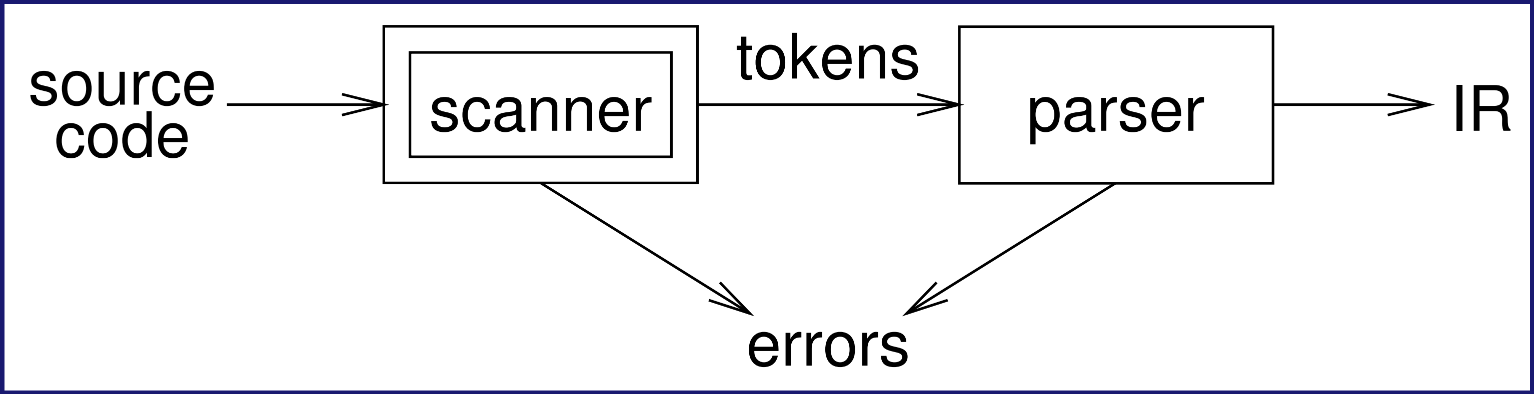
Scanner



- 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

  ⇒ use specialized recognizer (as opposed to `lex`)

Copyright ©2012 by Antony L. Hosking. *Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from hosking@cs.purdue.edu.*

Specifying patterns

A scanner must recognize the units of syntax

Some parts are easy:

white space

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

keywords and operators

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

comments

opening and closing delimiters: `/*` $\cdots$ `*/`

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

Regular expressions

Patterns are often specified as *regular languages*

Notations used to describe a regular language include both *regular expressions* and *regular grammars*

Regular expressions $RE(\Sigma)$ (*over an alphabet* Σ):

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

   $(r \mid s) \in RE(\Sigma)$ denoting $L(r) \bigcup L(s)$

   $(rs) \in RE(\Sigma)$ denoting $L(r)L(s)$

   $(r^*) \in RE(\Sigma)$ 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.

Reminder: Operations on languages

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

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 $\{a, b, ab, aab, aaab, aaaab, \ldots\}$

Algebraic properties of REs

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

Sometimes $\emptyset$ is also considered an RE

$r \mid \emptyset = \emptyset \mid r = r$	$\emptyset$ is unit for $\mid$
$r\emptyset = \emptyset r = \emptyset$	$\emptyset$ is zero for $\cdot$
$\emptyset^* = \varepsilon$	relation between * and $\emptyset$

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

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

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

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

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

Numbers can get much more complicated

Most tokens in programming languages can be described with REs

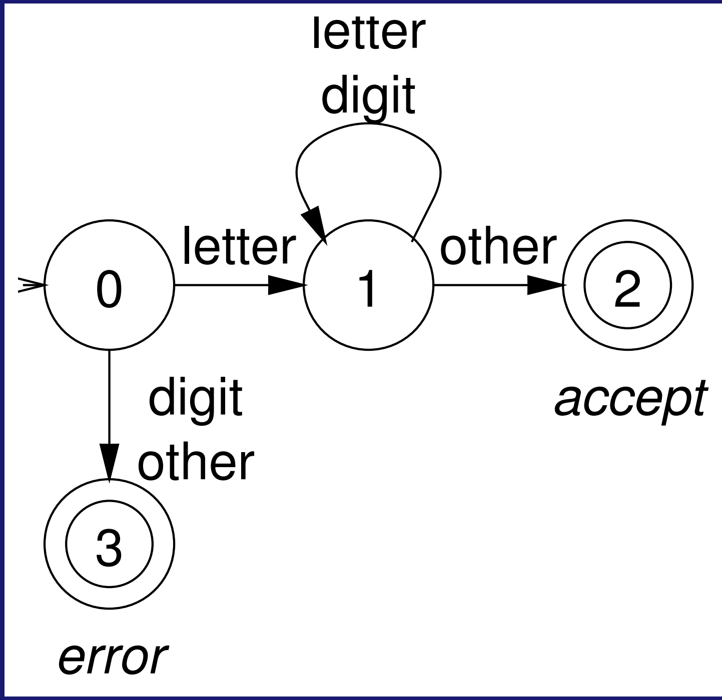
We can use REs to build scanners automatically

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* | *digit*)*

Code for the recognizer

```
char ← next_char();
state ← INITIAL_STATE;                /* 0 */
done ← false;
token_value ← "";          /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case BUILDING_ID_STATE:        /* 1 */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case ACCEPT_STATE:
         token_type ← IDENTIFIER;   /* 2 */
         done ← true;
         break;
      case ERROR_STATE:              /* 3 */
         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

From recognizers to scanners

1. Scanner runs multiple recognizers is parallel, one for each class of lexemes
2. Scanner reads input until all recognizers indicate error or accept (one character lookahead)
3. The recognizer, which accepts the longest prefix of the input, wins
   Principle of the longest match
4. If two recognizers accept the same prefix, then position in the specification breaks the tie

Consider the keyword `do`:

- `door_closing` — identifier (longer than prefix `do`)
- `do` — keyword (same length, `do` must precede identifier specification)

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 $L(r) = L(g)$

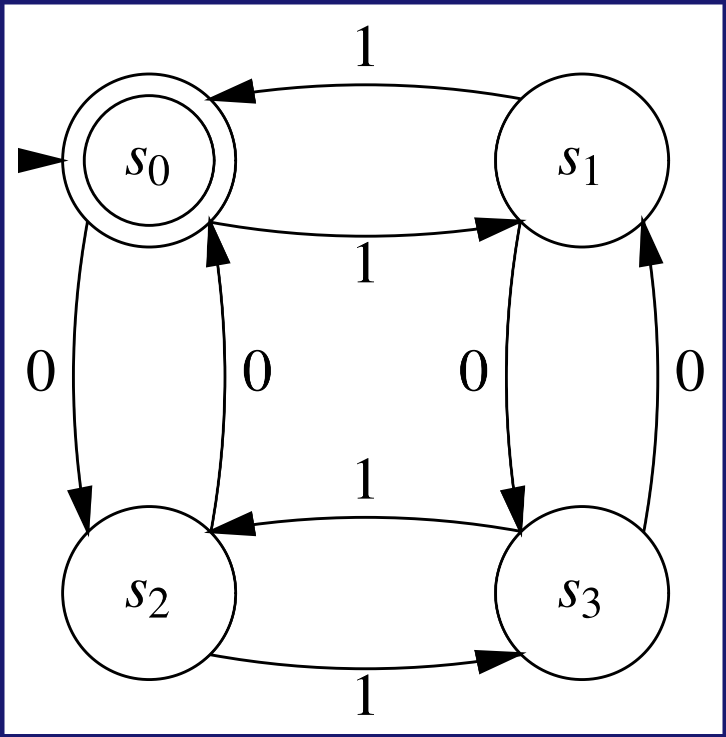
Grammars that generate regular sets are called *regular grammars*:

> 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

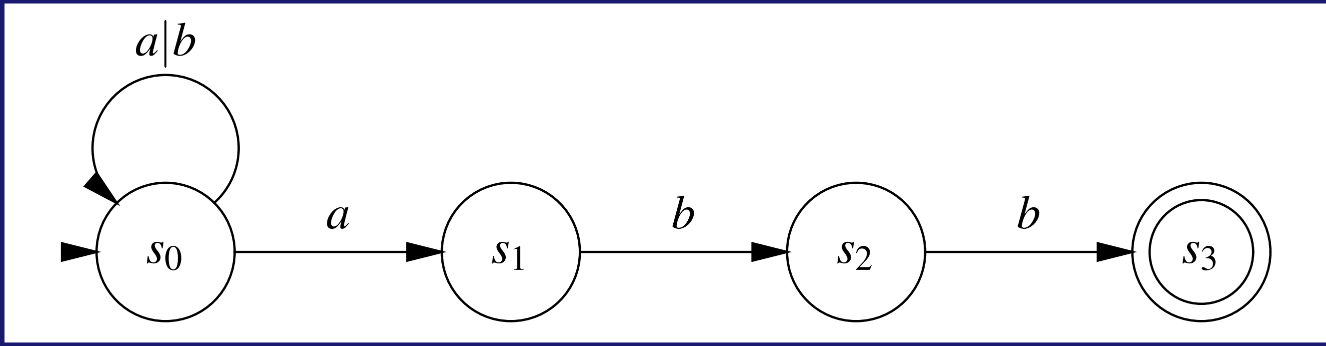
Example: the set of strings containing an even number of zeros and an even number of ones



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:

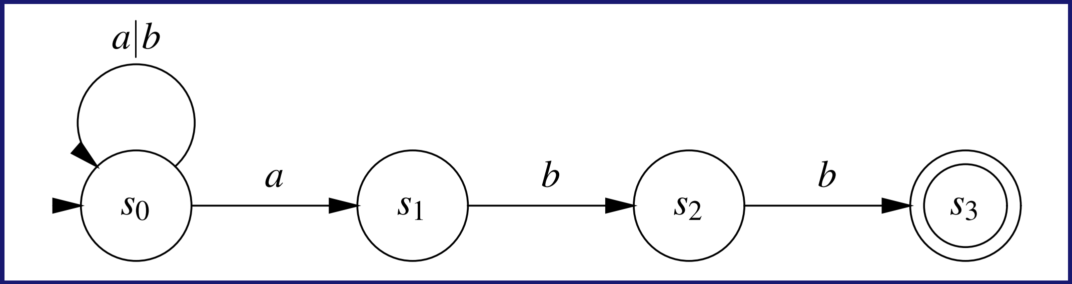
1. no state has a ε-transition, and
2. for each state s and input symbol a, there is at most one edge labelled a leaving s

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.

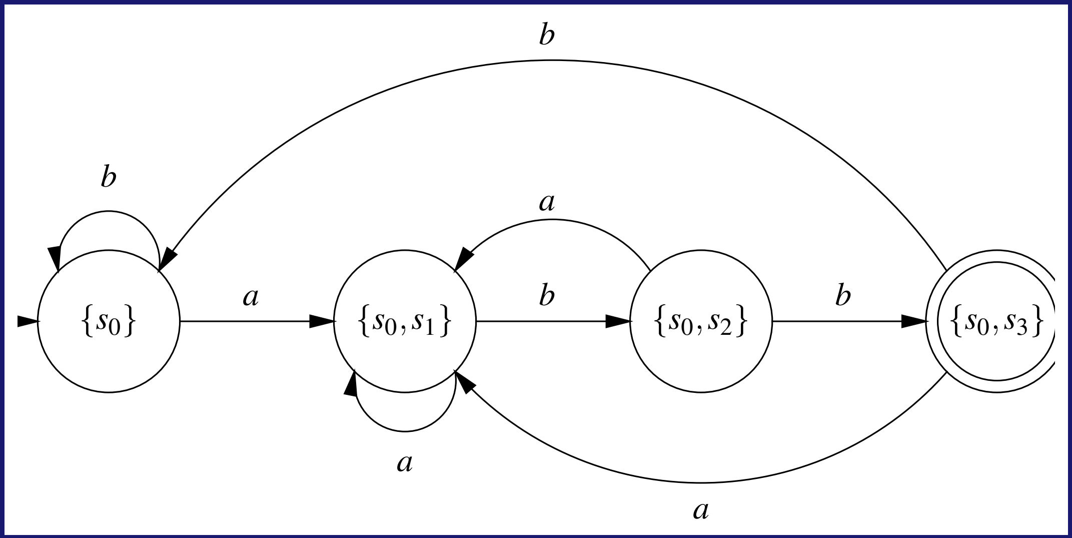
DFAs and NFAs are equivalent

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

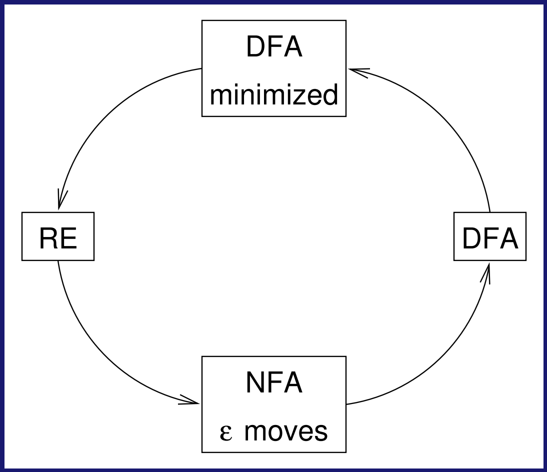
NFA to DFA using the subset construction: example 1



	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 $\rightarrow$NFA w/ε moves
build NFA for each term
connect them with ε moves

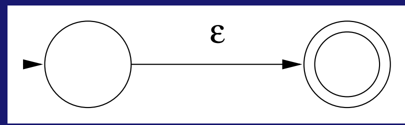
NFA w/ε moves to DFA
construct the simulation
the "subset" construction

DFA $\rightarrow$ minimized DFA
merge compatible states

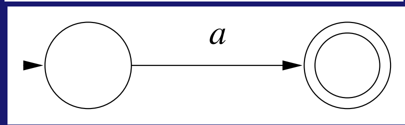
DFA $\rightarrow$ RE
construct $R^k_{ij} = R^{k-1}_{ik}(R^{k-1}_{kk})^* R^{k-1}_{kj} \bigcup R^{k-1}_{ij}$

RE to NFA

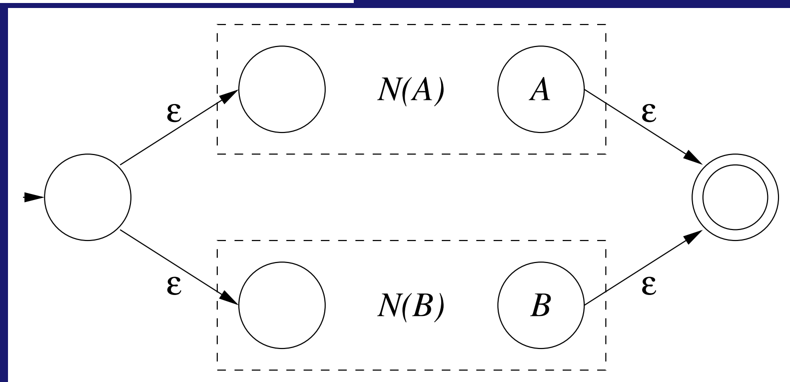
$N(\varepsilon)$



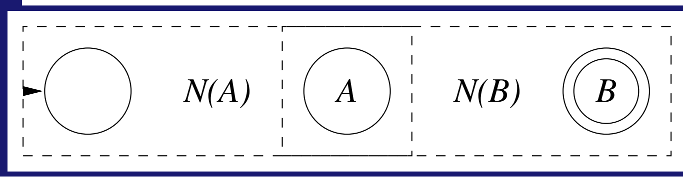
$N(a)$



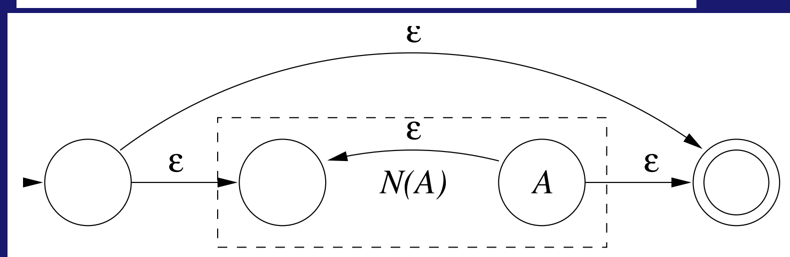
$N(A|B)$



$N(AB)$

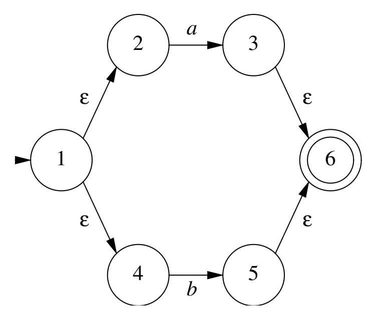


$N(A^*)$

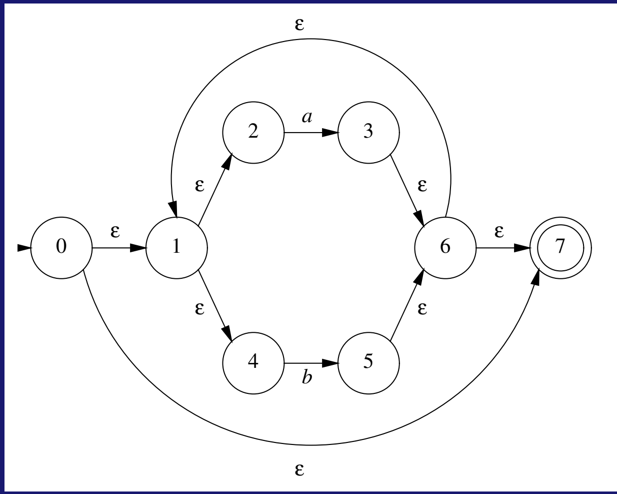


RE to NFA: example

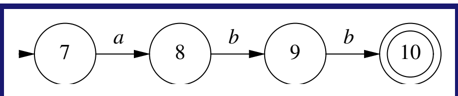
$a|b$



$(a|b)^*$



abb



NFA to DFA: the subset construction

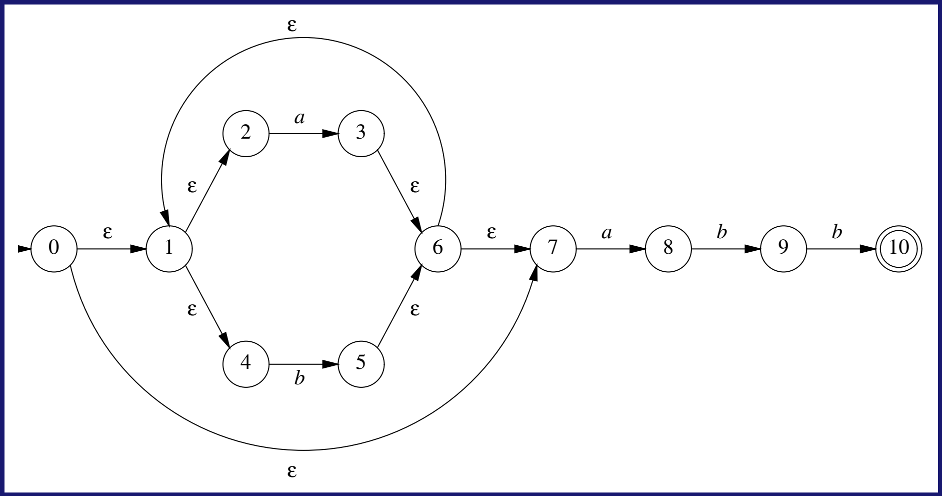
Input: NFA N
Output: A DFA D with states *Dstates* and transitions *Dtrans* such that $L(D) = L(N)$
Method: Let s be a state in N and T be a set of states, and using the following operations:

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

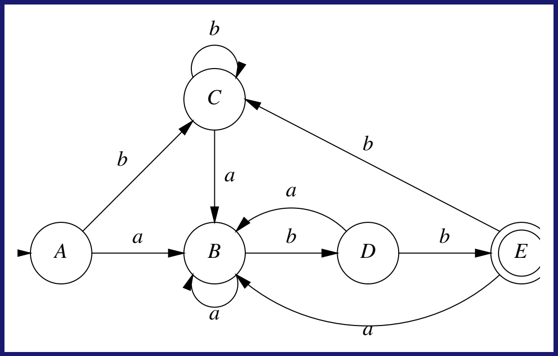


$A = \{0,1,2,4,7\}$ $\quad$ $D = \{1,2,4,5,6,7,9\}$

$B = \{1,2,3,4,6,7,8\}$ $\quad$ $E = \{1,2,4,5,6,7,10\}$

$C = \{1,2,4,5,6,7\}$

	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!)

So what is hard?

Language features that can cause problems:

reserved words

PL/I had no reserved words

`if then then then = else; else else = then;`

insignificant blanks

FORTRAN and Algol68 ignore blanks

```
do 10 i = 1,25
do 10 i = 1.25
```

string constants

special characters in strings

`newline`, `tab`, `quote`, `comment delimiter`

finite closures

some languages limit identifier lengths
adds states to count length
FORTRAN 66 $\rightarrow$ 6 characters

These can be swept under the rug in the language design

How bad can it get?

```
1           INTEGERFUNCTIONA
2           PARAMETER(A=6,B=2)
3           IMPLICIT CHARACTER*(A-B)(A-B)
4           INTEGER FORMAT(10),IF(10),DO9E1
5     100   FORMAT(4H)=(3)
6     200   FORMAT(4 )=(3)
7           DO9E1=1
8           DO9E1=1,2
9              IF(X)=1
10             IF(X)H=1
11             IF(X)300,200
12    300      CONTINUE
13          END
      C     this is a comment
          $ FILE(1)
14          END
```

Example due to Dr. F.K. Zadeck of IBM Corporation

Scanning MiniJava

White space:

- ' ', '\t', '\n', '\r', '\f'

Tokens:

- Operators, keywords (straightforward)
- Identifiers (straightforward)
- Integers (straightforward)
- Strings (tricky for escapes)