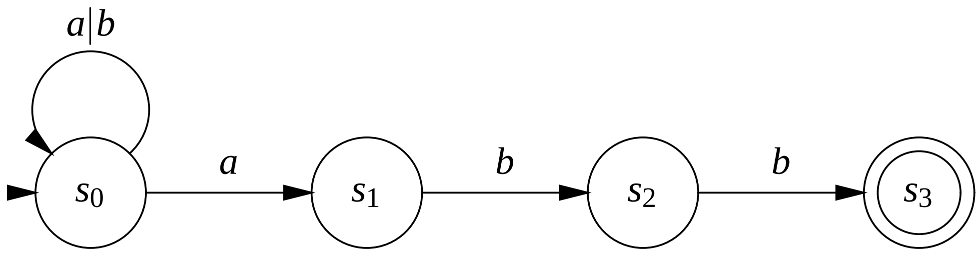
1. no state has a ε-transition, and
2. for each state s and input symbol a, there is at most one edge labelled a leaving s

A DFA *accepts* x iff. $\exists$ a *unique* path through the transition graph from s_0 to a final state such that the edges spell x.

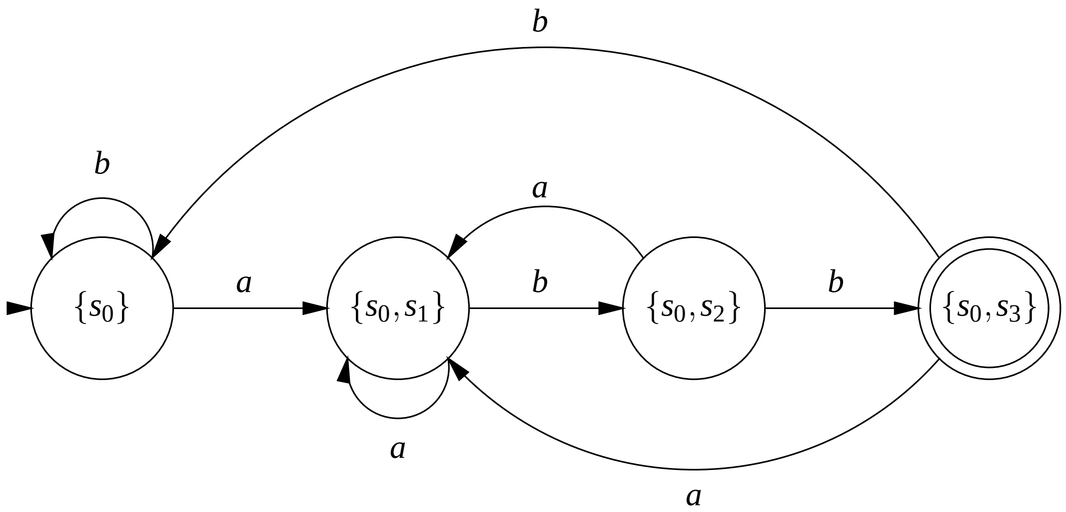
DFAs and NFAs are equivalent

1. DFAs are clearly a subset of NFAs
2. Any NFA can be converted into a DFA, by simulating sets of simultaneous states:
   - each DFA state corresponds to a set of NFA states
   - possible exponential blowup

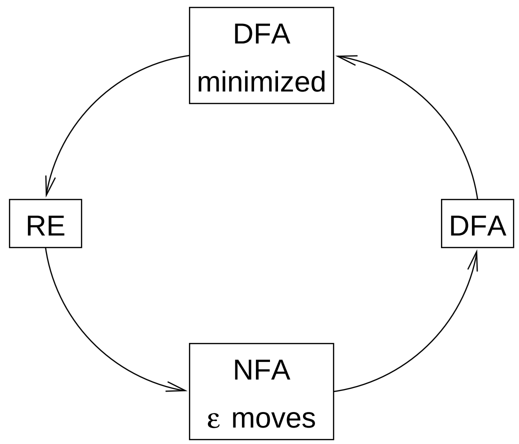
NFA to DFA using the subset construction: example 1



	a	b
$\{s_0\}$	$\{s_0,s_1\}$	$\{s_0\}$
$\{s_0,s_1\}$	$\{s_0,s_1\}$	$\{s_0,s_2\}$
$\{s_0,s_2\}$	$\{s_0,s_1\}$	$\{s_0,s_3\}$
$\{s_0,s_3\}$	$\{s_0,s_1\}$	$\{s_0\}$



Constructing a DFA from a regular expression



RE →NFA w/ε moves
- build NFA for each term
- connect them with ε moves

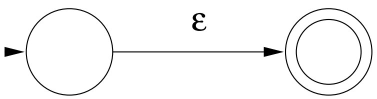
NFA w/ε moves to DFA
- construct the simulation
- the "subset" construction

DFA → minimized DFA
- merge compatible states

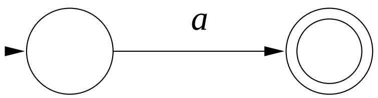
DFA → RE
- construct $R_{ij}^{k} = R_{ik}^{k-1}(R_{kk}^{k-1})^{*}R_{kj}^{k-1} \bigcup R_{ij}^{k-1}$

RE to NFA

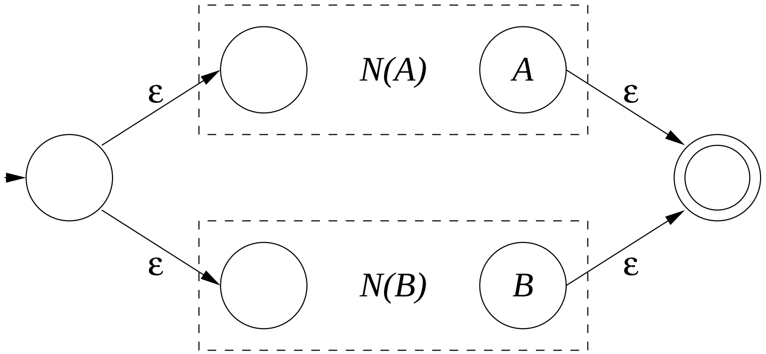
$N(\varepsilon)$



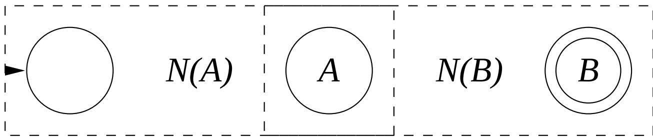
$N(a)$



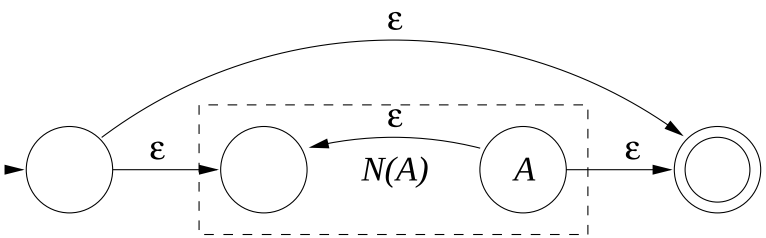
$N(A|B)$



$N(AB)$

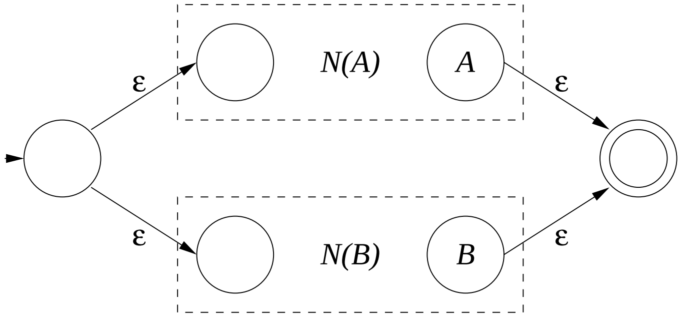


$N(A^*)$

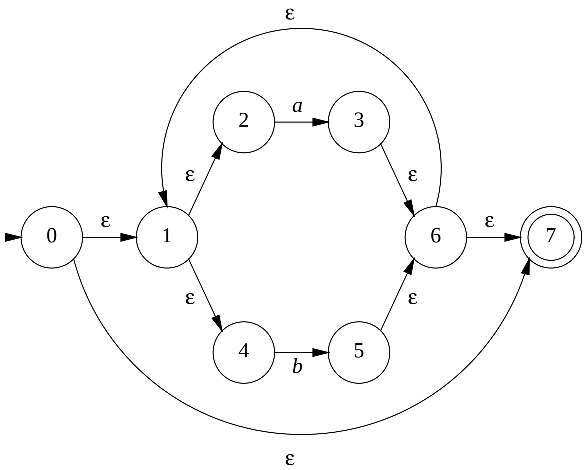


RE to NFA: example

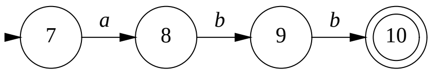
$a|b$



$(a|b)^*$



abb



NFA to DFA: the subset construction

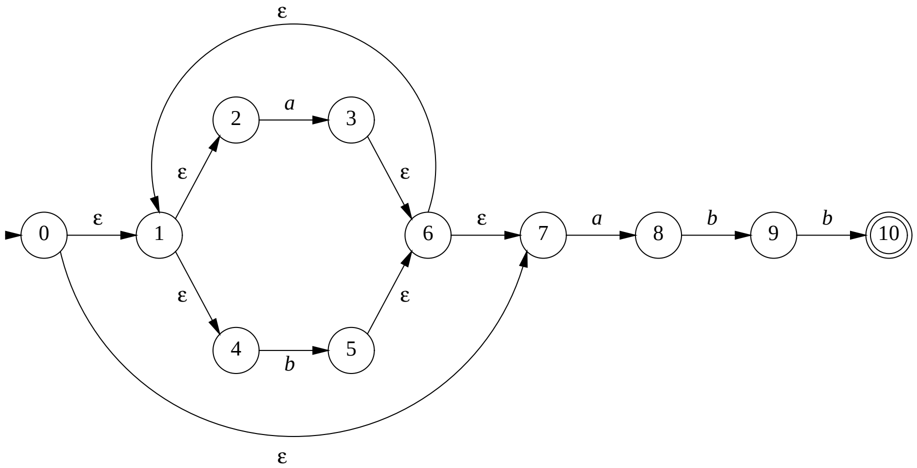
Input:	NFA N
Output:	A DFA D with states *Dstates* and transitions *Dtrans* such that $L(D) = L(N)$
Method:	Let s be a state in N and T be a set of states, and using the following operations:

Operation	Definition
ε-*closure*(s)	set of NFA states reachable from NFA state s on ε-transitions alone
ε-*closure*(T)	set of NFA states reachable from some NFA state s in T on ε-transitions alone
move(T, a)	set of NFA states to which there is a transition on input symbol a from some NFA state s in T

add state $T = \varepsilon$-*closure*(s_0) unmarked to *Dstates*
while $\exists$ unmarked state T in *Dstates*
    mark T
    for each input symbol a
        $U = \varepsilon$-*closure*(*move*(T, a))
        if $U \notin$ *Dstates* **then** add U to *Dstates* unmarked
        Dtrans$[T, a] = U$
    endfor
endwhile

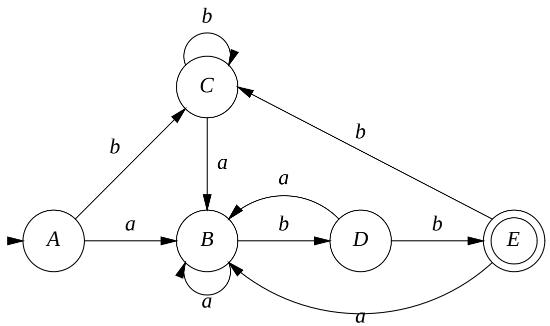
ε-*closure*(s_0) is the start state of D
A state of D is final if it contains at least one final state in N

NFA to DFA using subset construction: example 2



$$
\begin{aligned}
A &= \{0,1,2,4,7\} & D &= \{1,2,4,5,6,7,9\} \\
B &= \{1,2,3,4,6,7,8\} & E &= \{1,2,4,5,6,7,10\} \\
C &= \{1,2,4,5,6,7\}
\end{aligned}
$$

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



Limits of regular languages

Not all languages are regular

One cannot construct DFAs to recognize these languages:

- $L = \{p^k q^k\}$
- $L = \{wcw^r \mid w \in \Sigma^*\}$

Note: neither of these is a regular expression!
(DFAs cannot count!)

But, this is a little subtle. One can construct DFAs for:

- alternating 0's and 1's
  $(\varepsilon \mid 1)(01)^*(\varepsilon \mid 0)$
- sets of pairs of 0's and 1's
  $(01 \mid 10)^+$

Ramification - Internet Protocol

How does your browser establish a connection with a web server?

- The client sends a SYN message to the server.
- In response, the server replies with a SYN-ACK.
- Finally the client sends an ACK back to the server.

This is done through two DFAs in the client and server, respectively.

Ramification - Intrusion Detection

Code

```
FILE * f;
f=fopen("demo", "r");          ------>   SYS_OPEN
strcpy(...); //vulnerability
if (!f)
   printf("Fail to open\n");   ------>   SYS_WRITE
else
   fgets(f, buf);              ------>   SYS_READ
...
```

Operating System

```
SYS_OPEN
SYS_WRITE
SYS_READ
```

A DFA will be exercised simultaneously with the program on the OS side to detect intrusion.