Lexical Analysis

Matthew Might
University of Utah
matt.might.net

Join the google group!

Plan

- What is lexical analysis?
- Formal languages
- Regular languages
- Deterministic automata
- Nondeterministic automata
- Lexer-generation: `flex`

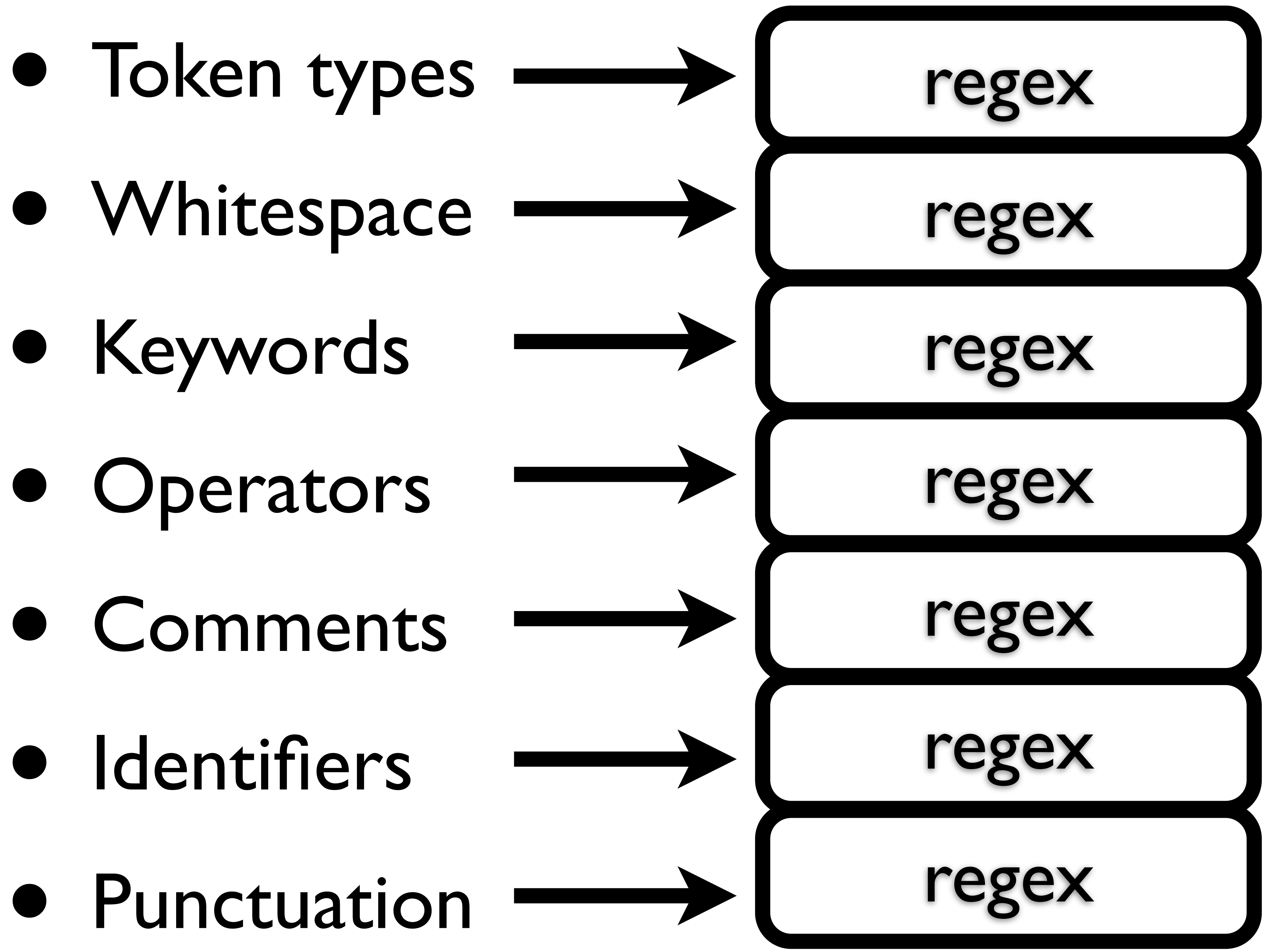
```
function id(x)
{
  return x ;  // comment
}
```

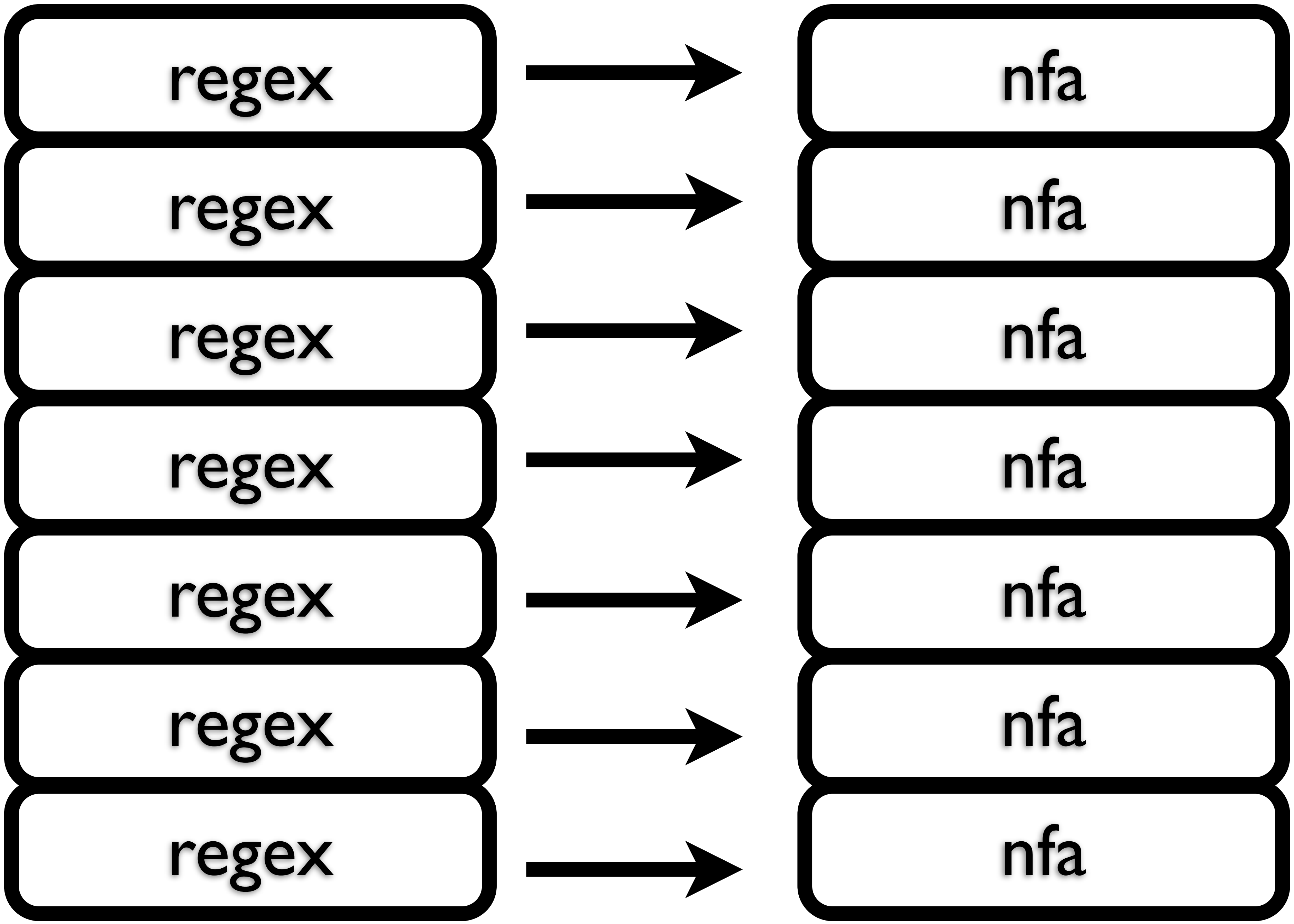
```
FUNCTION

IDENT(id)

LPAR

IDENT(x)

RPAR

LBRACE

RETURN

IDENT(x)

SEMI

RBRACE
```

Lexical specification

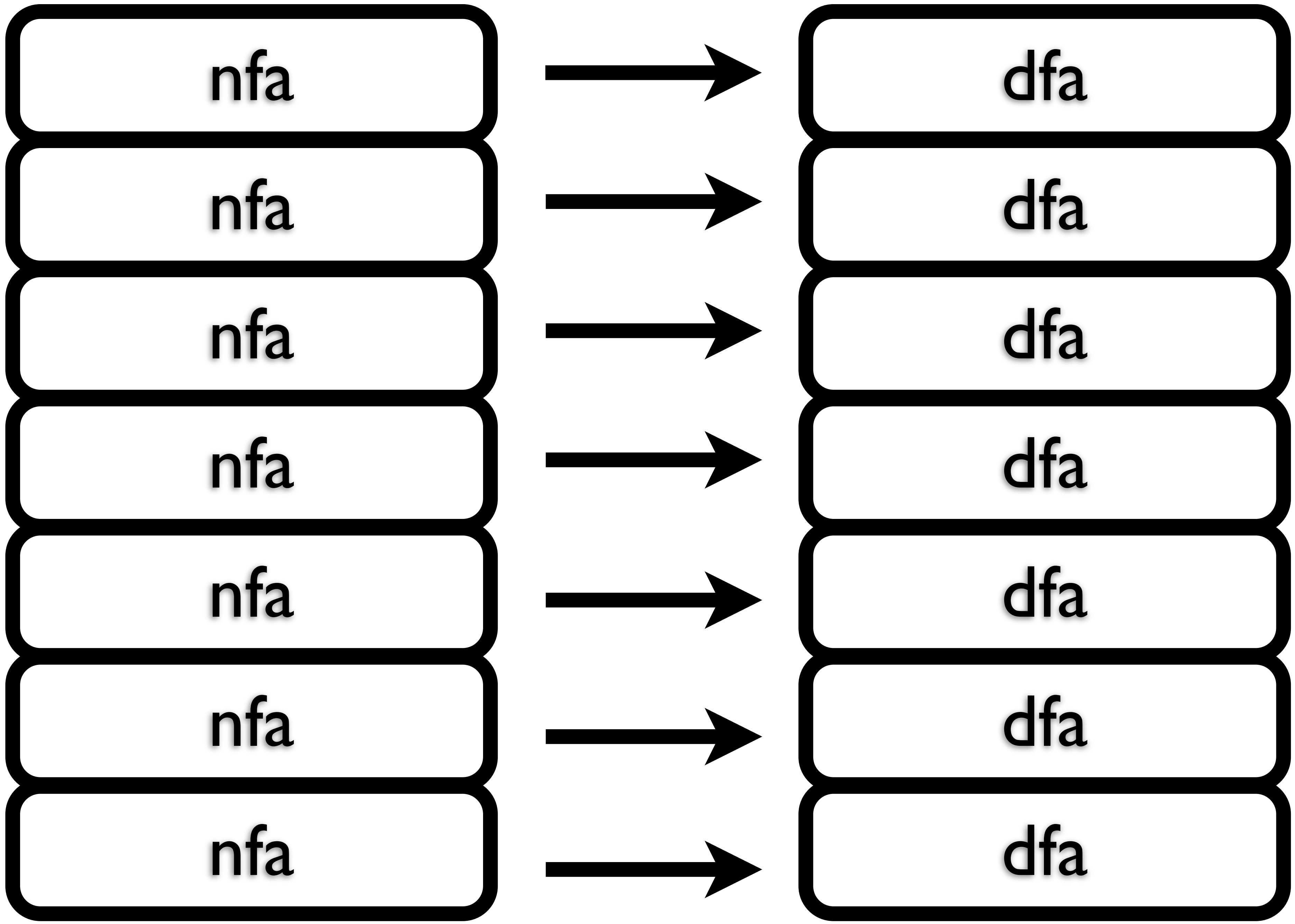
- Token types
- Whitespace
- Keywords
- Operators
- Comments
- Identifiers
- Punctuation

- Token types
- Whitespace
- Keywords
- Operators
- Comments
- Identifiers
- Punctuation

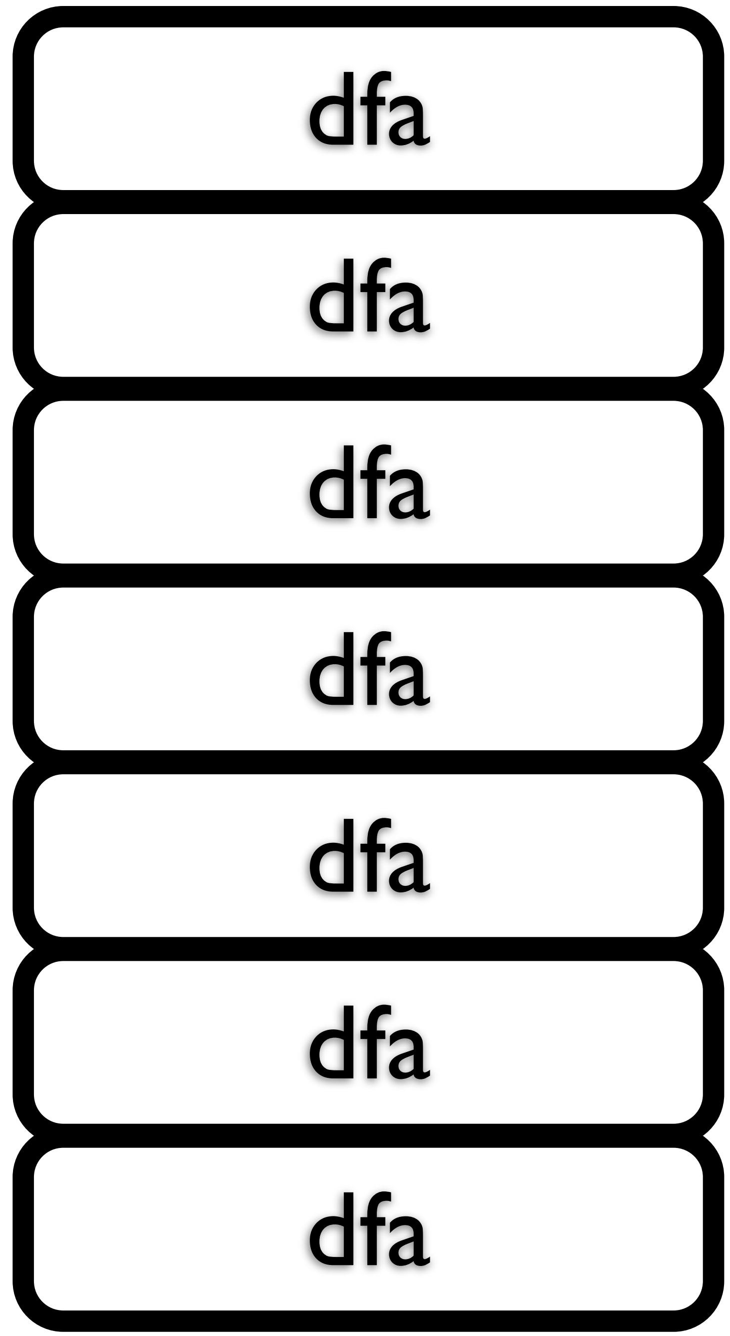

Token types
Whitespace
Keywords
Operators
Comments
Identifiers
Punctuation
regex
regex
regex
regex
regex
regex
regex



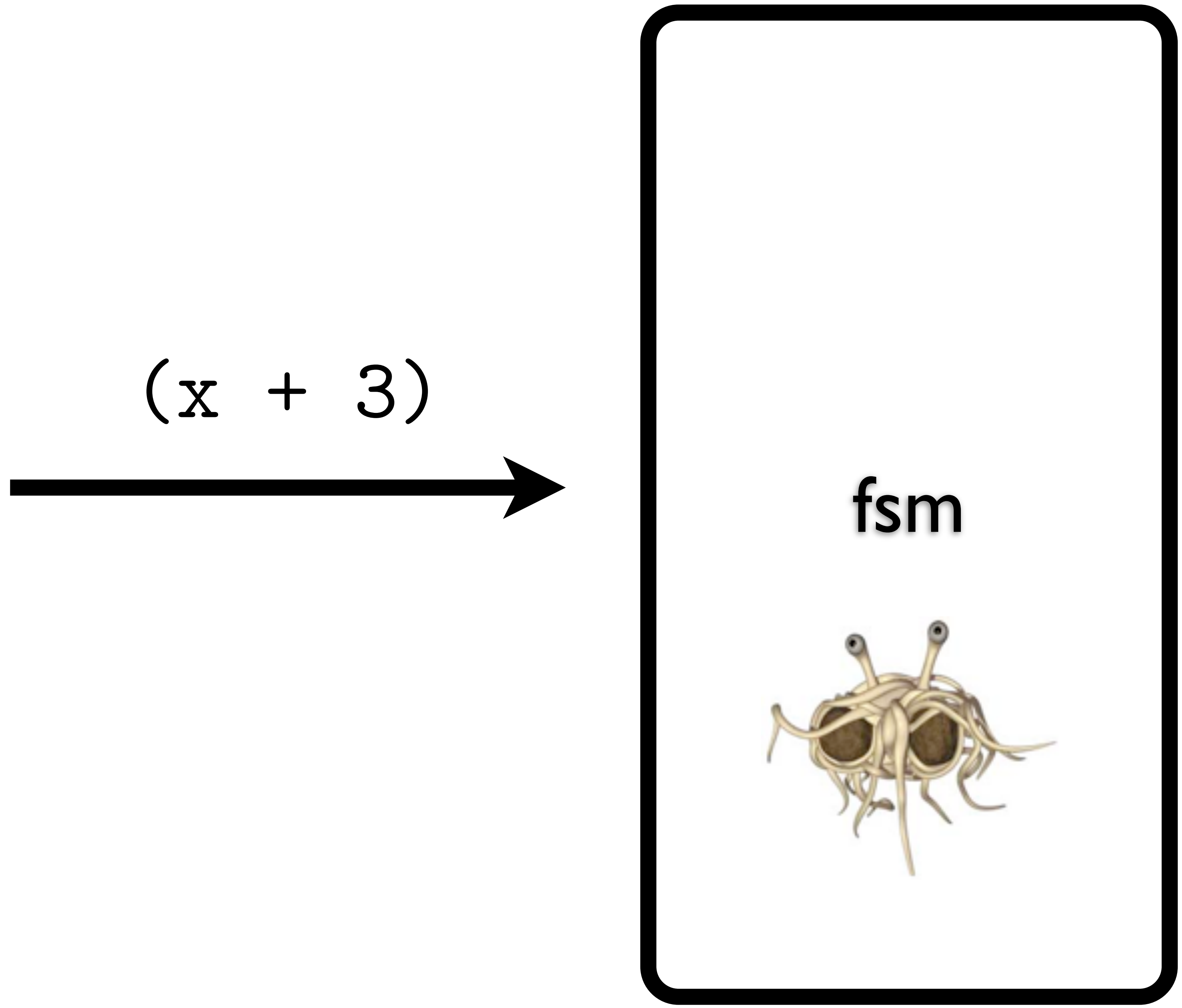
regex
nfa
regex
nfa
regex
nfa
regex
nfa
regex
nfa
regex
nfa
regex
nfa

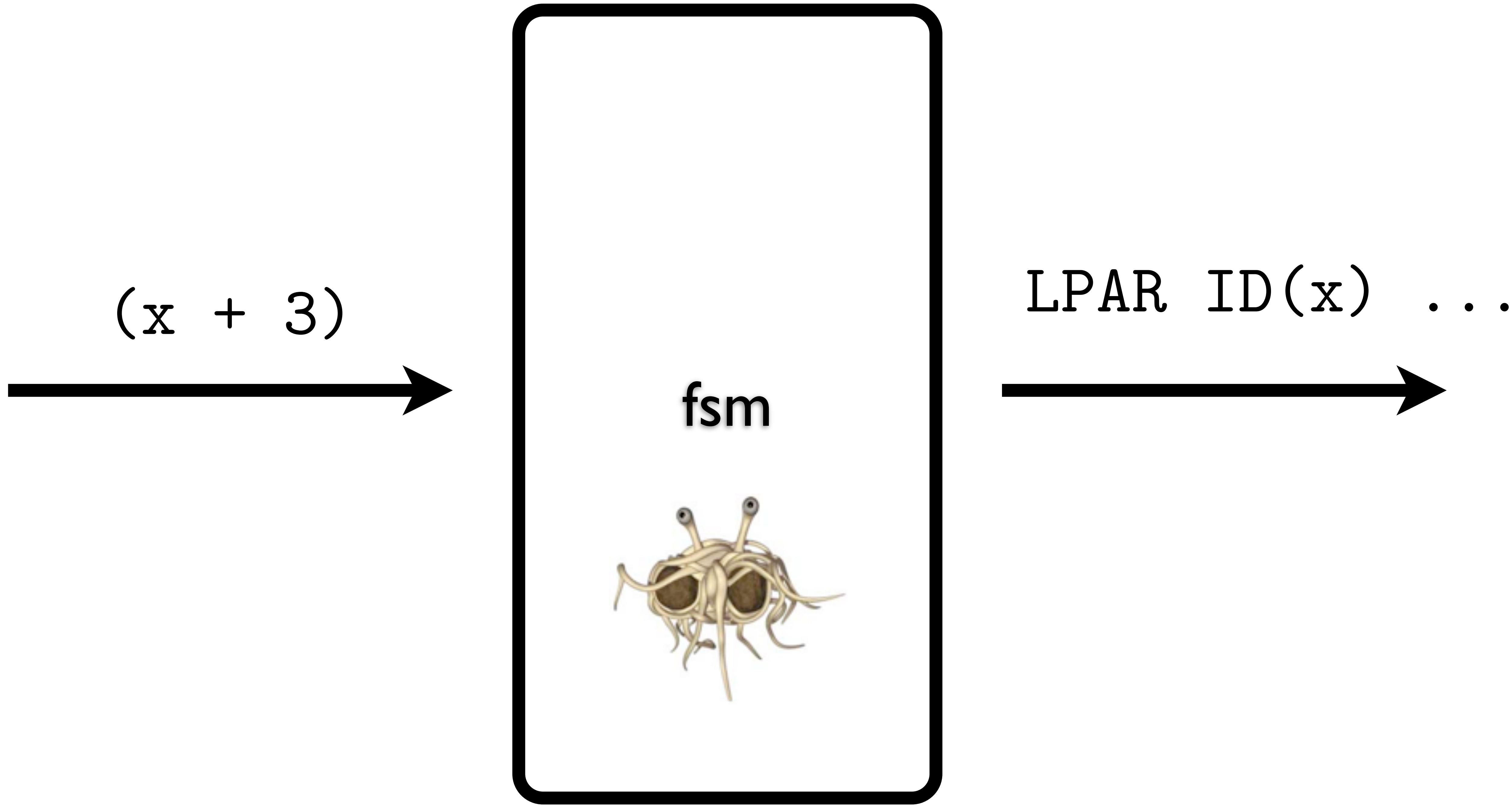


nfa
dfa
nfa
dfa
nfa
dfa
nfa
dfa
nfa
dfa
nfa
dfa
nfa
dfa


dfa
dfa
dfa
dfa
dfa
dfa
dfa

fsm


(x + 3)
fsm


(x + 3)
fsm
LPAR ID(x) ...

Formal languages

A **formal language** is a set of strings over an alphabet.

A **string** is a sequence
of characters from an alphabet.

Notation

A, Σ – Alphabets

L – Language

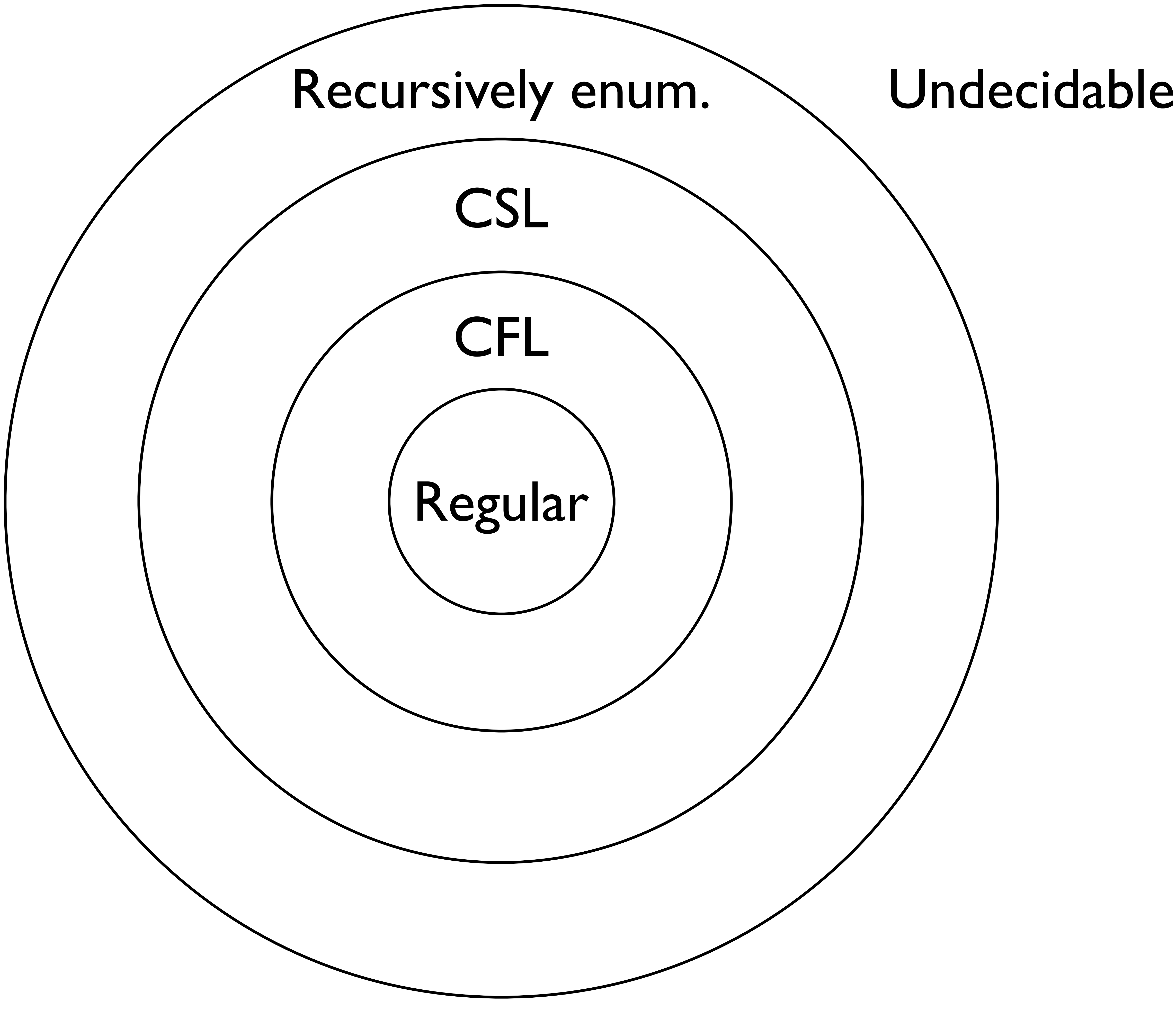
$L \subseteq A^*$ – Language over alphabet A

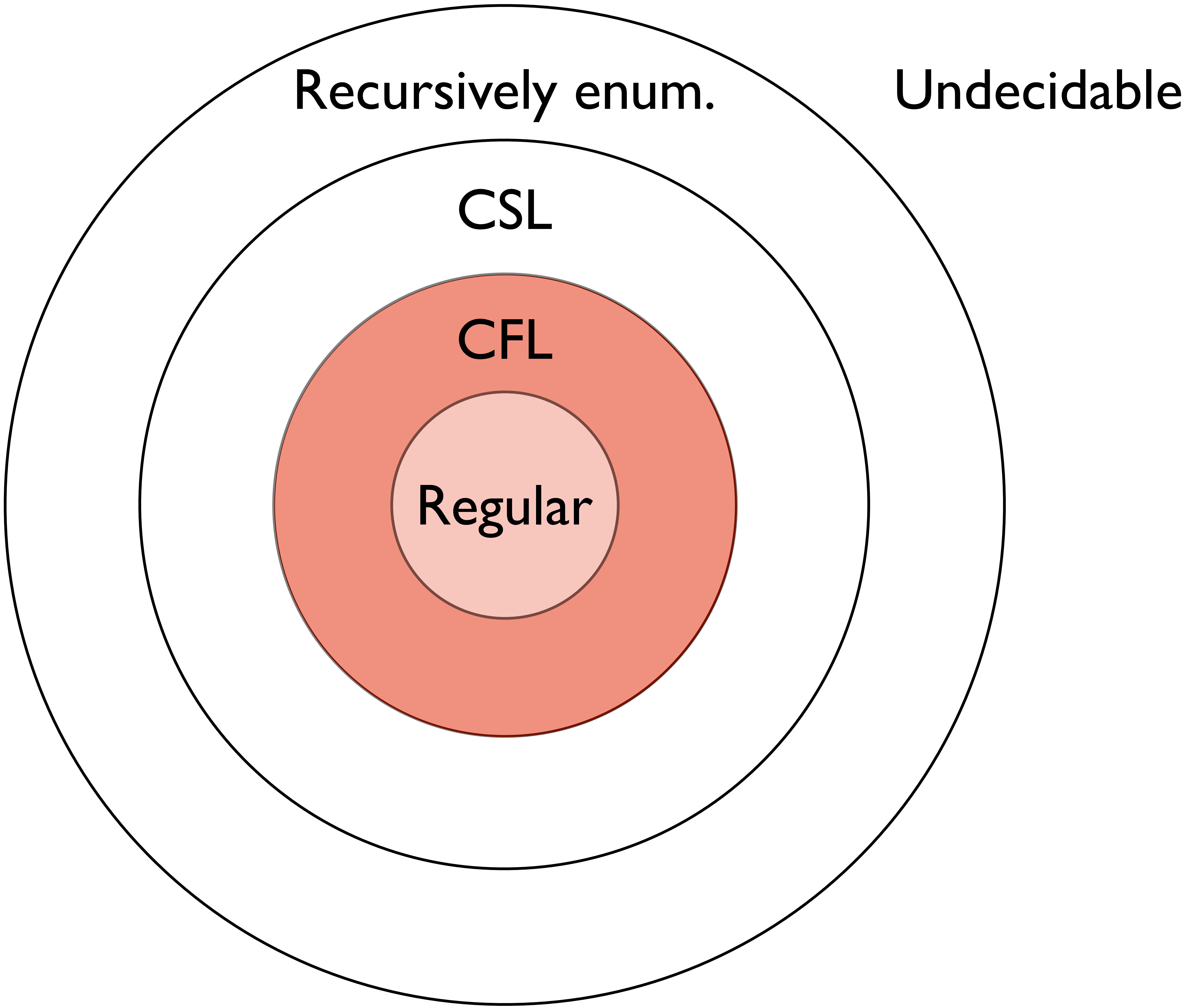
Examples

$$A = \{\texttt{a}, \texttt{b}\}$$

$$L_1 = \{\texttt{bab}, \texttt{abba}\}$$

$$L_2 = \{\texttt{a}, \texttt{aa}, \texttt{aaa}, \ldots\}$$


Recursively enum.
Undecidable
CSL
CFL
Regular


Recursively enum.
Undecidable
CSL
CFL
Regular

Atomic languages

$\emptyset$

ϵ

" "

$$\epsilon = \{\text{""}\}$$

$$\epsilon = ""$$

Primitive languages

c

$$c = \{\text{"}c\text{"}\}$$

Operations

Concatenation

$$L_1 \cdot L_2 = \{w_1 w_2 : w_1 \in L_1 \text{ and } w_2 \in L_2\}$$

Exercise

$$L_1 = \{\texttt{a}, \texttt{b}\}$$

$$L_2 = \{\texttt{c}, \texttt{d}\}$$

Union

$$L_1 \cup L_2 = \{w : w \in L_1 \text{ or } w \in L_2\}$$

Exercise

$$L_1 = \{\mathtt{a}, \mathtt{b}\}$$

$$L_2 = \{\mathtt{c}, \mathtt{d}\}$$

Exponentiation

$$L^n = \{w_1 \dots w_n : w_i \in L\}$$

$$L^0 = \epsilon$$

Exercise

$$\{\mathtt{a}, \mathtt{b}\}^3$$

Kleene star

$$L^\star = \bigcup_{n=0}^{\infty} L^n$$

Exercise

$$\{\mathtt{a}, \mathtt{b}\}^*$$

Concatenation,

union,

Kleene star.

Regular Languages

More regular operations!

Option

$$L^? \equiv L \cup \{\epsilon\}$$

Kleene plus

$$L^{+} = \bigcup_{n=1}^{\infty} L^{n}$$

Intersection

$$L_1 \cap L_2 = \{w : w \in L_1 \text{ and } w \in L_2\}$$

Difference

$$L_1 - L_2 = \{w : w \in L_1 \text{ and } w \notin L_2\}$$

Complement

$$\overline{L} = \{w : w \notin L\}$$

Reversal

$$L^R = \{\langle a_n, \ldots, a_1 \rangle : \langle a_1, \ldots, a_n \rangle \in L\}$$

Prefix

$$L^{\leq} = \{w_1 : \text{there exists } w_2 \text{ such that } w_1 w_2 \in L\}$$

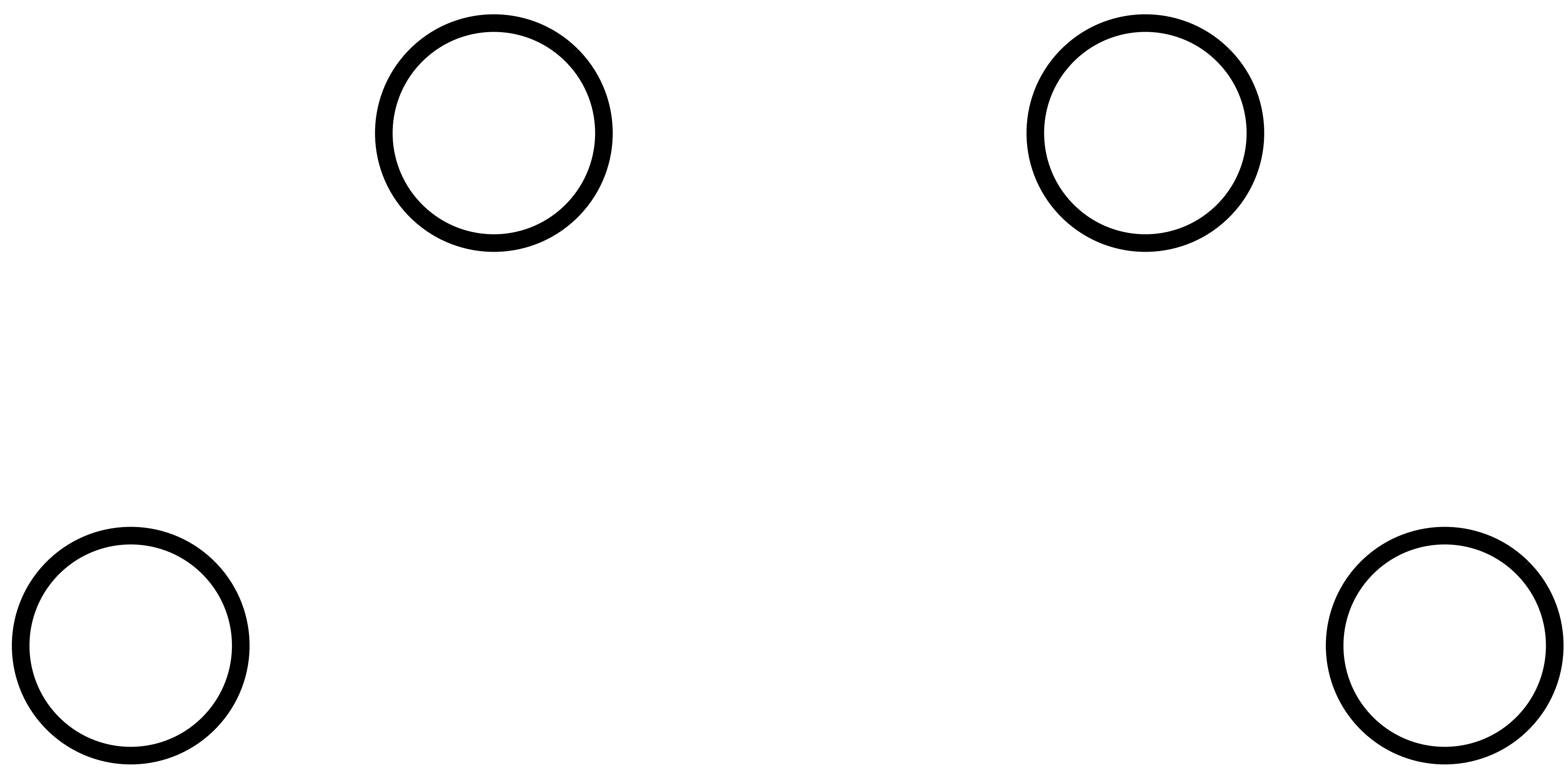
$$w \in L?$$

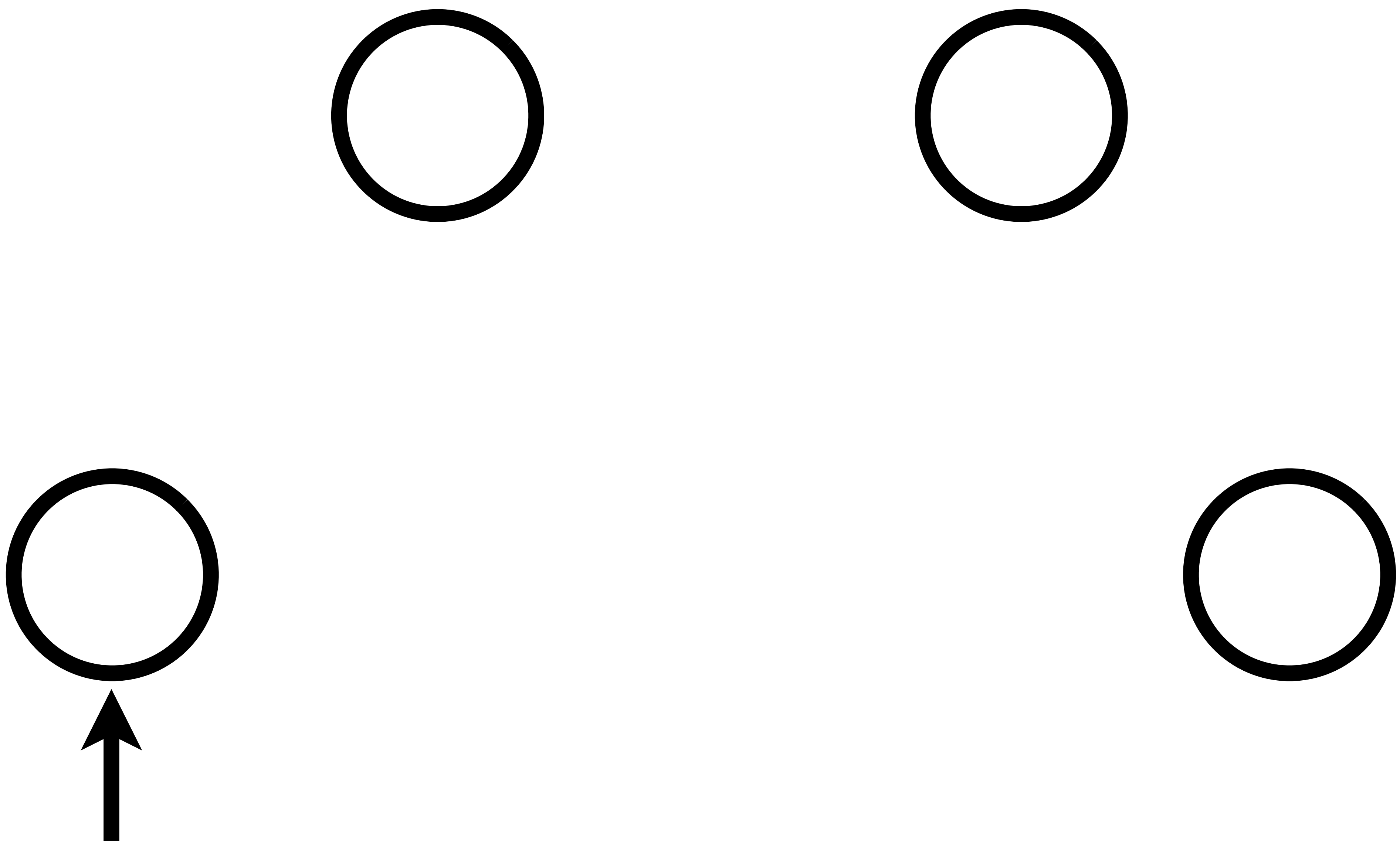
RegEx => NFA => DFA

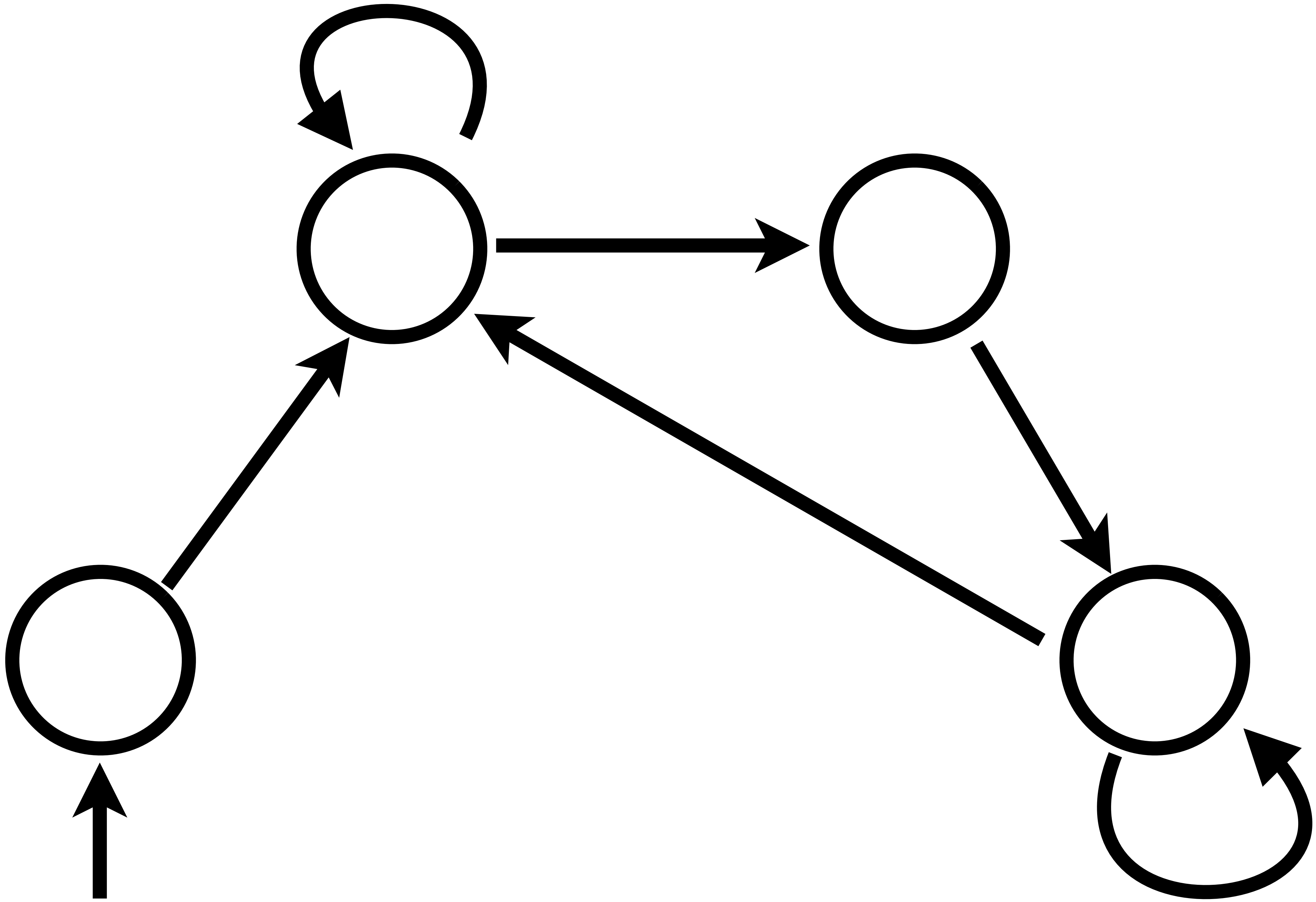
An **automaton** is a state machine that recognizes a formal language.

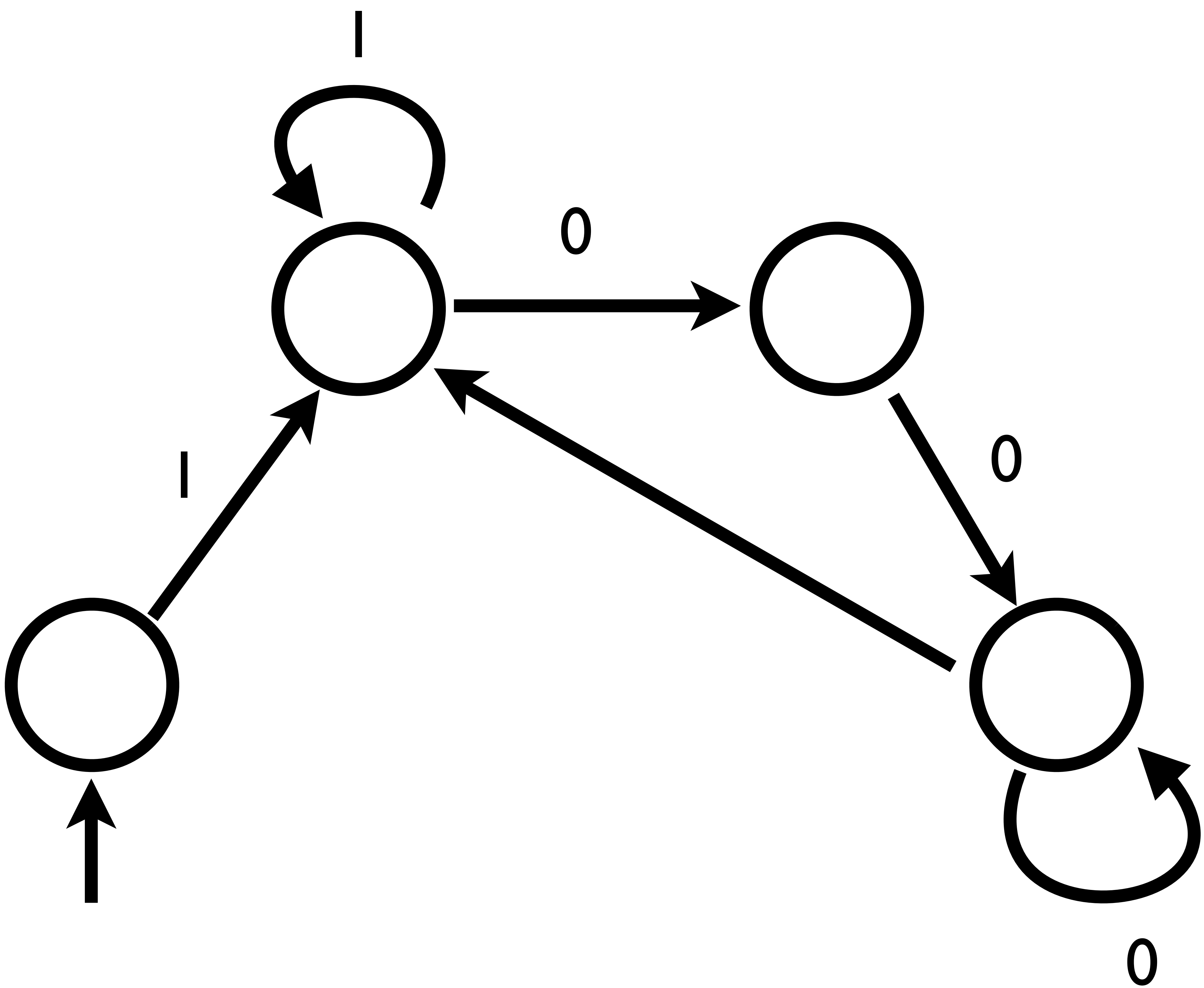
Formally, a **deterministic automaton** M is a 5-tuple (A, Q, q_0, δ, F) where:

- The set A is an alphabet.
- The set Q is a set of control states.
- The control state q_0 is the initial control state.
- The function $\delta : Q \times A \to Q$ determines the next state.
- The set F determines the final (accepting) states.

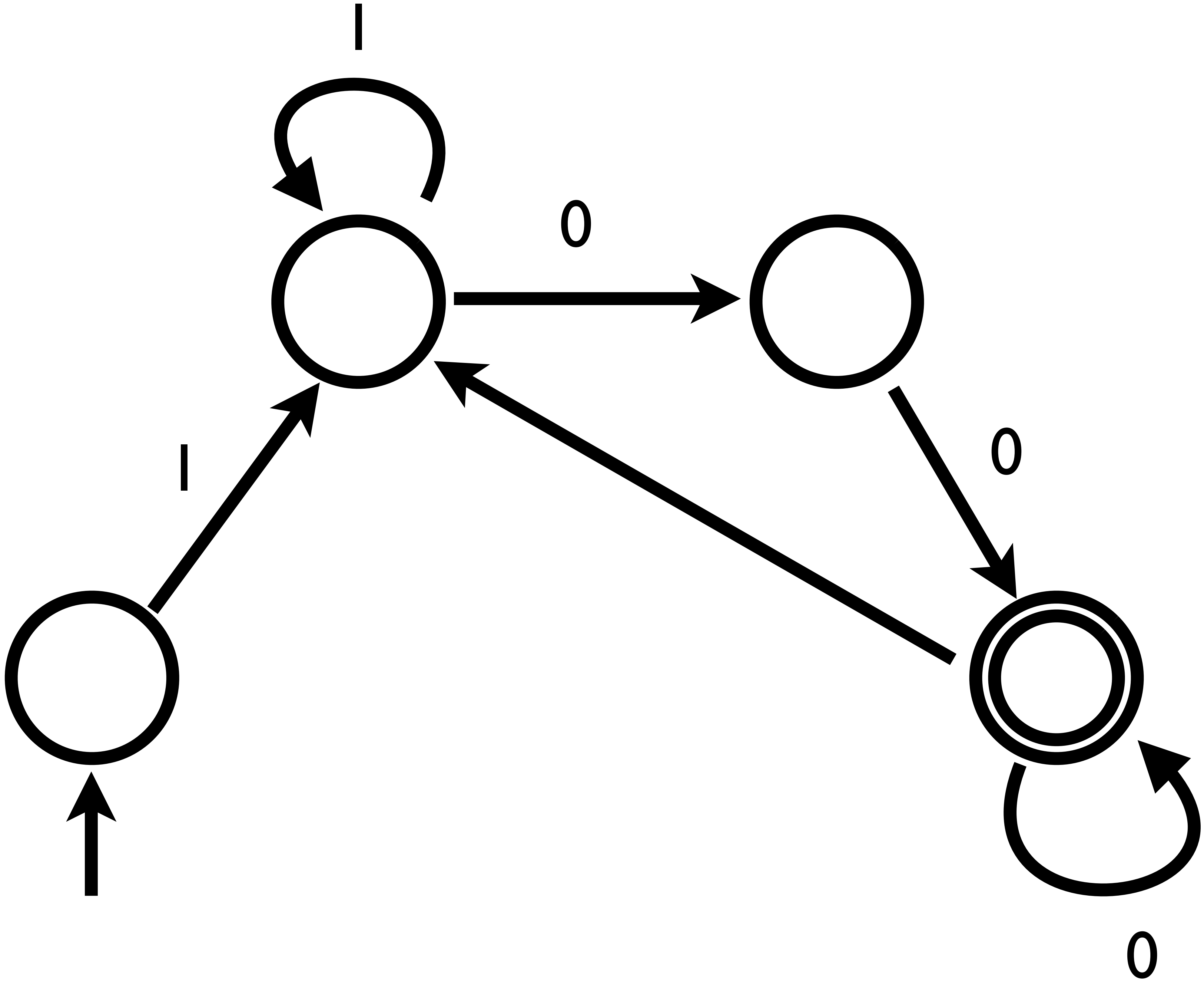








1
0
1
0
0


1
0
1
0
0

$$\mathcal{L}(A, Q, q_0, \delta, F) \subseteq A^*$$

If P_1 and $\ldots$ and P_n, then Q.

$$P_1 \quad \cdots \quad P_n$$

$$Q$$

$$\frac{P_1 \quad \cdots \quad P_n}{Q}$$

If P_1 and $\ldots$ and P_n, then Q.

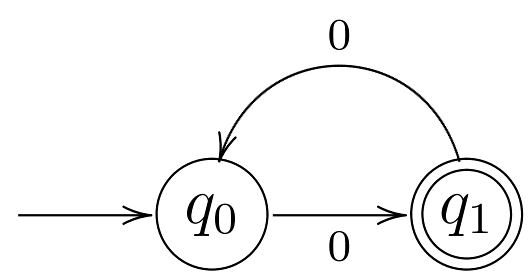
$$\frac{q_0 \in F}{\epsilon \in \mathcal{L}(A, Q, q_0, \delta, F)}$$

$$\frac{w \in \mathcal{L}(A, Q, \delta(q_0, c), \delta, F)}{cw \in \mathcal{L}(A, Q, q_0, \delta, F)}$$

Example 4.2. The automaton (A, Q, q_0, δ, F), where:

$$
\begin{aligned}
A &= \{0\} & \qquad \delta(q_0, 0) &= q_1 \\
Q &= \{q_0, q_1\} & \qquad \delta(q_1, 0) &= q_0 \\
F &= \{q_1\}\,, & &
\end{aligned}
$$

is depicted graphically as:



and it accepts exactly the set of strings with odd length—$\{0, 000, 00000, \ldots\}$.

Formally, a **nondeterministic automaton** M is also a 5-tuple (A, Q, q_0, δ, F) where:

- The set A is an alphabet.
- The set Q is a set of control states.
- The control state q_0 is the initial control state.
- The function $\delta : Q \times (A \cup \{\epsilon\}) \to \mathcal{P}(Q)$ determines the next state.
- The set F determines the final (accepting) states.

$$\delta : Q \times A \rightarrow Q$$

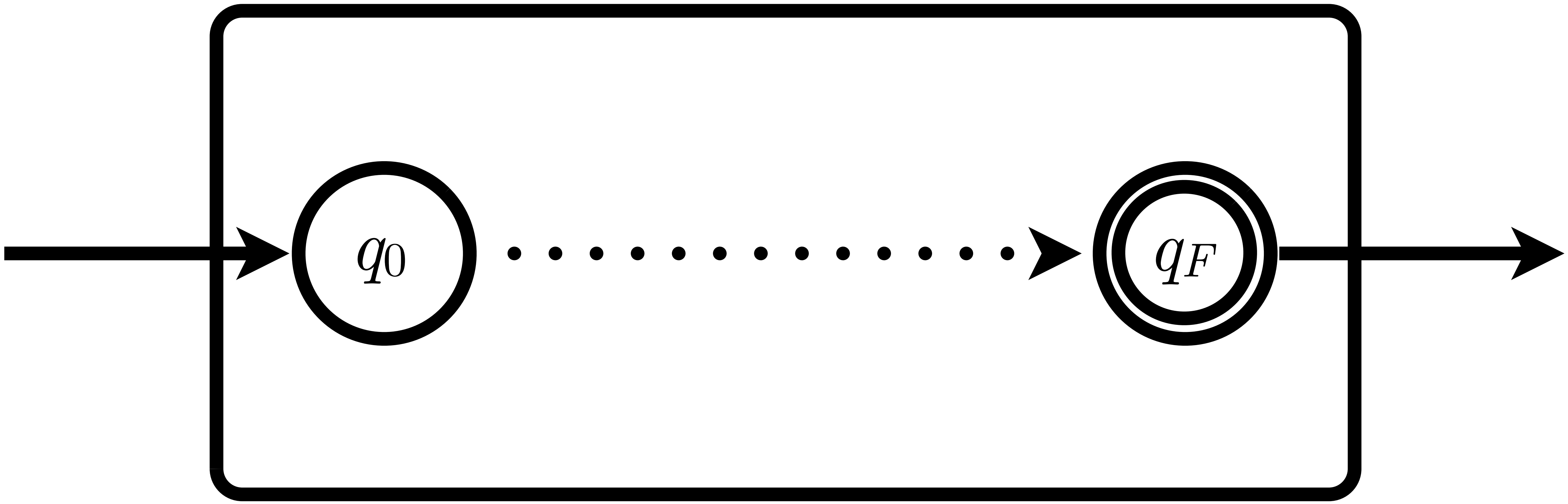
$$\delta : Q \times \left(A \cup \{\epsilon\}\right) \rightarrow \mathcal{P}(Q)$$

$$\frac{q_0 \in F}{\epsilon \in \boldsymbol{\mathcal{L}}(A, Q, q_0, \delta, F)}$$

$$\frac{w \in \mathcal{L}(A, Q, q', \delta, F) \qquad q' \in \delta(q_0, c)}{cw \in \mathcal{L}(A, Q, q_0, \delta, F)}$$

$$\frac{w \in \mathcal{L}(A, Q, q', \delta, F) \qquad q' \in \delta(q_0, \epsilon)}{w \in \mathcal{L}(A, Q, q_0, \delta, F)}$$

RegEx => NFA


q_0
q_F

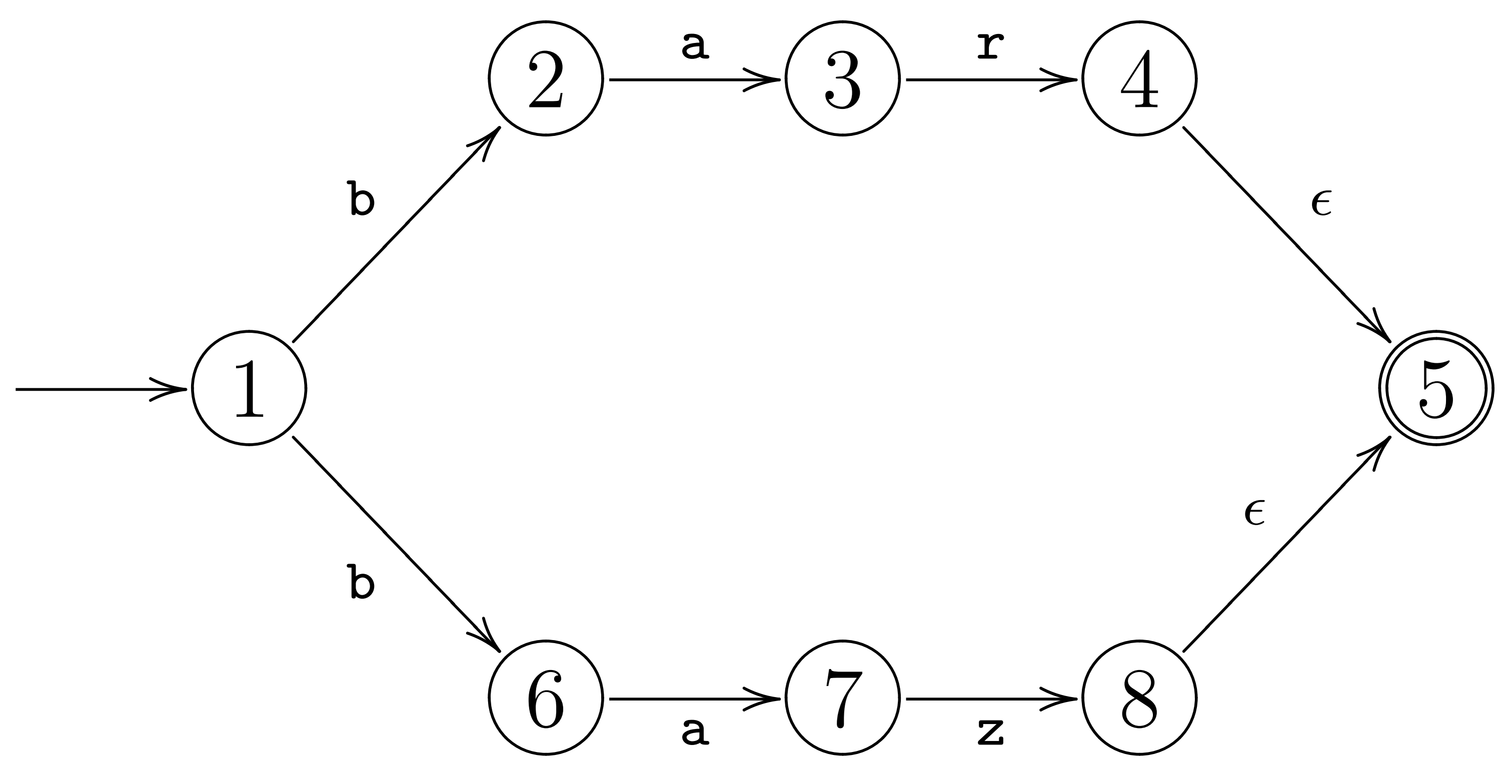
RegEx => NFA

- Empty language
- Empty-string singleton
- One-character singleton
- Concatenation
- Union
- Repetition

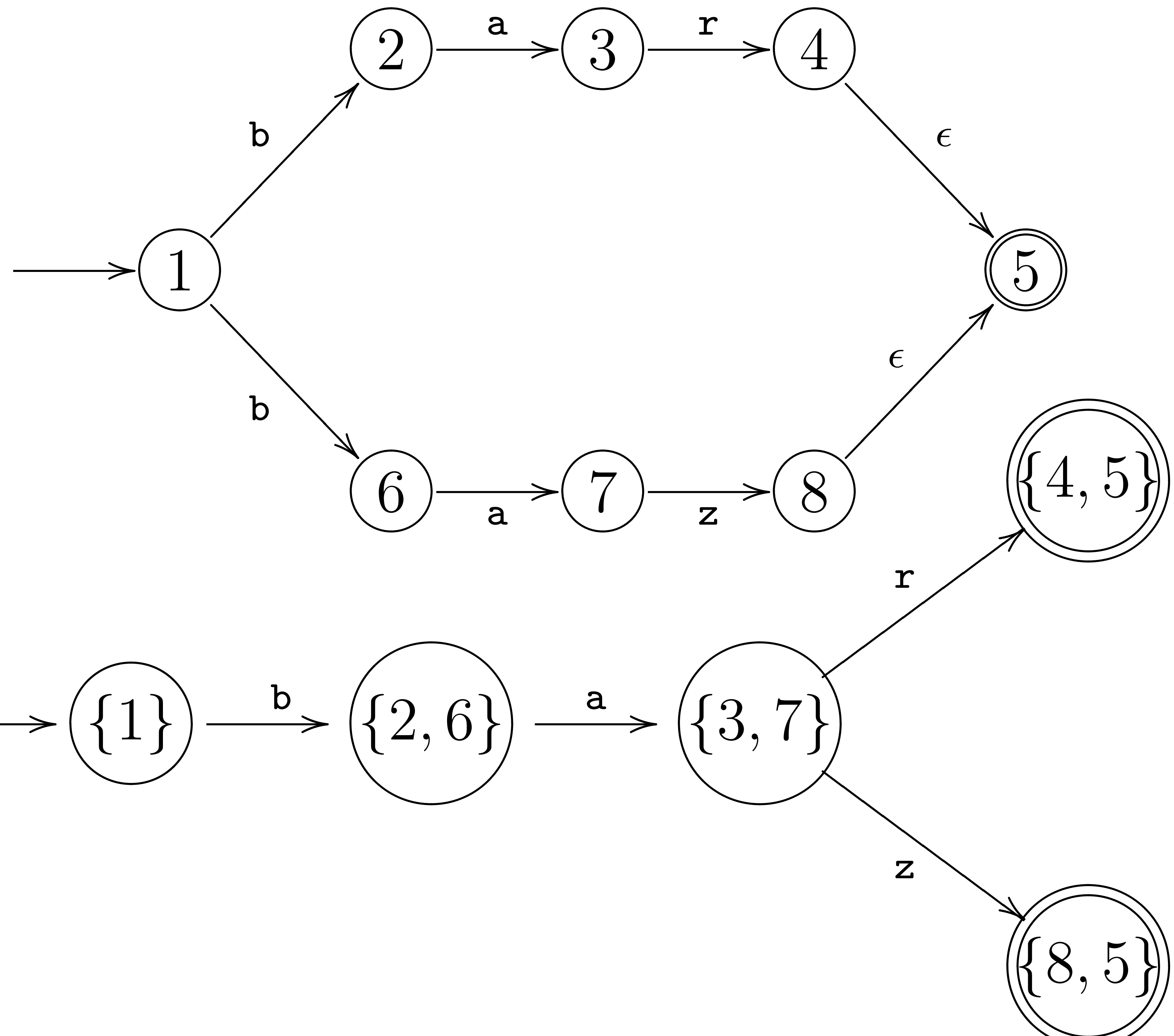
NFA => DFA

Thompson subset construction.

Example: NFA => DFA



Example: NFA => DFA



RegEx => NFA => DFA

lex

flex

lexical spec => lexer

- Identifiers match `[A-Za-z_][A-Za-z0-9_]*`
- Delimiters match `[();]`
- Operators match `[-*+/^=]`
- Integers match `-?([1-9][0-9]*|0)`
- Whitespace is ignored
- Comments match ^`#[^\n]*`

How lex works

foo.l

```
%%
[0-9]+   return 1;
[A-Z]+   return 2;
[ \t\n]  {}
%%
```

foo.l

```
%%
[0-9]+    return 1;
[A-Z]+    return 2;
[ \t\n]   {}
%%
```

lex.yy.c

```
int yylex() {
  ...
}
```

```
defs, opts, decs
%%
rules
%%
C code
```

Definitions

`name regex`

`{name}`

```
digit [0-9]
int   {digit}+

alpha  [A-Za-z_]
alnum  {alpha}|{digit}
id     {alpha}{alnum}*
```

Options

```
%option case-insensitive
```

```
%option yylineno
```

```
%option noyywrap
```

Declarations

```
%{
 C code
%}
```

`%s` $state_1$ $state_2$ ...

`%x` $state_1$ $state_2$ ...

Rules

pattern { *action* }

<state> pattern { action }

If in state $state$,
and $pattern$ matches the longest prefix,
then perform $action$.

```
%s state
%%
pat action
<state> pat action
```

```
%x state
%%
pat action
<state> pat action
```

Actions

{}

```
return token_type ;
```

BEGIN(*state*);

ECHO;

REJECT;

Example

- <MAIN> ‘‘(‘‘ { return(LPAR) ; }
- <MAIN> ‘‘)’’ { return(RPAR) ; }
- <MAIN> ‘‘/*’’ { BEGIN(COMMENT) ; }
- <COMMENT> ‘‘*/’’ { BEGIN(MAIN) ; }
- <COMMENT> . { }

Matching length

- Suppose the pattern: `f(oo)*`
- How many ways can match:

  `foooooooooooooooooooo`

Options

- Shortest match
- Longest match

Longest match

Given a string *w* and a set of regular expressions *R*, which regex can match the longest prefix of *w*?

Algorithm

```
NaïveRemoveLongestMatch(w ∈ A*,R ⊆ 2^A*)
  suffix ← w
  while (w ≠ ε)
   if ∃L ∈ R : ε ∈ L
      suffix ← w
   c:w ← w
   R ← D_c.R
  return suffix
```

Algorithm

```
RemoveLongestMatch(w ∈ A*,R ⊆ 2^A*)
  suffix ← w
  while (R ≠ ∅ or w ≠ ε)
   if ∃L ∈ R : ε ∈ L
      suffix ← w
   c:w ← w
   R ← D_c.R - {∅}
  return suffix
```

Example

- RegEx: fo*
- RegEx: foobar
- RegEx: foob
- Input string: foobarbaz

Examples

```
%{
 int num_lines = 0, num_chars = 0;
%}

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;
%%
int main(int argc, char* argv[]) {
   yylex();
   printf("# of lines = %d, # of chars = %d\n",
          num_lines, num_chars );
}
```

Questions?