CMPU 240 · Theory of Computation

Regular Expressions

12 February 2026

Where are we?

A language L is a ***regular language*** if there is a DFA D such that $L(D) = L$.

THEOREM The following are equivalent:

L is a regular language.

There is a DFA *D* where $L(D) = L$.

There is an NFA *N* where $L(N) = L$.

If $w \in \Sigma^*$ and $x \in \Sigma^*$, then wx is the *concatenation* of w and x.

If L_1 and L_2 are languages over Σ, the *concatenation* of L_1 and L_2 is the language L_1L_2, defined as

$$L_1L_2 = \{wx \mid w \in L_1 \text{ and } x \in L_2\}.$$

For example, if $L_1 = \{\texttt{a}, \texttt{ba}, \texttt{bb}\}$ and $L_2 = \{\texttt{aa}, \texttt{bb}\}$, then

$$L_1L_2 = \{\texttt{aaa}, \texttt{abb}, \texttt{baaa}, \texttt{babb}, \texttt{bbaa}, \texttt{bbbb}\}$$

Consider the language L = {aa, b}.

LL is the set of strings formed by concatenating pairs of strings in L:

{aaaa, aab, baa, bb}

LLL is the set of strings formed by concatenating triples of strings in L:

{aaaaaa, aaaab, aabaa, aabb, baaaa, baab, bbaa, bbb}

$LLLL$ is the set of strings formed by concatenating quadruples of strings in L…

We can define what it means to "exponentiate" a language as follows:

$L^0 = \{\varepsilon\}$

Base case: Any string formed by concatenating zero strings together is just the empty string.

$L^{n+1} = LL^n$

Recursive case: Concatenating $n+1$ strings together works by concatenating n strings, then concatenating one more.

We can define what it means to "exponentiate" a language as follows:

$L^0 = \{\varepsilon\}$

Base case: Any string formed by concatenating zero strings together is just the empty string.

*The only string you can form by gluing **no** strings together is the empty string!*

$L^{n+1} = LL^n$

Recursive case: Concatenating $n+1$ strings together works by concatenating n strings, then concatenating one more.

An important operation on languages is the *Kleene closure*, which is defined as

$$L^* = \{w \in \Sigma^* \mid \exists n \in \mathbb{N}_0 \,.\, w \in L^n\}.$$

That is, a word is in L^* iff it's in

the language L^0 or

the language L^1 or

the language L^2 or …

L^* consists of all the possible ways of concatenating zero or more strings in L.

If L = {a, bb}, then L^* = {

ε,

a, bb,

aa, abb, bba, bbbb,

aaa, aabb, abba, abbbb, bbaa, bbabb, bbbba, bbbbbb,

…

}

Last class, we saw that the class of regular languages (**REG**) is *closed* under the following operations:

Complement

Union

Intersection

Concatenation

Kleene star

That is, if L_1 and L_2 are regular languages over an alphabet Σ, then so are the following languages:

Complement	$\overline{L_1}$
Union	$L_1 \cup L_2$
Intersection	$L_1 \cap L_2$
Concatenation	$L_1 L_2$
Kleene star	L_1^*

Another view of regular languages

We've seen we can show a language is regular by

constructing a DFA for it or

constructing an NFA for it.

We can also show a language is regular by using closure properties to build it out of other regular languages.

Photograph by Benjamin D. Esham

This is a bottom-up approach to the regular languages:

- Start with a small set of simple languages we know to be regular.
- Use closure properties to combine these to form more elaborate languages.

Regular expressions provide a concise notation for describing this way of building regular languages out of simpler pieces.

They're use just about everywhere:

- They're built into JavaScript and used for data validation.
- They're used in the Unix **`grep`** tool to search for strings and **`flex`** to build compilers.
- They're used to clean and scrape data for large-scale analysis projects.

But, for the moment, put aside anything you know about writing regular expressions when programming.

Atomic regular expressions

Regular expression	*Language*	
∅	$\emptyset$	
ε	$\{\varepsilon\}$	
α	$\{\alpha\}$	*for any* $\alpha \in \Sigma$

Compound regular expressions

Regular expression	*Language*	
$(R_1 \cup R_2)$	$L(R_1) \cup L(R_2)$	*for any regular expressions* R_1 *and* R_2
$(R_1 \circ R_2)$	$L(R_1) \circ L(R_2)$	*for any regular expressions* R_1 *and* R_2
(R_1^*)	$L(R_1)^*$	*for any regular expression* R_1

DEFINITION R is a *regular expression* if R is

1	α for some $\alpha \in \Sigma$	*Basis*
2	ε	
3	$\varnothing$	
4	$(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions	*Recursive cases*
5	$(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions	
6	(R_1^*), where R_1 is a regular expression	

As with strings and languages, we can omit the $\circ$ and just indicate concatenation by writing two regular expressions next to each other.

$$\mathbf{ab} = \mathbf{a} \circ \mathbf{b}$$

Order of operations

We can omit parentheses to make regular expressions more compact, but this makes them ambiguous unless we define precedence:

1. Parentheses (R)
2. Kleene star R^*
3. Concatenation $R_1 \circ R_2$ or $R_1 R_2$
4. Union $R_1 \cup R_2$

ab*cud

```
ab*cud
```

```
a(b*)cud
```

$$ab^*cud$$

$$a(b^*)cud$$

$$(a \circ (b^*))cud$$

$$ab^*cud$$

$$a(b^*)cud$$

$$(a \circ (b^*))cud$$

$$((a \circ (b^*)) \circ c)ud$$

$$ab^*c \cup d$$

$$a(b^*)c \cup d$$

$$(a \circ (b^*))c \cup d$$

$$((a \circ (b^*)) \circ c) \cup d$$

$$(((a \circ (b^*)) \circ c) \cup d)$$

$$ab^*cud$$

$$a(b^*)cud$$

$$(a \circ (b^*))cud$$

$$((a \circ (b^*)) \circ c)ud$$

$$(((a \circ (b^*)) \circ c)ud)$$

This is the fully parenthesized version, following our formal, recursive definition of regular expressions.

Examples

$L(\texttt{oh}) = \{\text{oh}\}$

$L(\texttt{ohuawww*}) = \{\text{oh}, \text{aww}, \text{awww}, \text{awwww}, \text{awww}, \ldots\}$

$L(\texttt{(oua)(huwww*)}) =$

Examples

L(**oh**) = {oh}

L(**ohuawww***) = {oh, aww, awww, awwww, awww, …}

L(**(oua)(huwww*)**) = {

oh,

oww, owww, owwww, …,

ah,

aww, awww, awwww, …

}

Designing regular expressions

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains } \mathtt{aa} \text{ as a substring}\}$.

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

Let $L = \{w \in \Sigma^* \mid w$ contains `aa` as a substring$\}$.

`(aub)*aa(aub)*`

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

Let $L = \{w \in \Sigma^* \mid w$ contains `aa` as a substring$\}$.

```
(a∪b)*aa(a∪b)*
```

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

Let $L = \{w \in \Sigma^* \mid w$ contains `aa` as a substring$\}$.

`(a∪b)*aa(a∪b)*`

bbabbbaabab

aaaa

bbbbbabbbbaabbbbb

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

Let $L = \{w \in \Sigma^* \mid w$ contains `aa` as a substring$\}$.

`(a∪b)*aa(a∪b)*`

bbabbbaabab

aaaa

bbbbbabbbbaabbbbb

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

Let $L = \{w \in \Sigma^* \mid w$ contains aa as a substring$\}$.

Σ*aaΣ*

A convenient shorthand!

bbabbbaabab

aaaa

bbbbbabbbbaabbbbb

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

We can write $|w|$ to denote the length of the string w.

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

ΣΣΣΣ

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

$\Sigma\Sigma\Sigma\Sigma$

aaaa

baba

bbbb

baaa

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

ΣΣΣΣ

aaaa

baba

bbbb

baaa

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

Σ^4

aaaa

baba

bbbb

baaa

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

Σ^4

Another convenient shorthand!

aaaa

baba

bbbb

baaa

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

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

Σ^4

Another convenient shorthand!

aaaa

baba

bbbb

baaa

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

Here are some candidate regular expressions for L. Which are correct?

- $\Sigma^* a \Sigma^*$
- $b^* a b^* \cup b^*$
- $b^* (a \cup \varepsilon) b^*$
- $b^* a^* b \cup b^*$
- $b^* (a^* \cup \varepsilon) b^*$

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

b*(aυε)b*

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

b*(aυε)b*

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

b*(aυε)b*

bbbbabbb

bbbbbb

abbb

a

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

b*(aυε)b*

bbbbabbb

bbbbbb

abbb

a

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

Let $L = \{w \in \Sigma^* \mid w$ contains at most one $a\}$.

b*a?b*

bbbbabbb

bbbbbb

abbb

a

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

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

b*a?b*

Another convenient shorthand!

bbbbabbb

bbbbbb

abbb

a

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

```
mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
```

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

```
mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
```

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

aa*

mvassar@vassar.edu

matthew.vassar@vassarbrewery.com

matt@cs.vassar.edu

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

aa*

mvassar@vassar.edu

matthew.vassar@vassarbrewery.com

matt@cs.vassar.edu

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

```
aa*(.aa*)*
```

```
mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
```

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

```
aa*(.aa*)*
```

```
mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
```

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

```
aa*(.aa*)*@
```

```
mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
```

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

```
aa*(.aa*)*@
```

```
mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
```

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

```
aa*(.aa*)*@aa*.aa*
```

```
mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
```

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

```
aa*(.aa*)*@aa*.aa*
```

```
mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
```

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

```
aa*(.aa*)*@aa*.aa*(.aa*)*
```

```
mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
```

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

```
aa*(.aa*)*@aa*.aa*(.aa*)*
```

```
mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
```

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

```
a+(.aa*)*@aa*.aa*(.aa*)*
```

```
mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
```

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

You guessed it – another shorthand!

$$a^+(.aa^*)^*@aa^*.aa^*(.aa^*)^*$$

mvassar@vassar.edu

matthew.vassar@vassarbrewery.com

matt@cs.vassar.edu

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

$$a^+(.a^+)^*@a^+.a^+(.a^+)^*$$

mvassar@vassar.edu

matthew.vassar@vassarbrewery.com

matt@cs.vassar.edu

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

$$a^+(.a^+)^*@a^+.a^+(.a^+)^*$$

mvassar@vassar.edu

matthew.vassar@vassarbrewery.com

matt@cs.vassar.edu

A more elaborate design

Let Σ = {a, ., @}, where a represents “any letter”.

Let’s make a regular expression for email addresses.

$$a^+(.a^+)^*@a^+(.a^+)^+$$

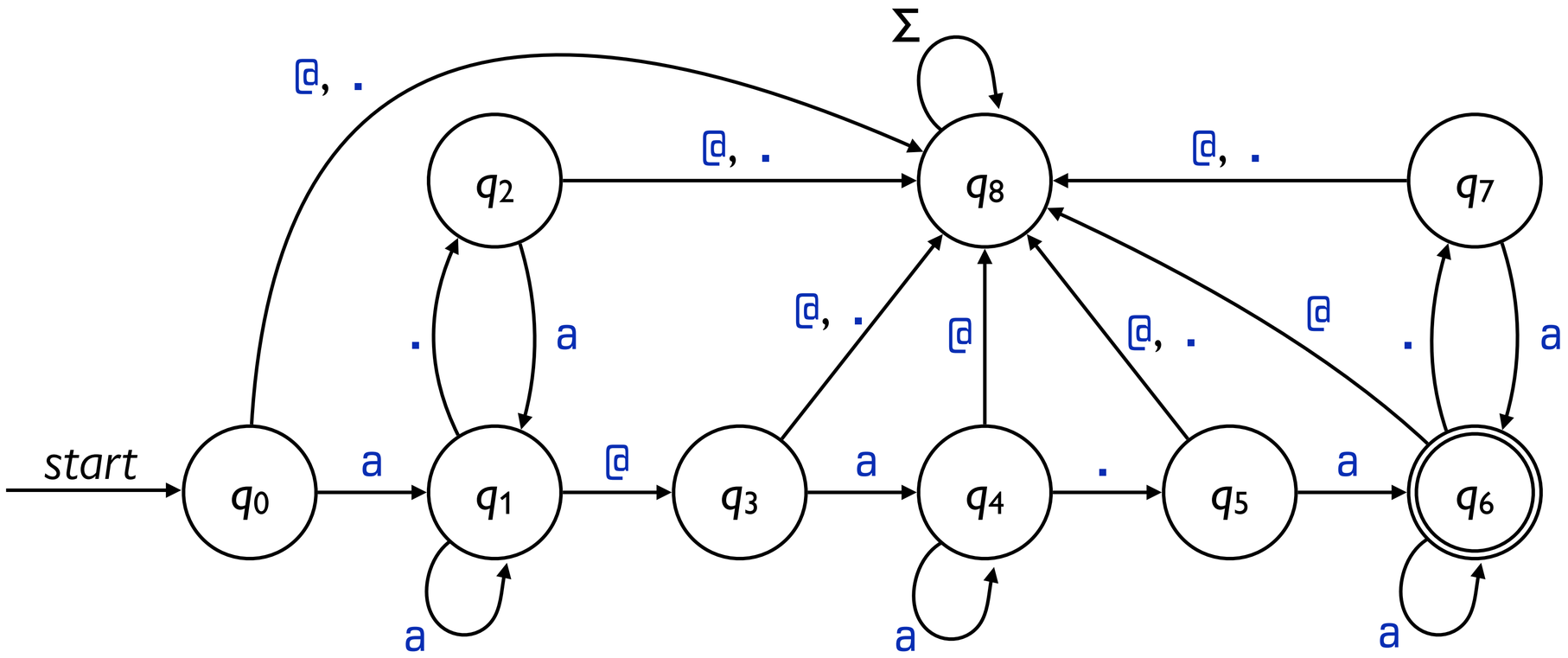
mvassar@vassar.edu

matthew.vassar@vassarbrewery.com

matt@cs.vassar.edu

For comparison

`a+(.a+)*@a+(.a+)+`



Shorthands summary

Σ is a shorthand for “any character in Σ”

R^n is a shorthand for $RR\ldots R$ (n times)

R`?` is shorthand for `(`*R*`|ε)` – that is, zero or one copies of *R*.

R^+ is a shorthand for RR^* – that is, one or more copies of *R*.

Optional interlude: Unix regular expressions

From the beginning (of time), Unix has used regular expressions in many places, including the **`grep`** command.

`grep` = **g**lobal (search for a) **r**egular **e**xpression and **p**rint

Many Unix commands use an extended RE notation, but it still expresses only the regular languages.

A big difference is that Unix regular expressions typically match *anywhere* in a string, whereas our regular expressions must match the *entire* string.

Unix regular expression notation

$R_1 \cup R_2$ is written as $R_1 \mid R_2$

Σ is written as .

Unix regular expression notation

$[a_1 a_2 \ldots a_n] = a_1 \cup a_2 \cup \cdots \cup a_n$.

As a shorthand, you can also specify ranges of ASCII characters, e.g.,

- `[a-z]` = any lowercase letter
- `[a-zA-Z]` = any letter

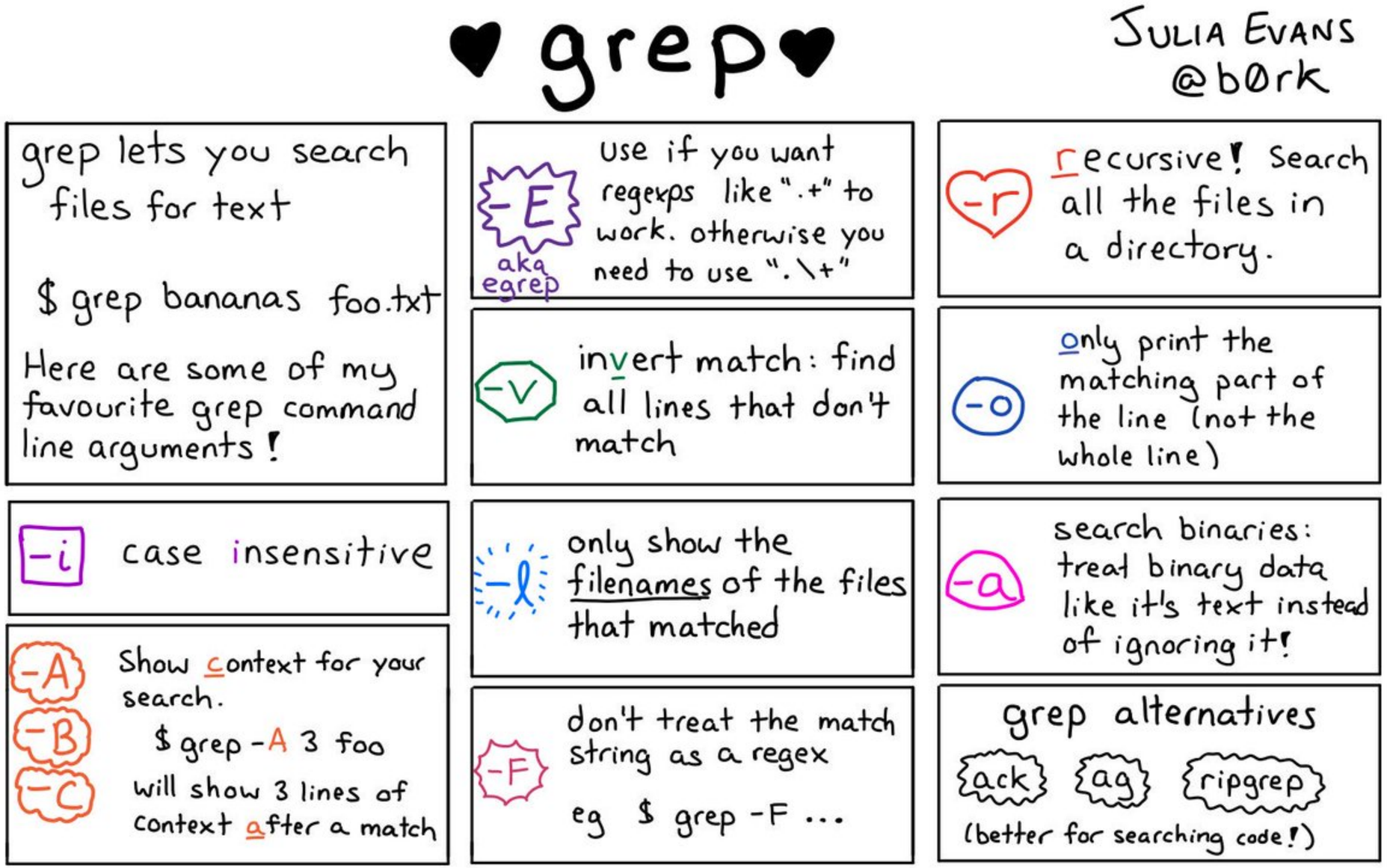
Unix regular expression notation

Since characters like brackets, dashes, and dots have special meaning, if you want to match them, you need to quote with backslash (\).

Perl, Python, Emacs, …

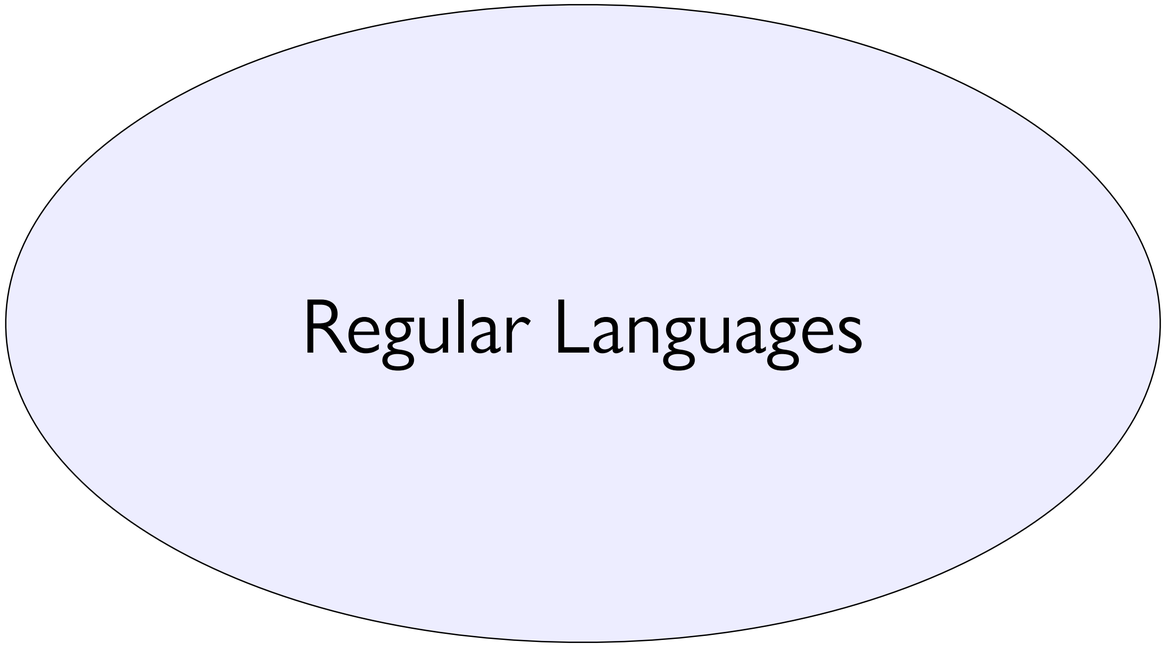
Include additional extensions, notably character classes like `\b` for word boundary characters, `\w` for word characters, etc.

With each implementation of regular expressions, they become less standard, so what you write for one language or application won't work in another.

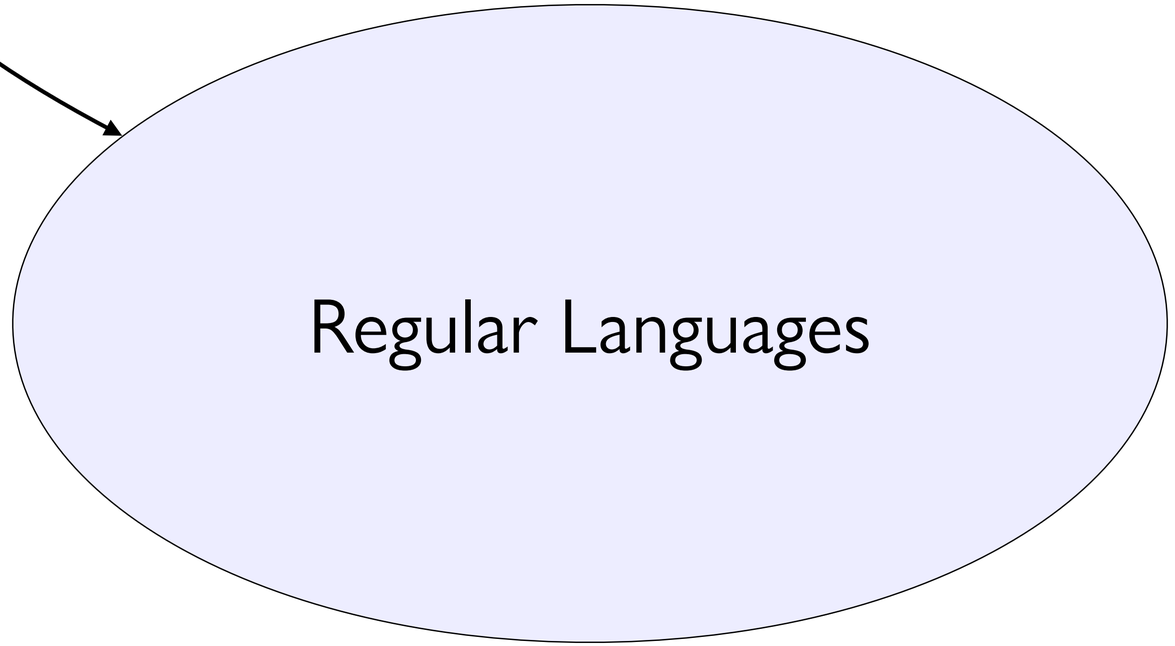


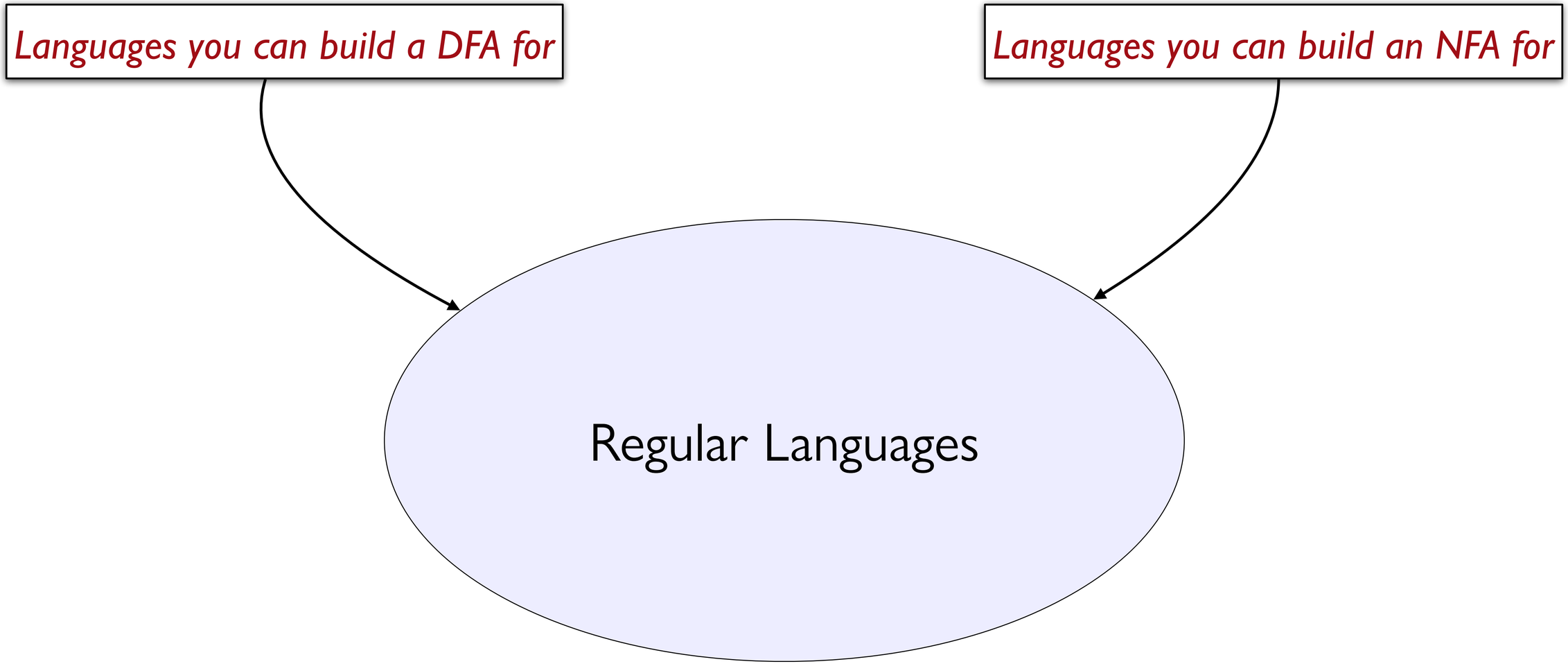
nitter.net/bork/status/991880504805871616

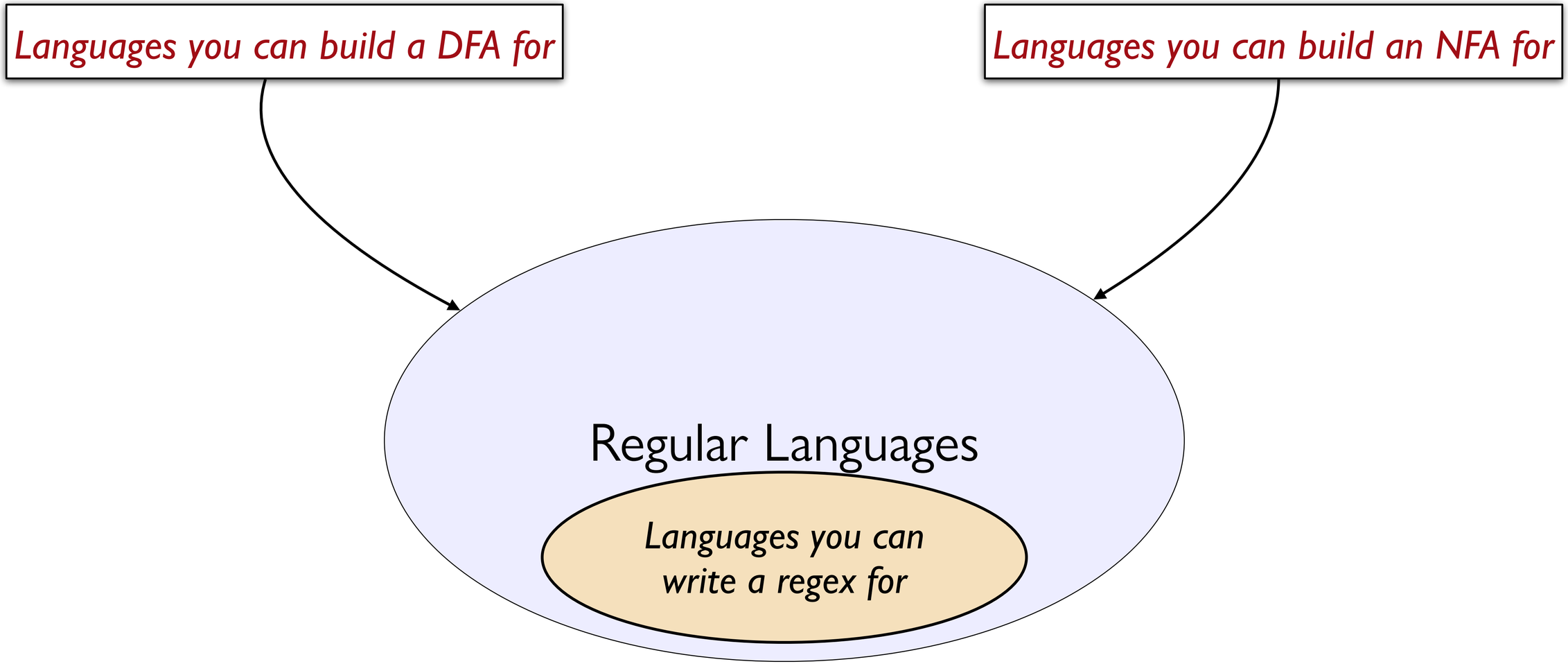
The power of regular expressions

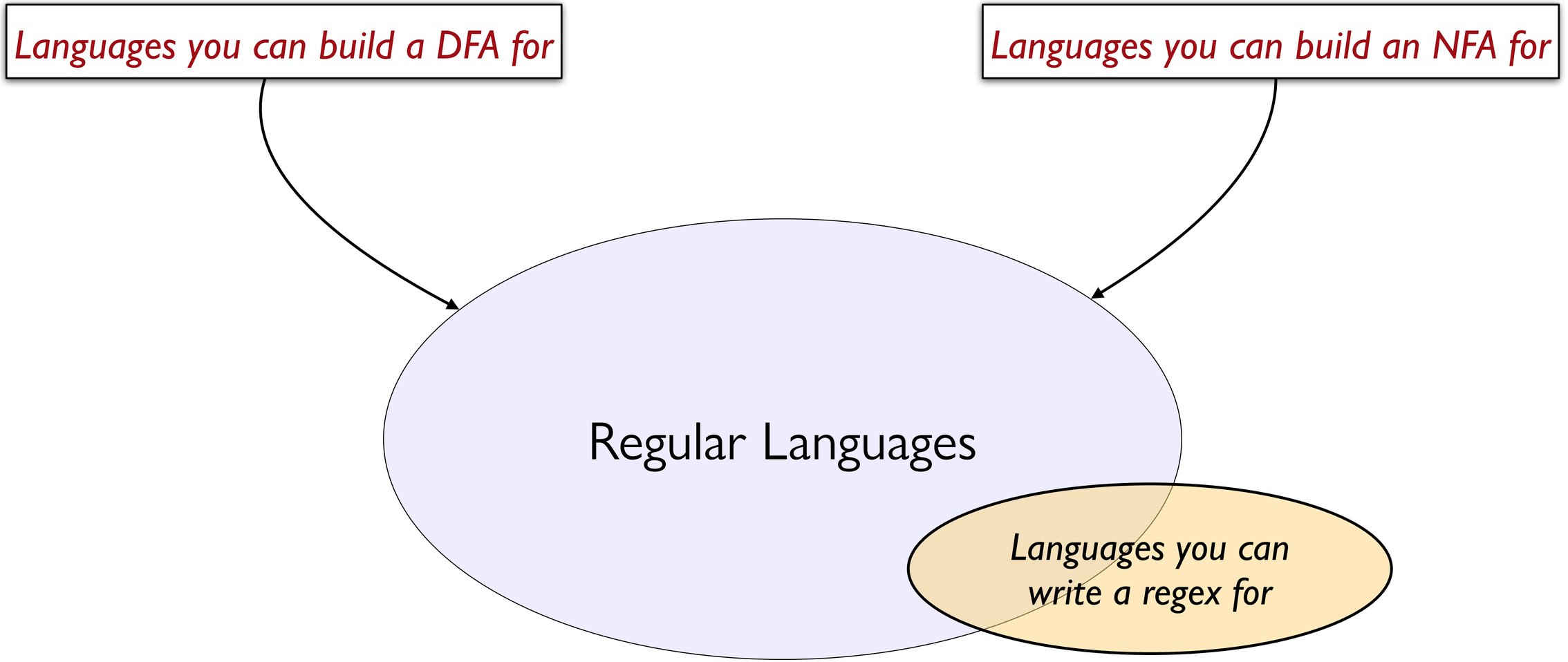

Regular Languages

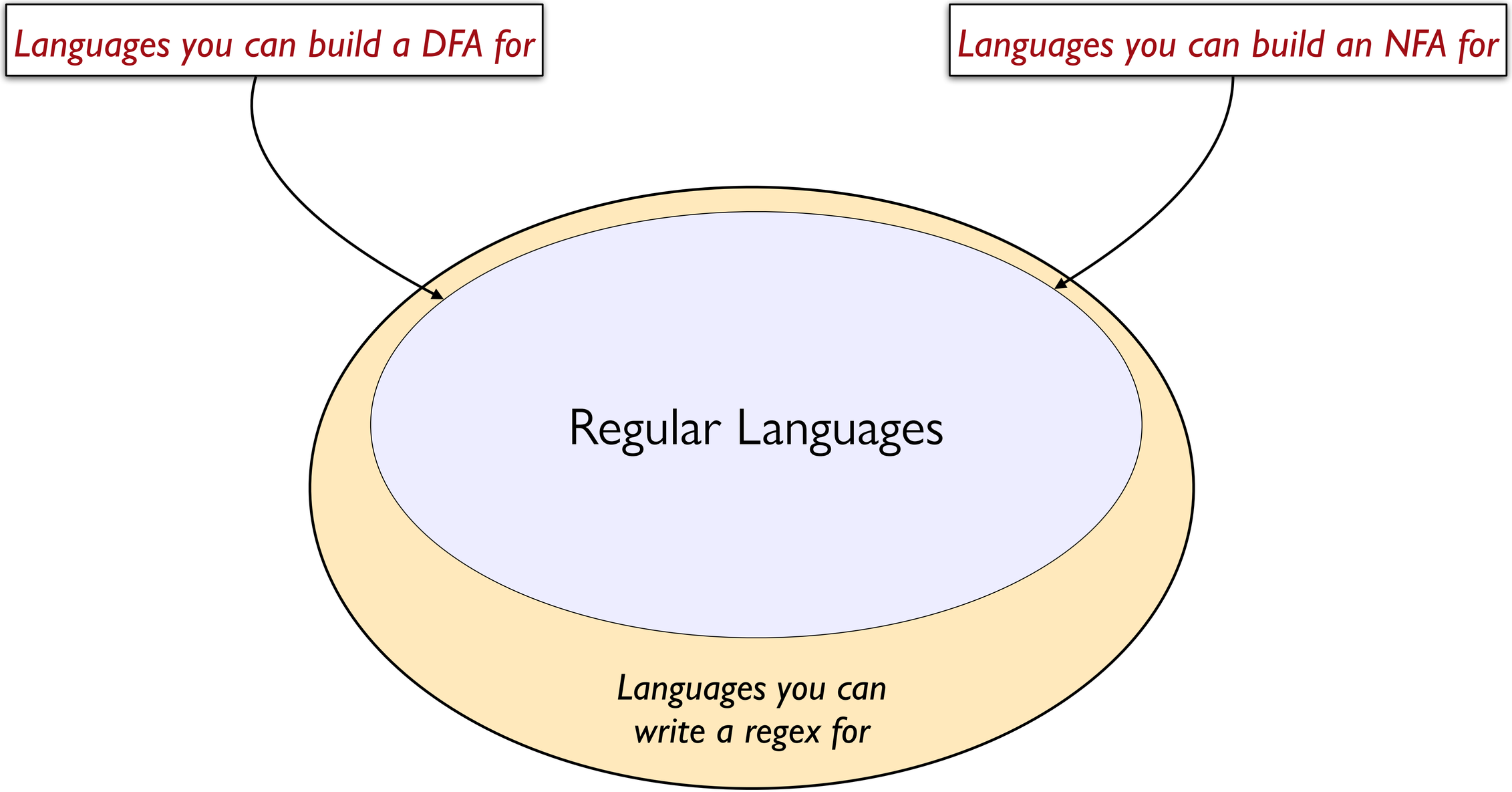
Languages you can build a DFA for

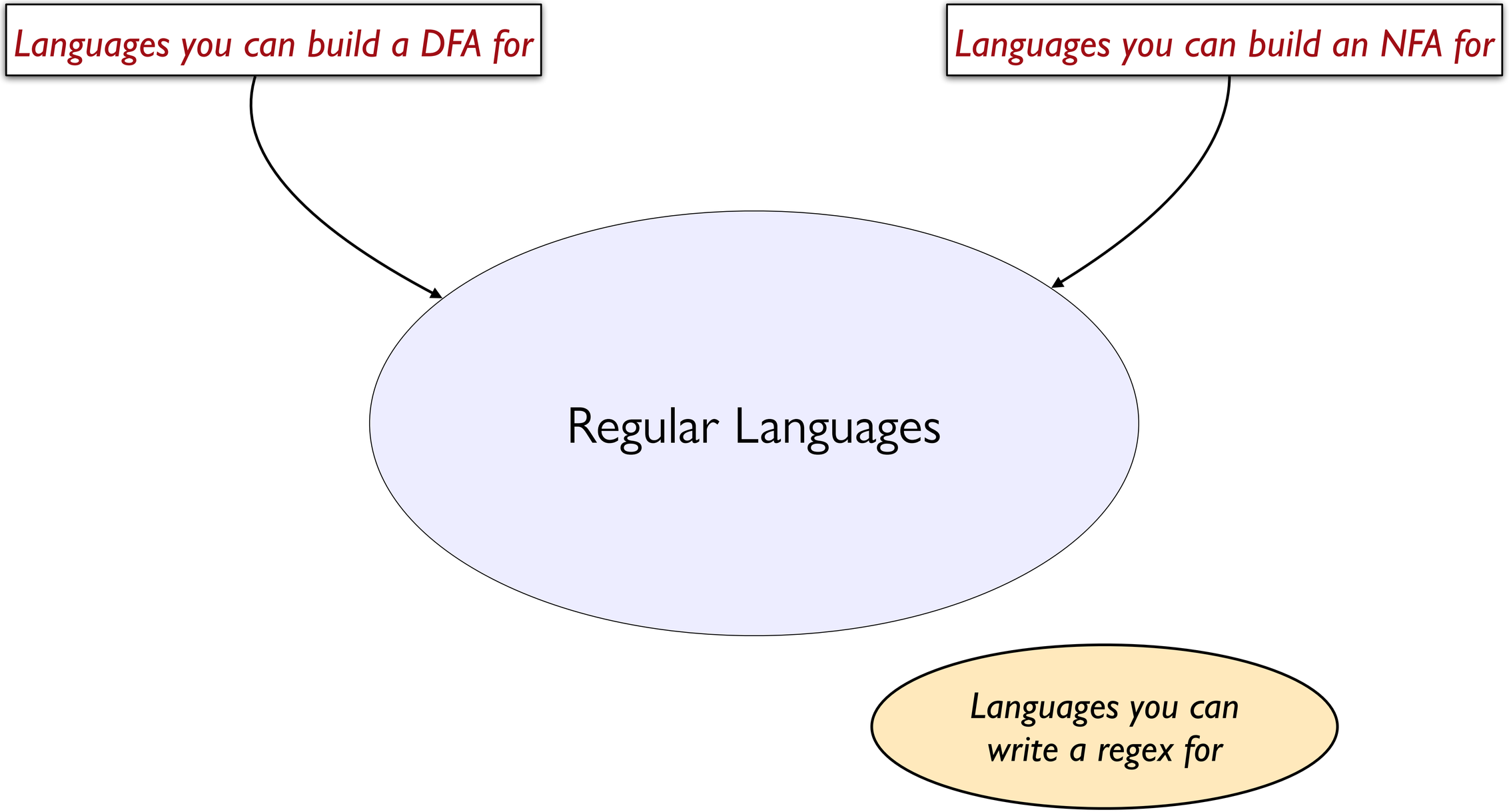


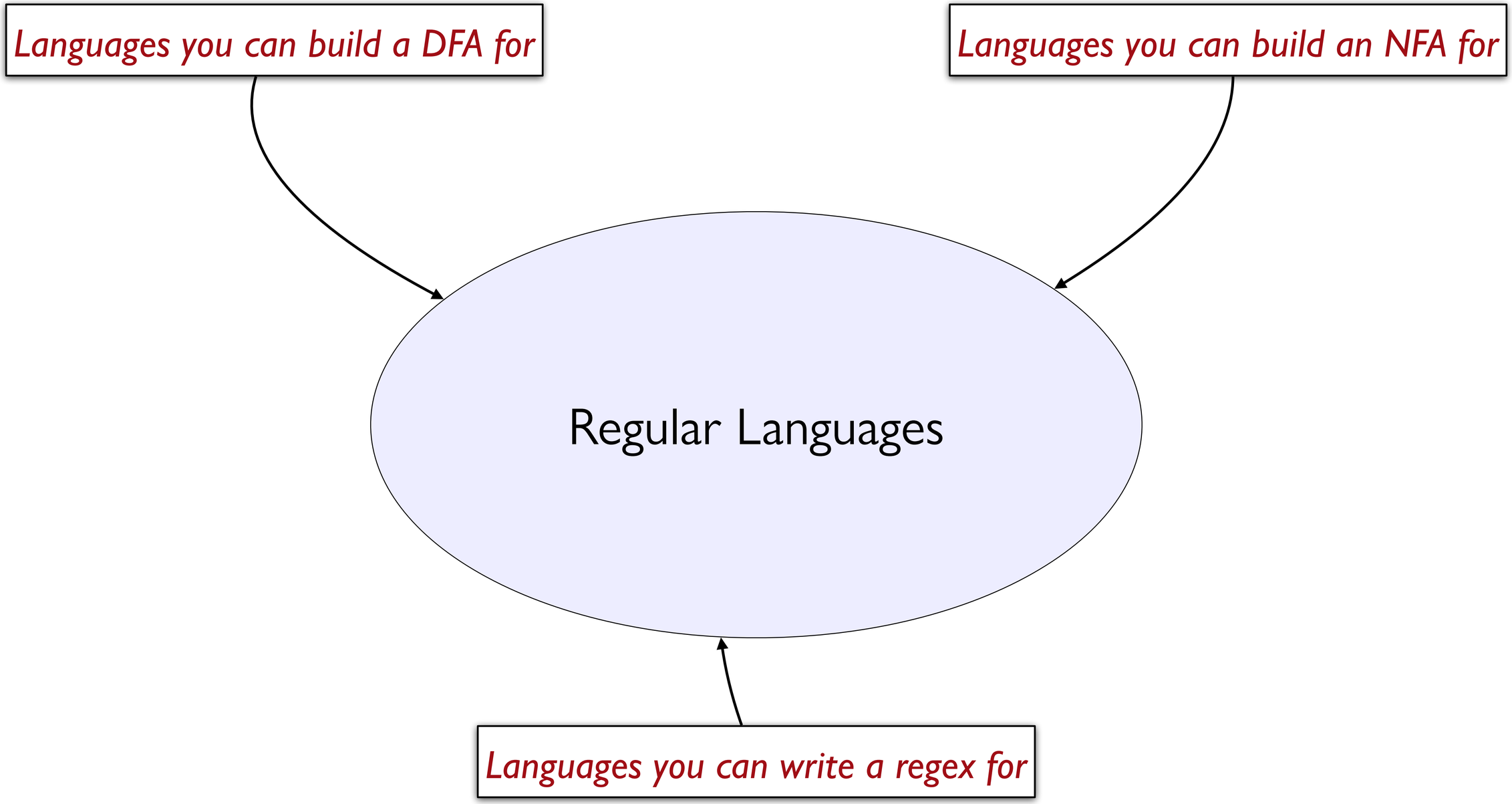

Languages you can build a DFA for
Languages you can build an NFA for
Regular Languages


Languages you can build a DFA for
Languages you can build an NFA for
Regular Languages
Languages you can write a regex for


Languages you can build a DFA for
Languages you can build an NFA for
Regular Languages
Languages you can write a regex for


Languages you can build a DFA for
Languages you can build an NFA for
Regular Languages
Languages you can
write a regex for


Languages you can build a DFA for
Languages you can build an NFA for
Regular Languages
Languages you can write a regex for


Languages you can build a DFA for
Languages you can build an NFA for
Regular Languages
Languages you can write a regex for

THEOREM If R is a regular expression, then $L(R)$ is regular.

THEOREM If R is a regular expression, then $L(R)$ is regular.

PROOF IDEA Use induction!

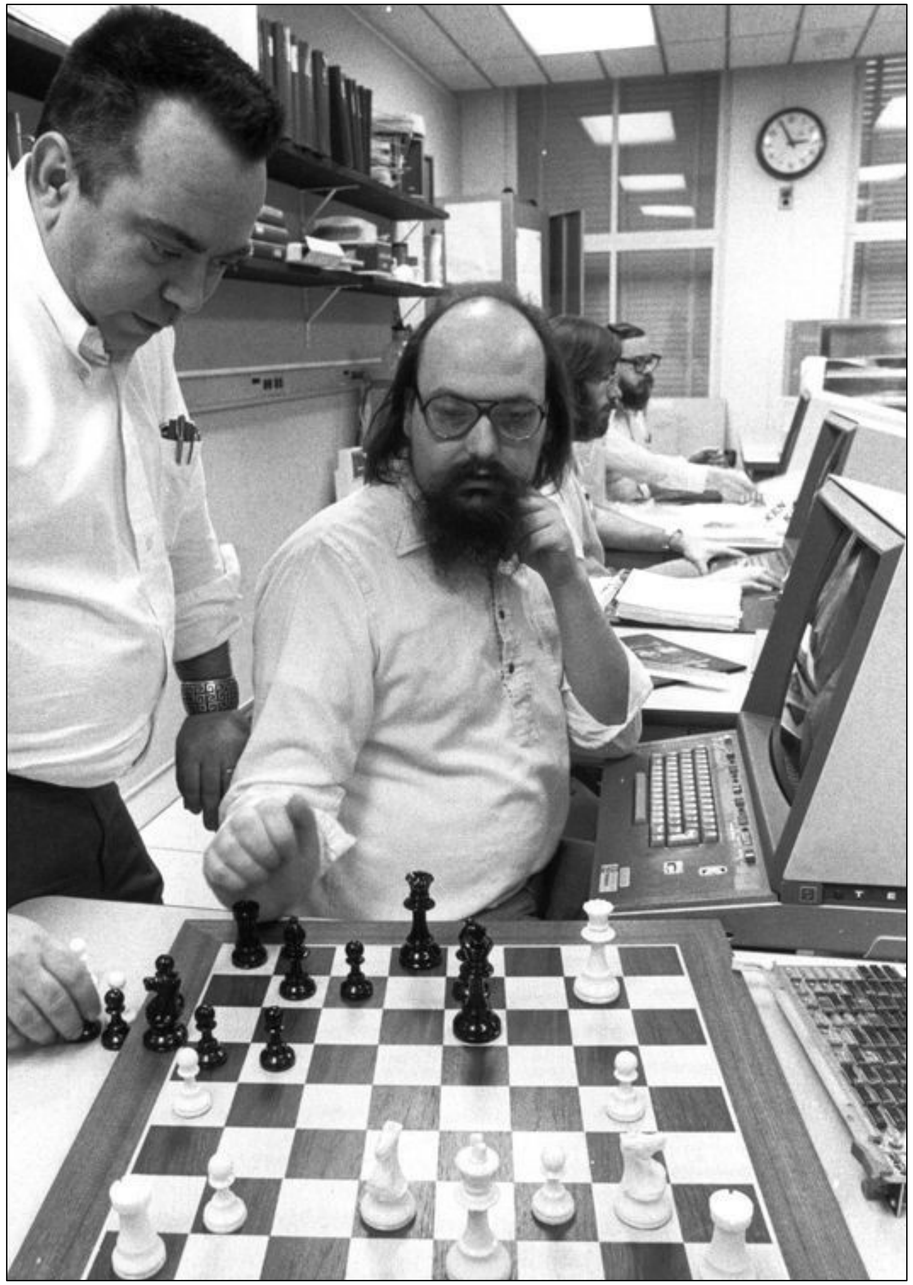
The atomic regular expressions all represent regular languages.

The combination steps represent closure properties.

So, anything you can make from them must be regular!

In practice, many regex matchers – including **`grep`** – use an algorithm called *Thompson's algorithm* to convert regular expressions into equivalent finite automata.

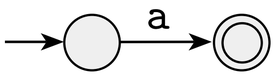
The "Thompson" is computing pioneer Ken Thompson, a co-inventor of Unix.



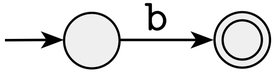
Example

(abua)*

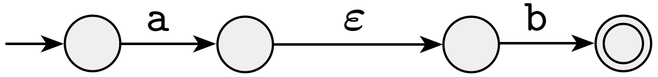
a



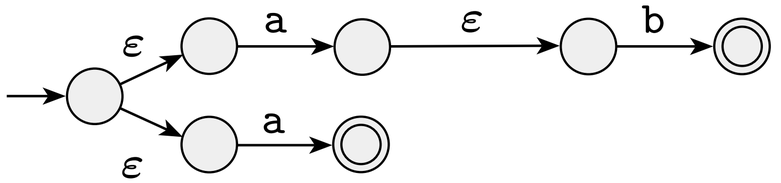
b



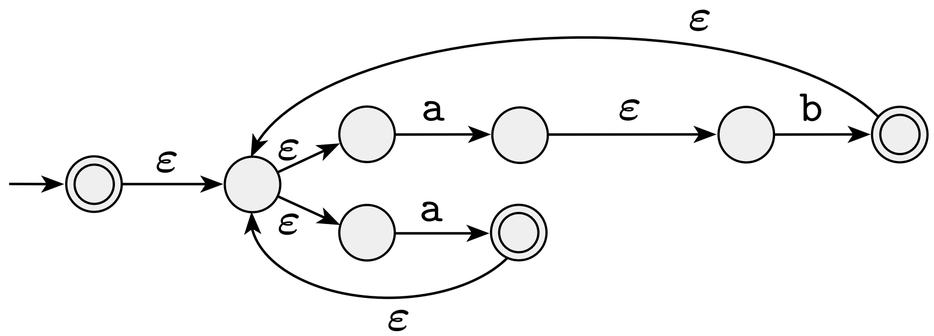
ab



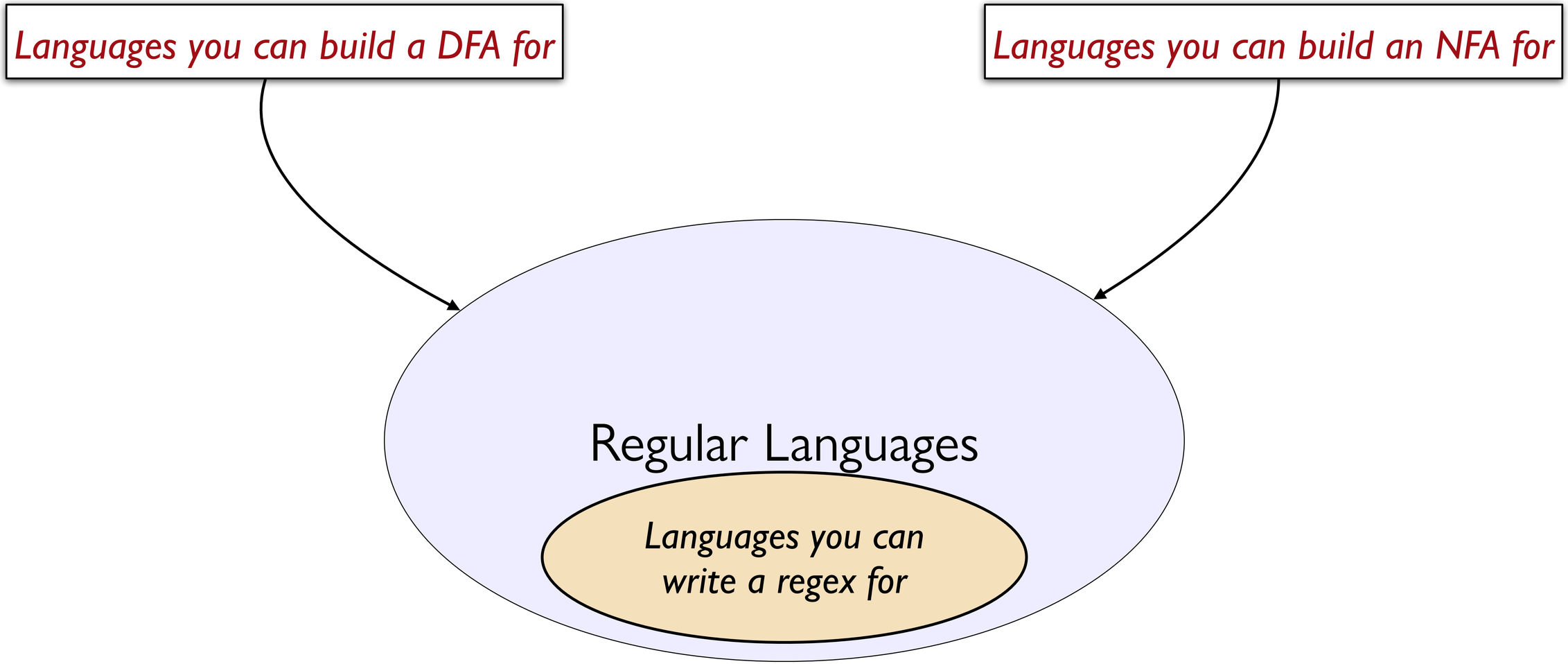
ab ∪ a

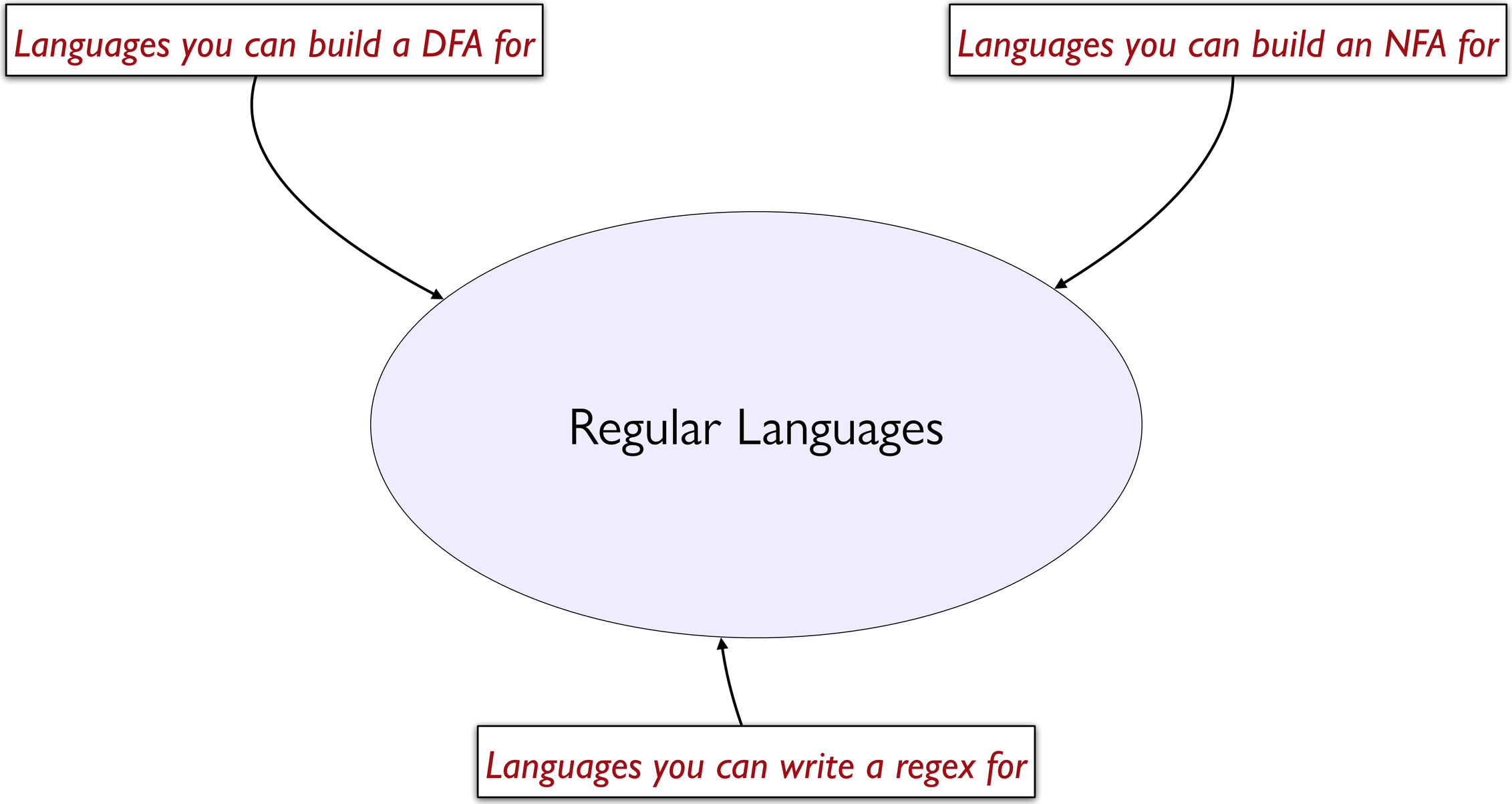


(ab ∪ a)*



Solution from Sipser


Languages you can build a DFA for
Languages you can build an NFA for
Regular Languages
Languages you can write a regex for

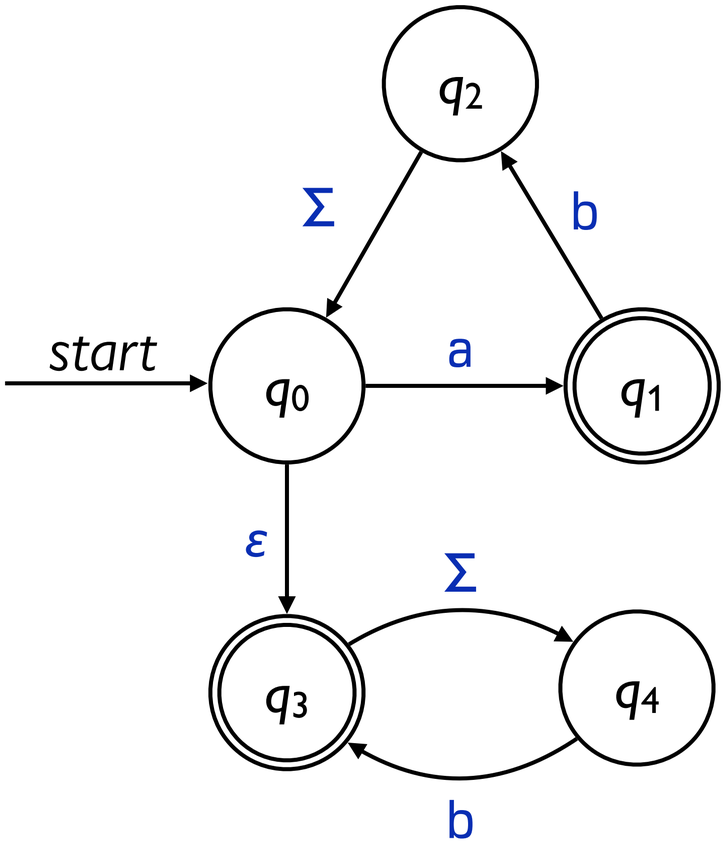

Languages you can build a DFA for
Languages you can build an NFA for
Regular Languages
Languages you can write a regex for

THEOREM If L is a regular language, then there is a regular expression for L.

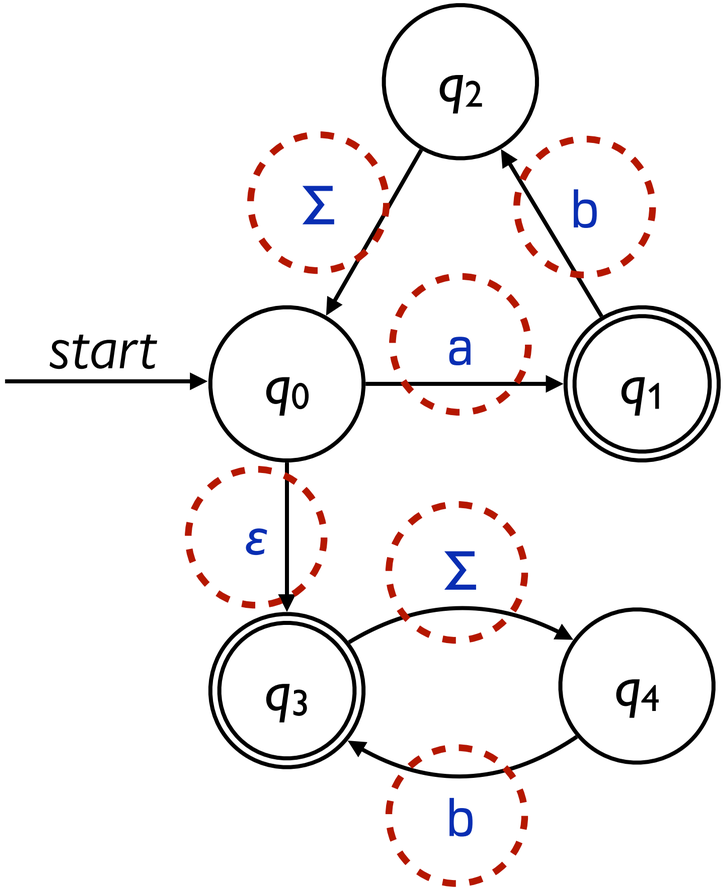
PROOF IDEA Show how to convert an arbitrary NFA into a regular expression.

This is not obvious!

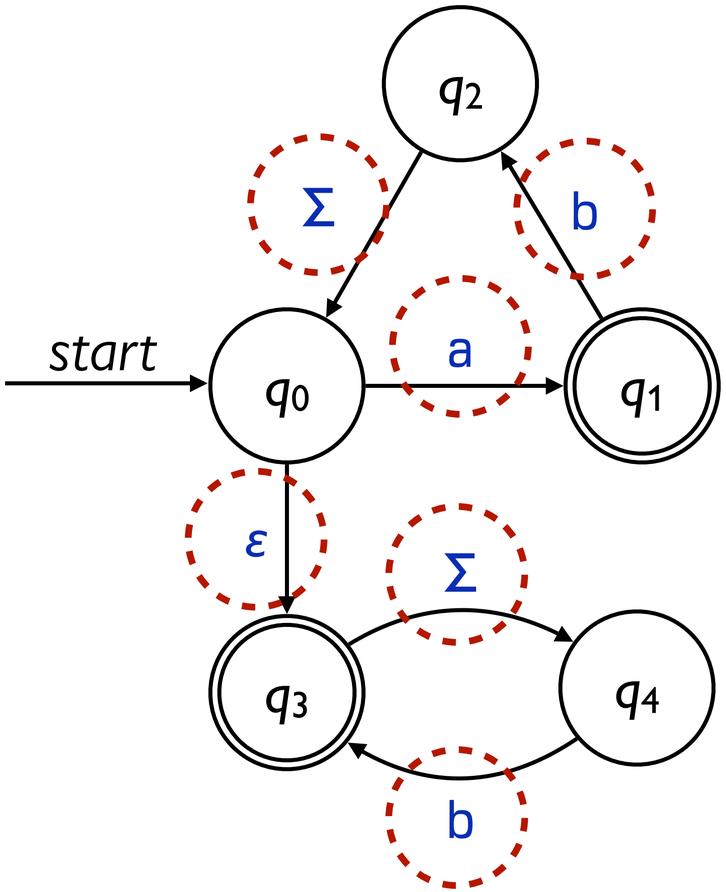
Generalizing NFAs



Generalizing NFAs

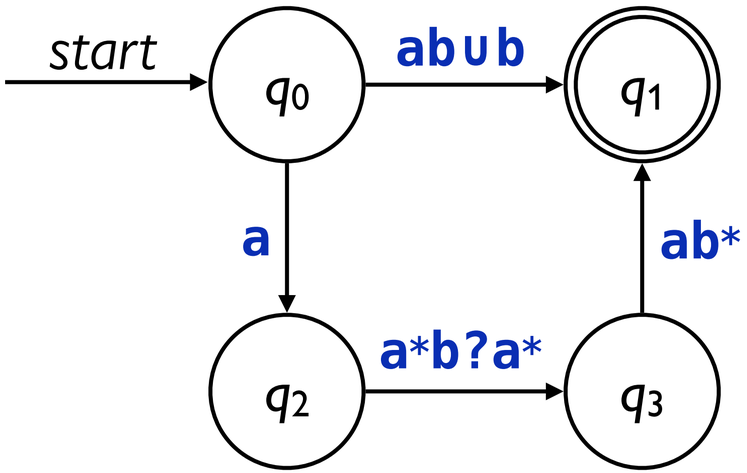


Generalizing NFAs



These are all regular expressions!

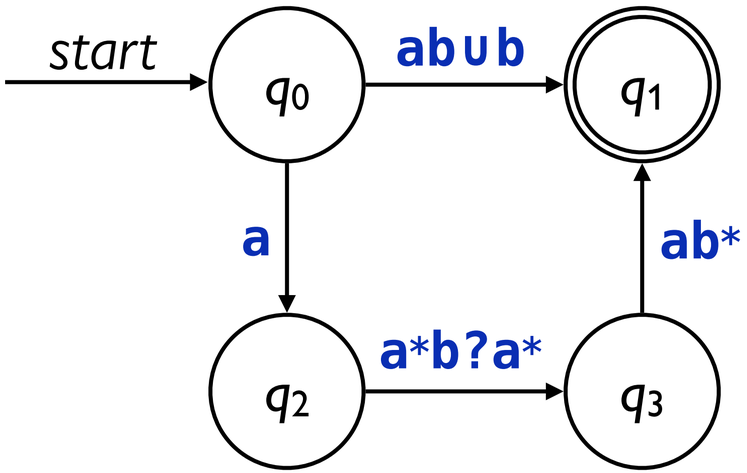
Generalizing NFAs



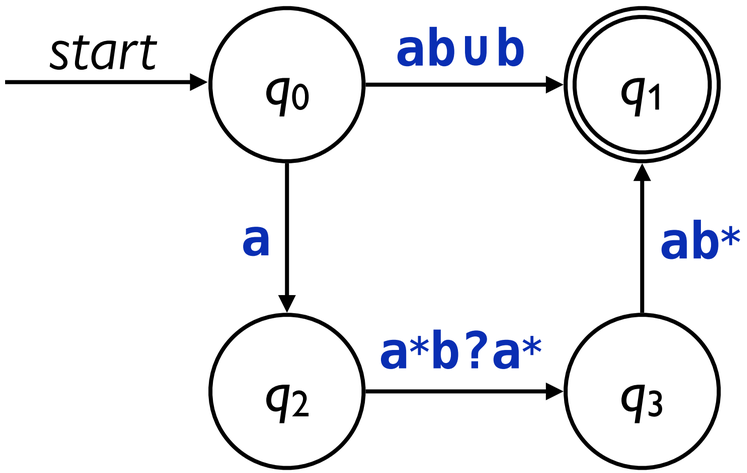
Note: NFAs **aren't** *allowed to have transitions like these.*

This is just a thought experiment.

Generalizing NFAs

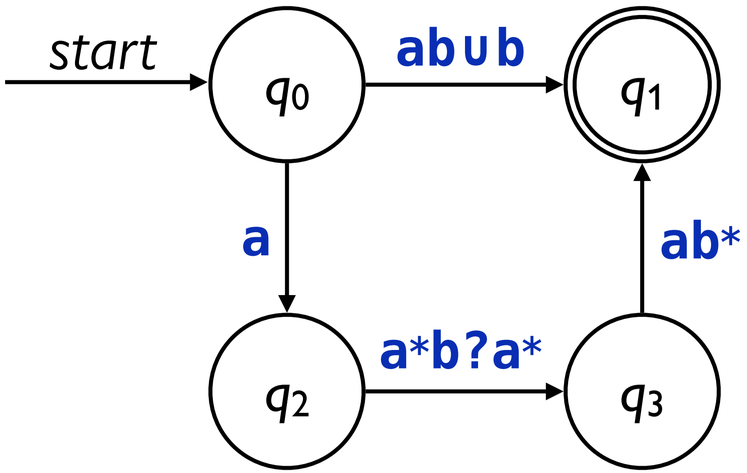


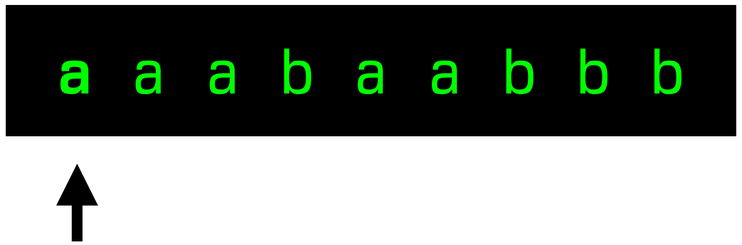
Generalizing NFAs



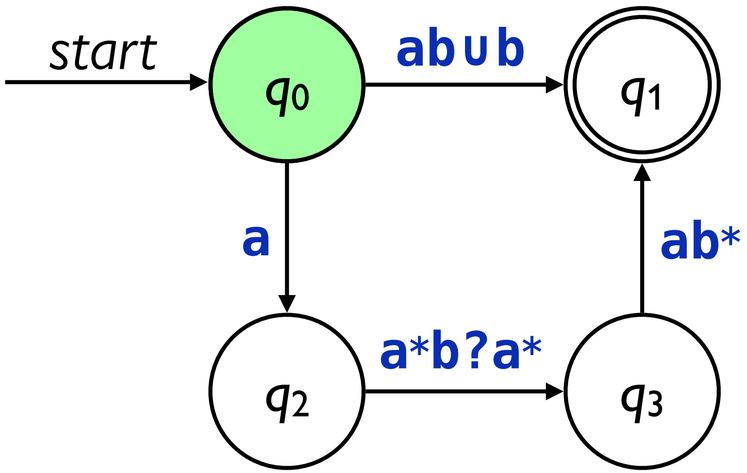
a a a b a a b b b

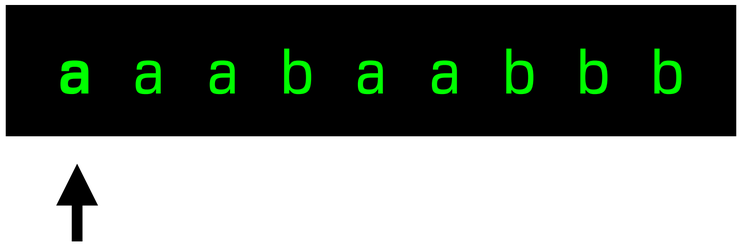
Generalizing NFAs



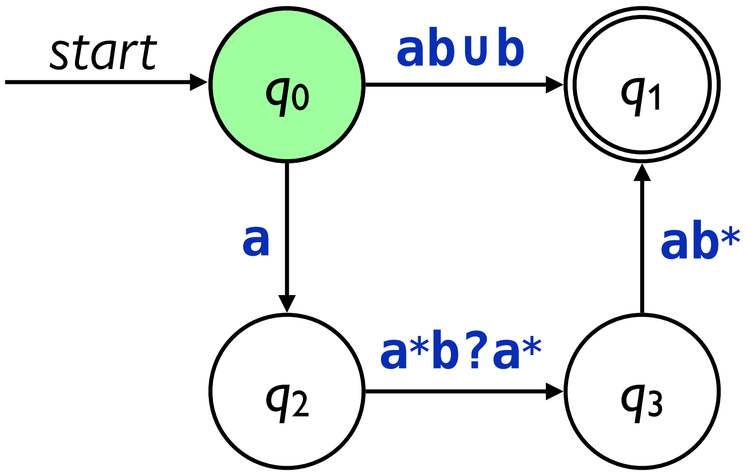


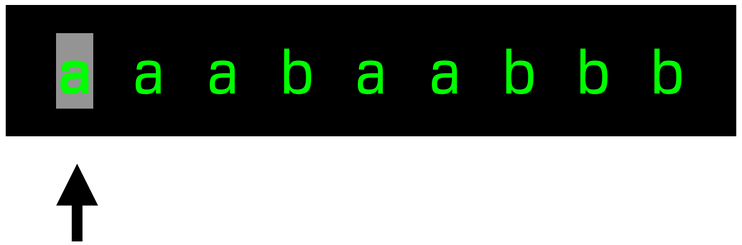
Generalizing NFAs



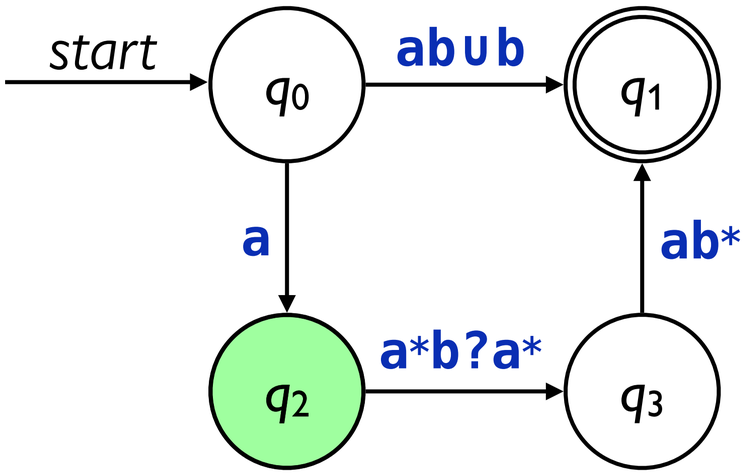


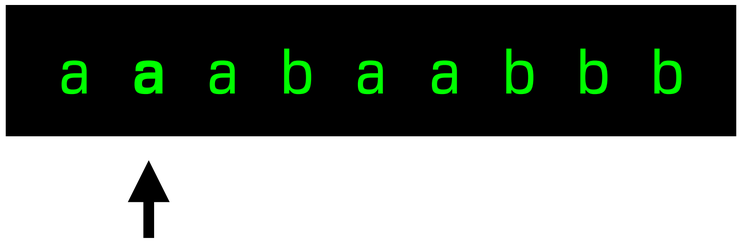
Generalizing NFAs



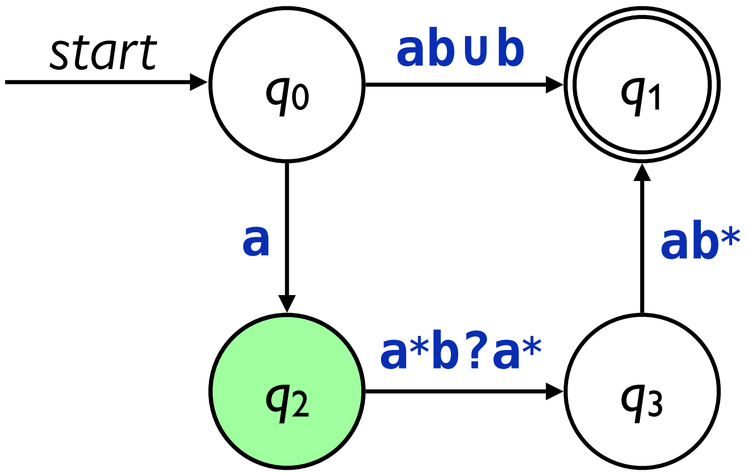


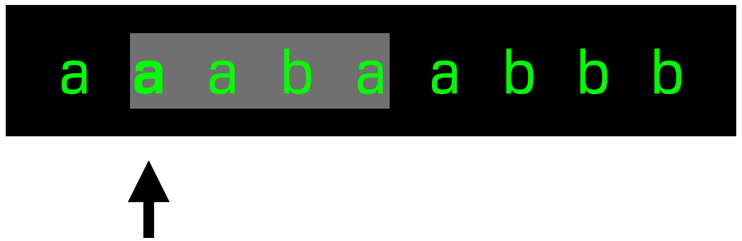
Generalizing NFAs



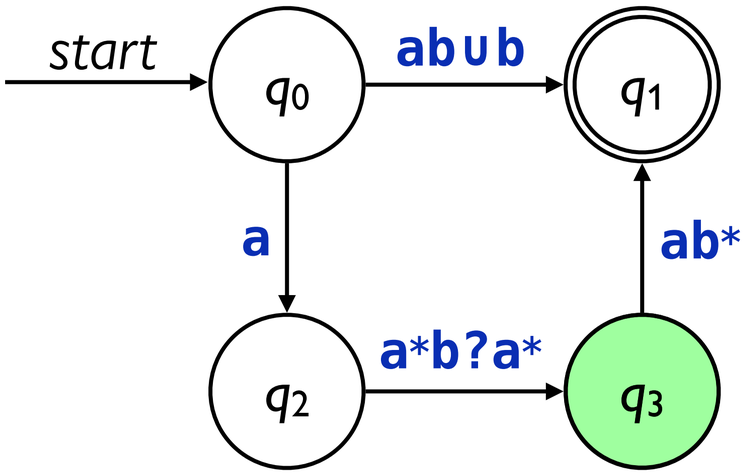


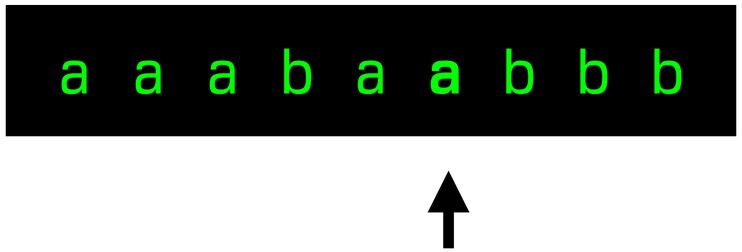
Generalizing NFAs



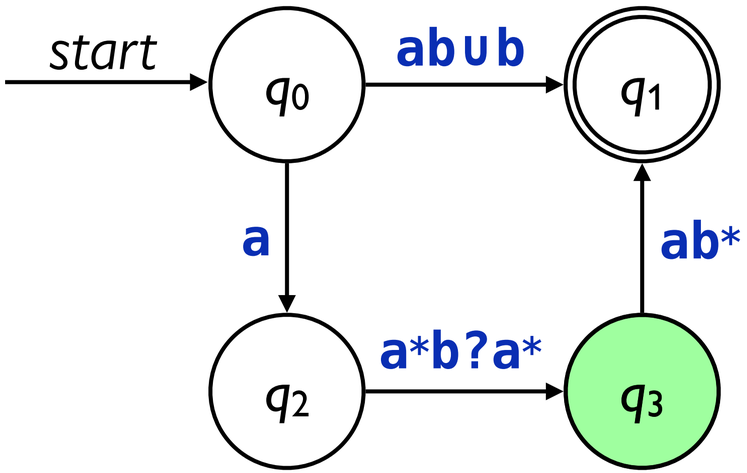


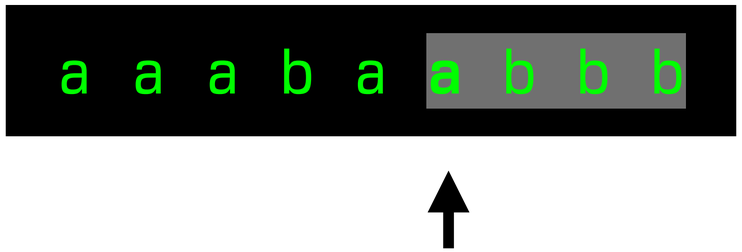
Generalizing NFAs



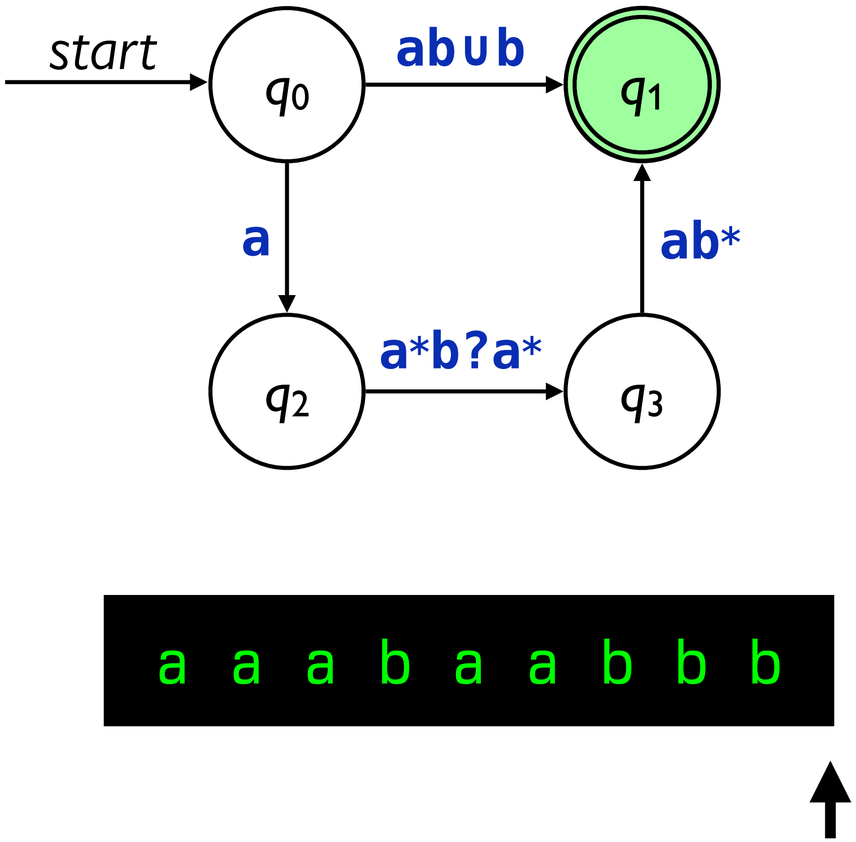


Generalizing NFAs



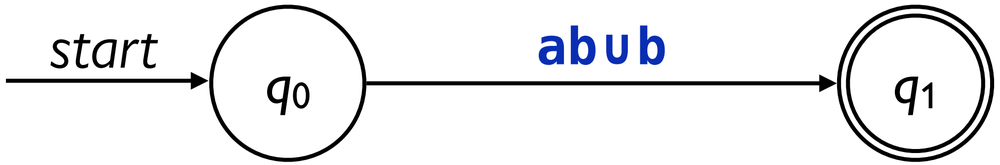


Generalizing NFAs



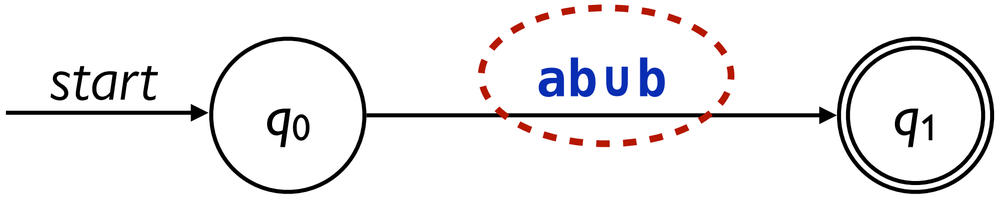
Key idea 1: Imagine that we can label transitions in an NFA with arbitrary regular expressions.

Generalizing NFAs



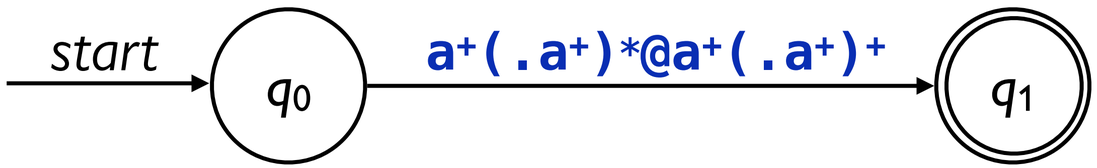
Is there a simple regular expression for the language of this generalized NFA?

Generalizing NFAs



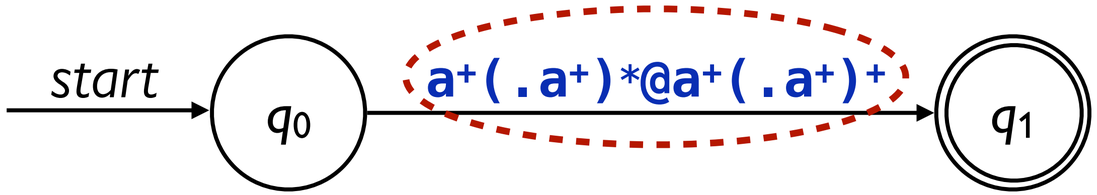
Is there a simple regular expression for the language of this generalized NFA?

Generalizing NFAs



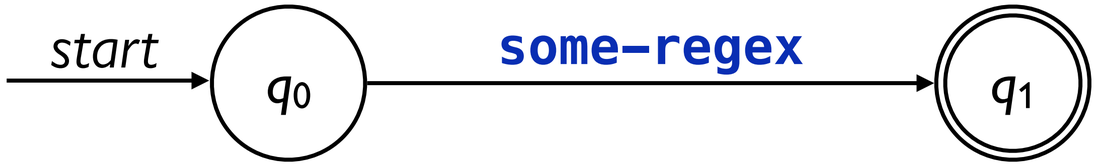
Is there a simple regular expression for the language of this generalized NFA?

Generalizing NFAs



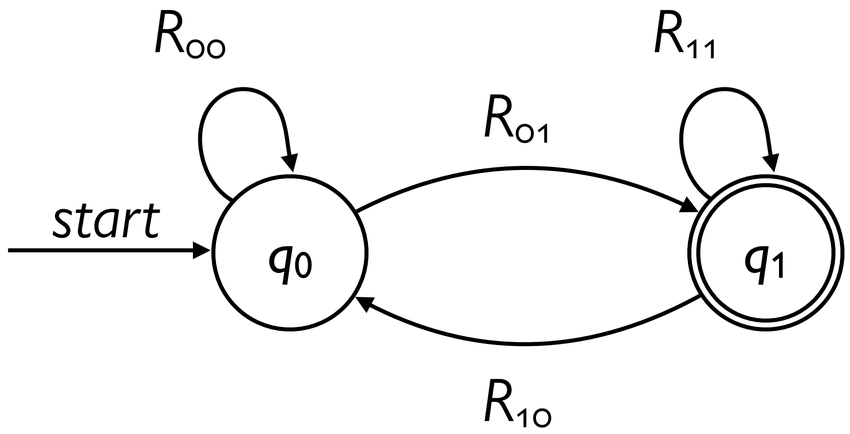
Is there a simple regular expression for the language of this generalized NFA?

Key idea 2: If we can convert an NFA into a generalized NFA that looks like this,



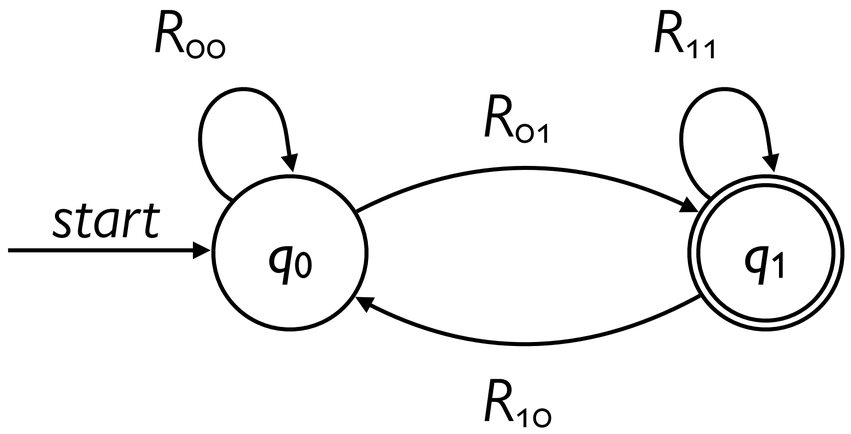
then we can easily read off a regular expression for the original NFA.

From GNFAs to regular expressions



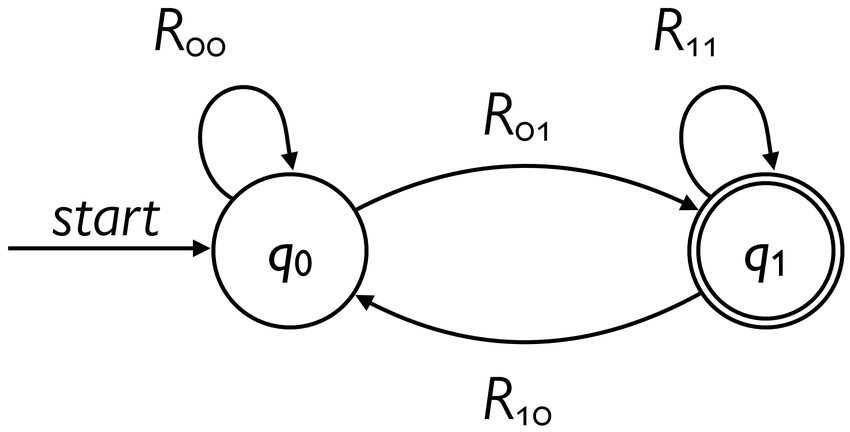
R_{00}, R_{01}, R_{11}, and R_{10} are variables for arbitrary regular expressions.

From GNFAs to regular expressions

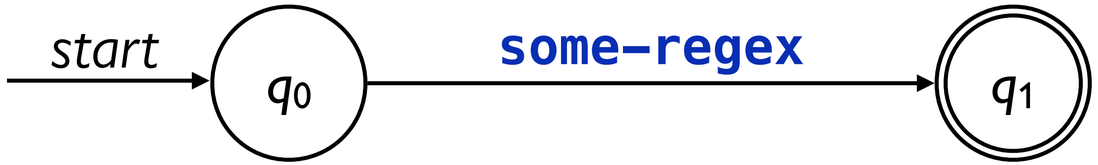


Can we get a clean regular expression from this NFA?

From GNFAs to regular expressions

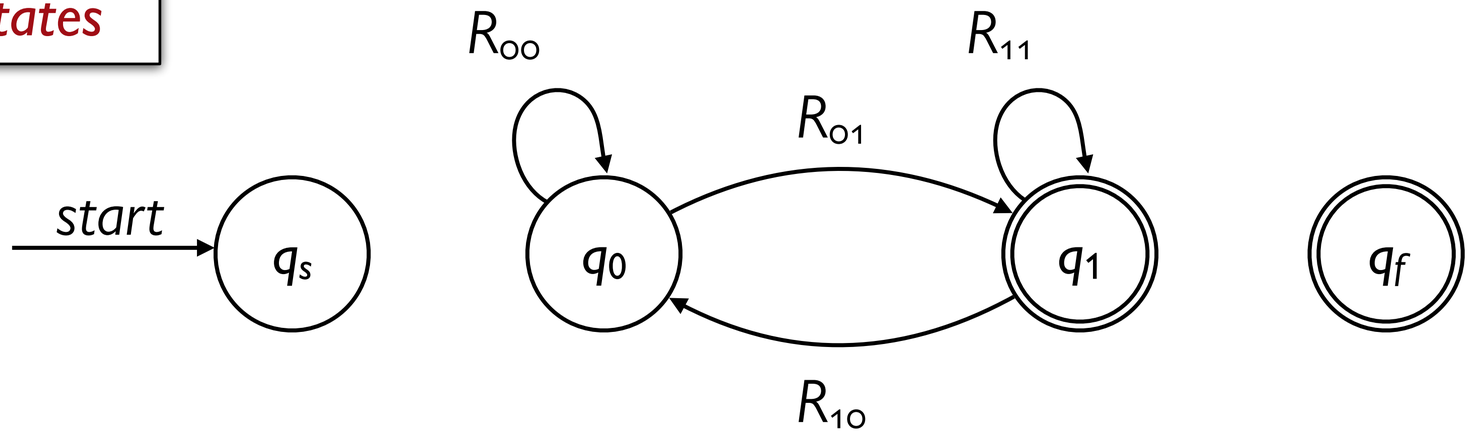


Key idea 3: Transform a GNFA so it looks like this:



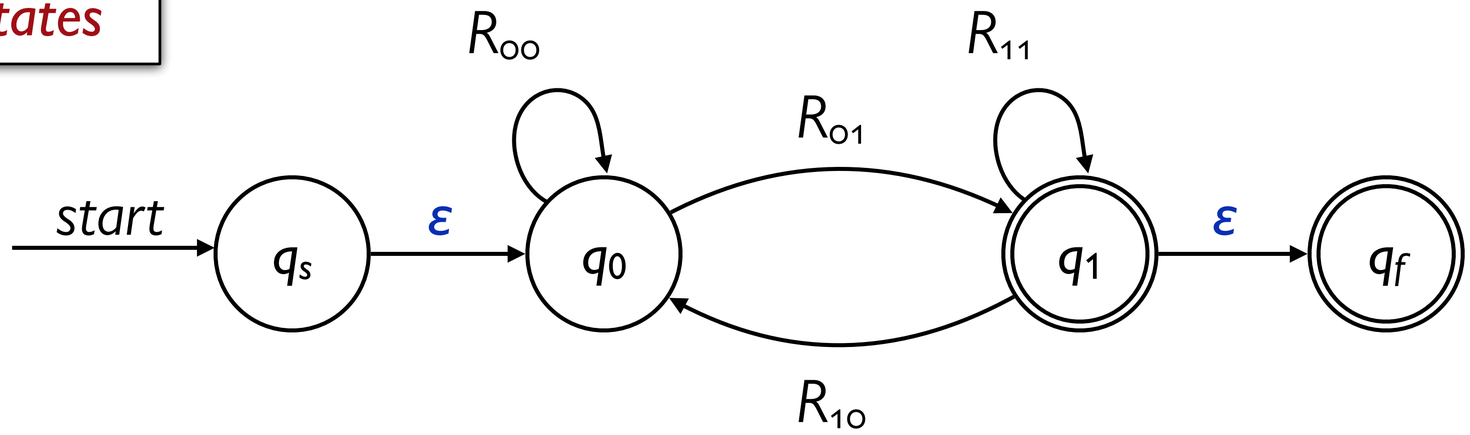
From GNFAs to regular expressions

First add new start and accept states



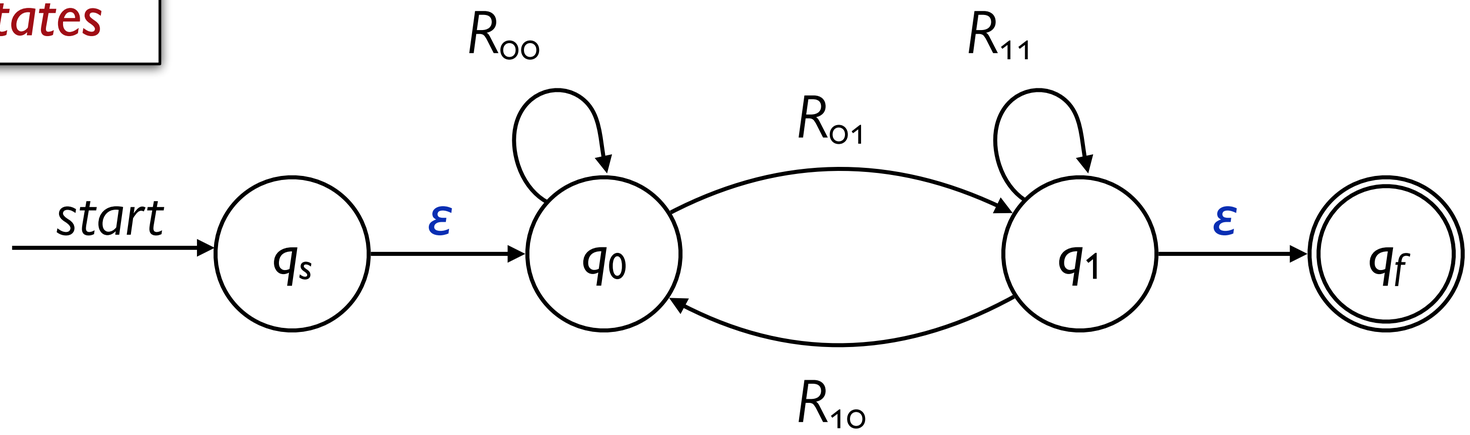
From GNFAs to regular expressions

First add new start and accept states

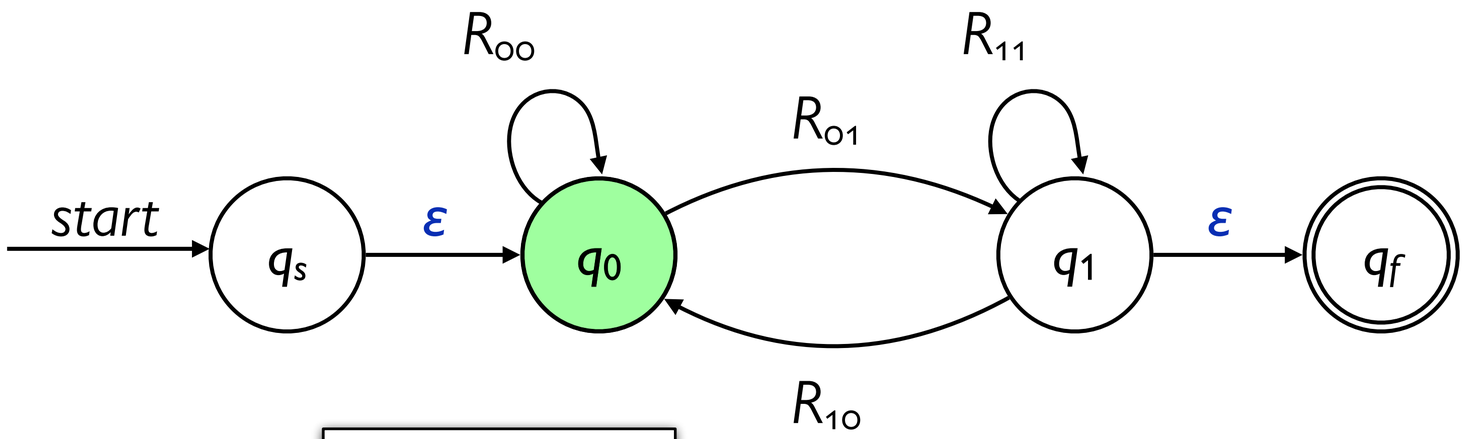


From GNFAs to regular expressions

First add new start and accept states

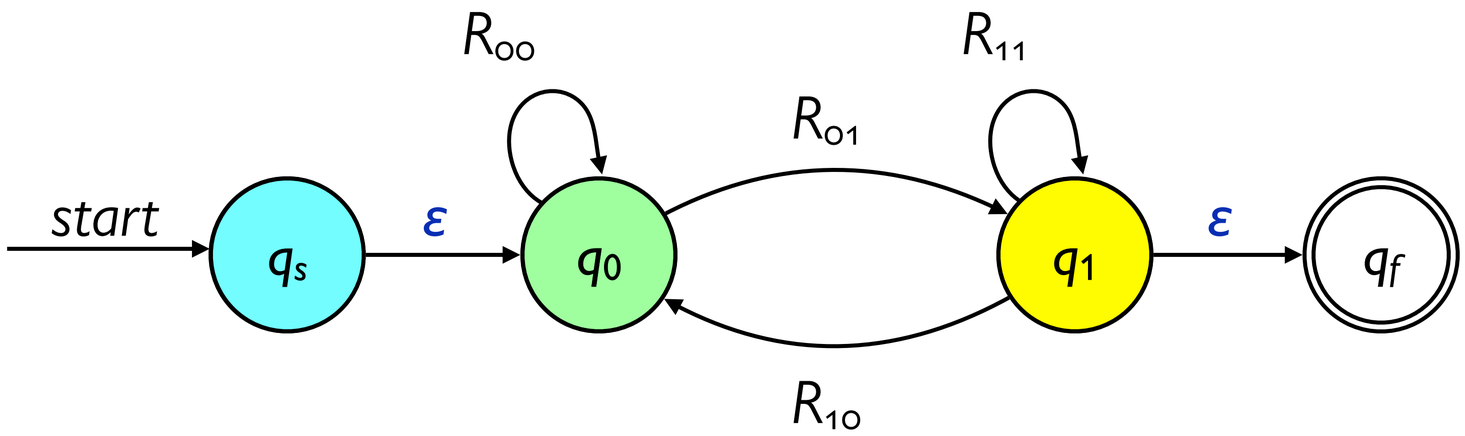


From GNFAs to regular expressions

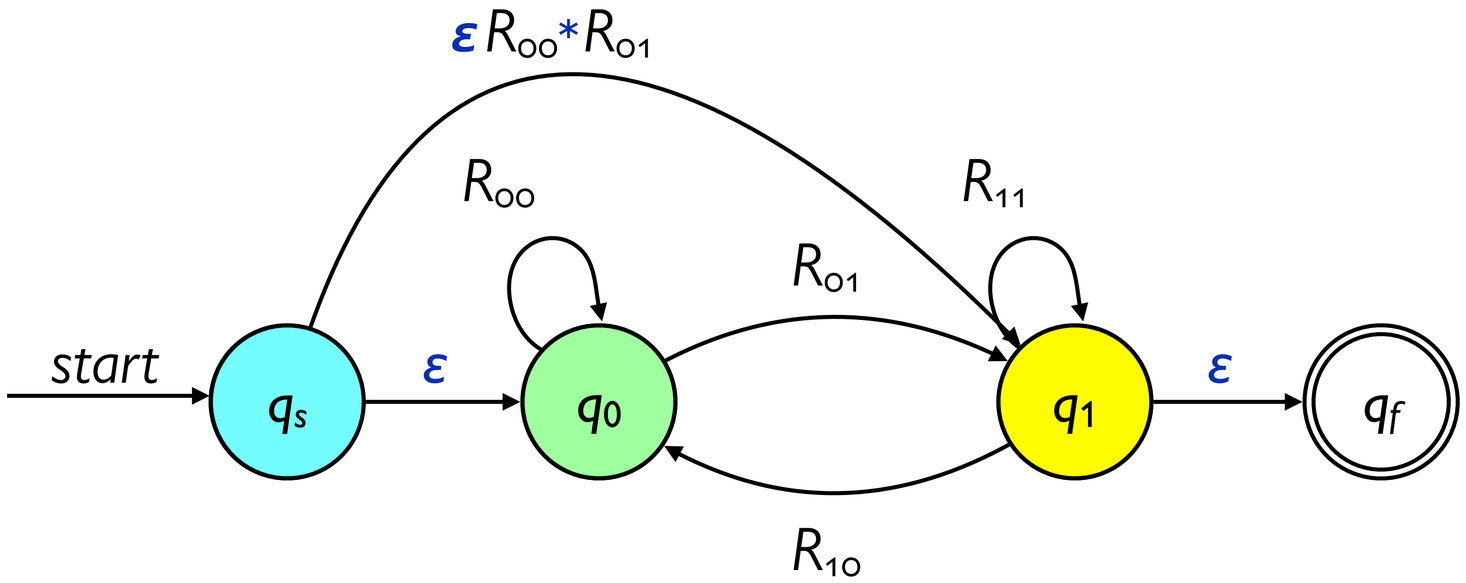


Could we eliminate this state from the GNFA?

From GNFAs to regular expressions

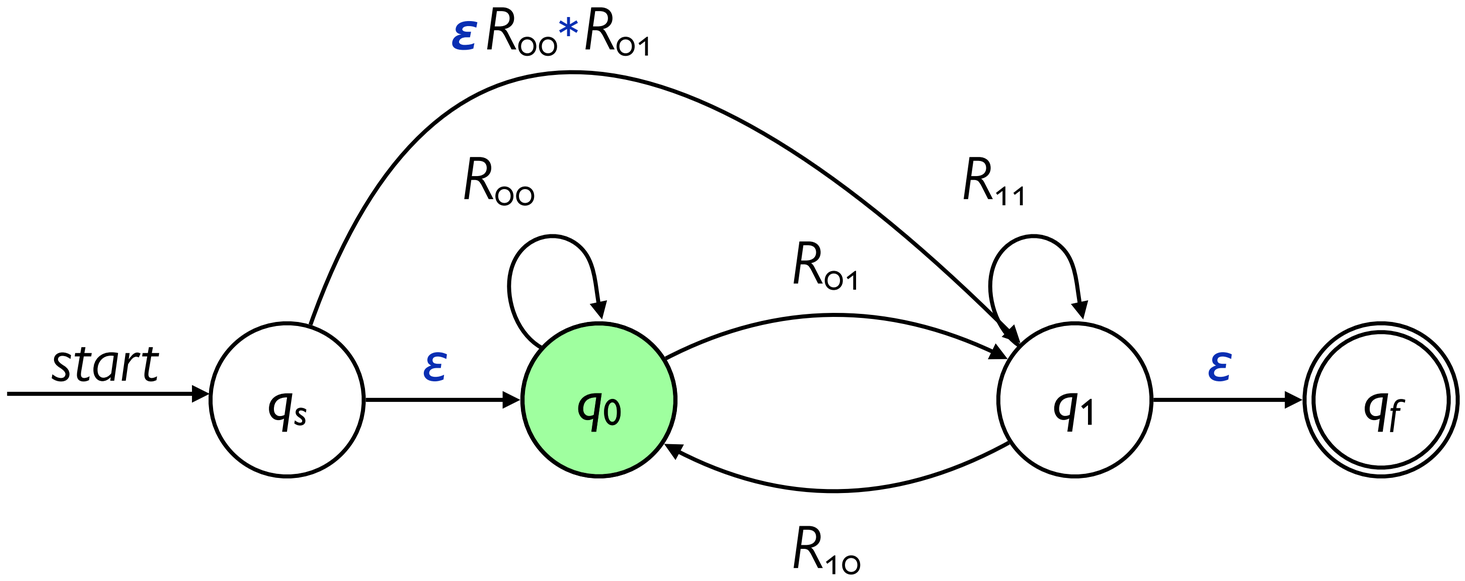


From GNFAs to regular expressions

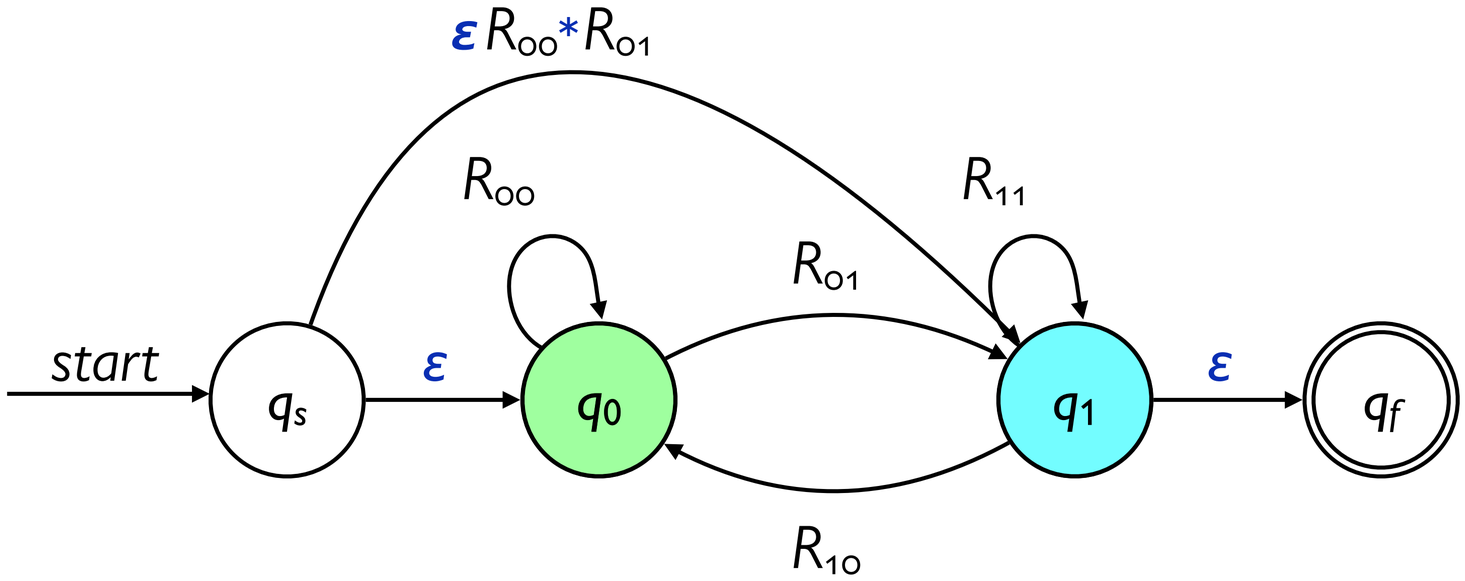


We can use concatenation and Kleene closure to skip this state.

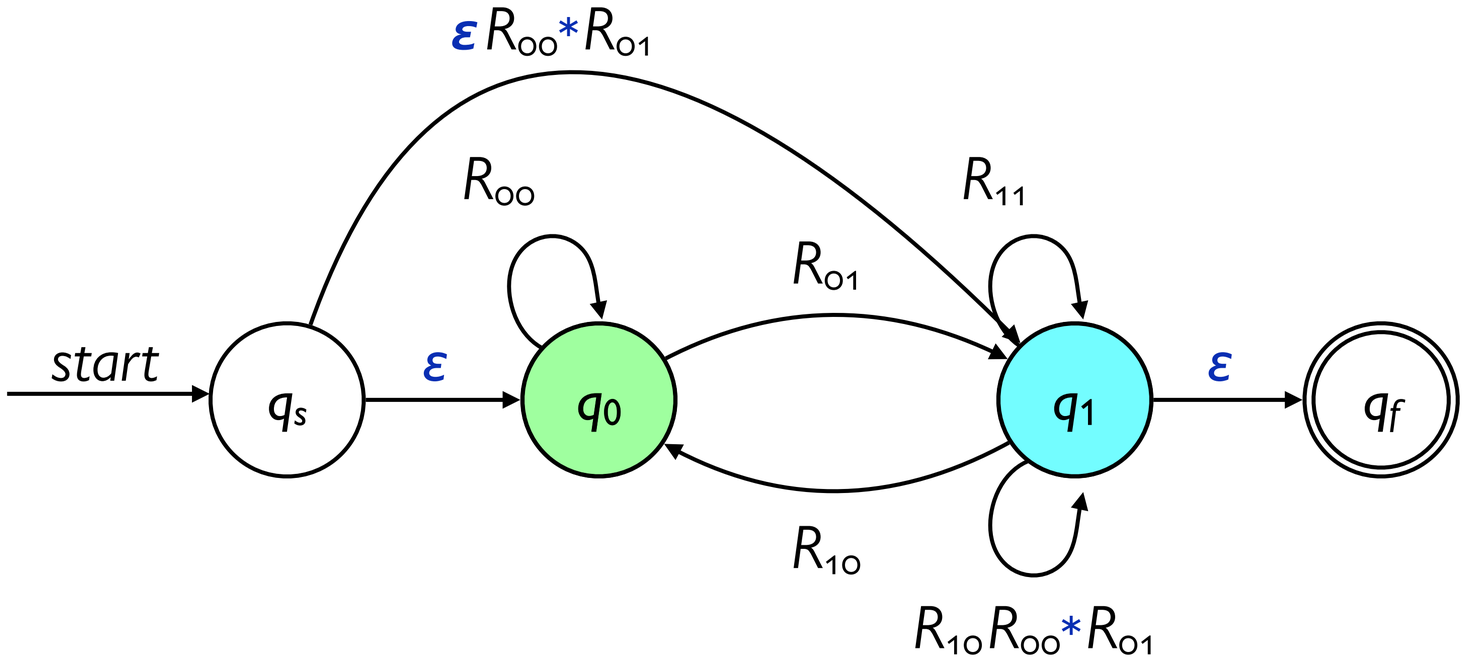
From GNFAs to regular expressions



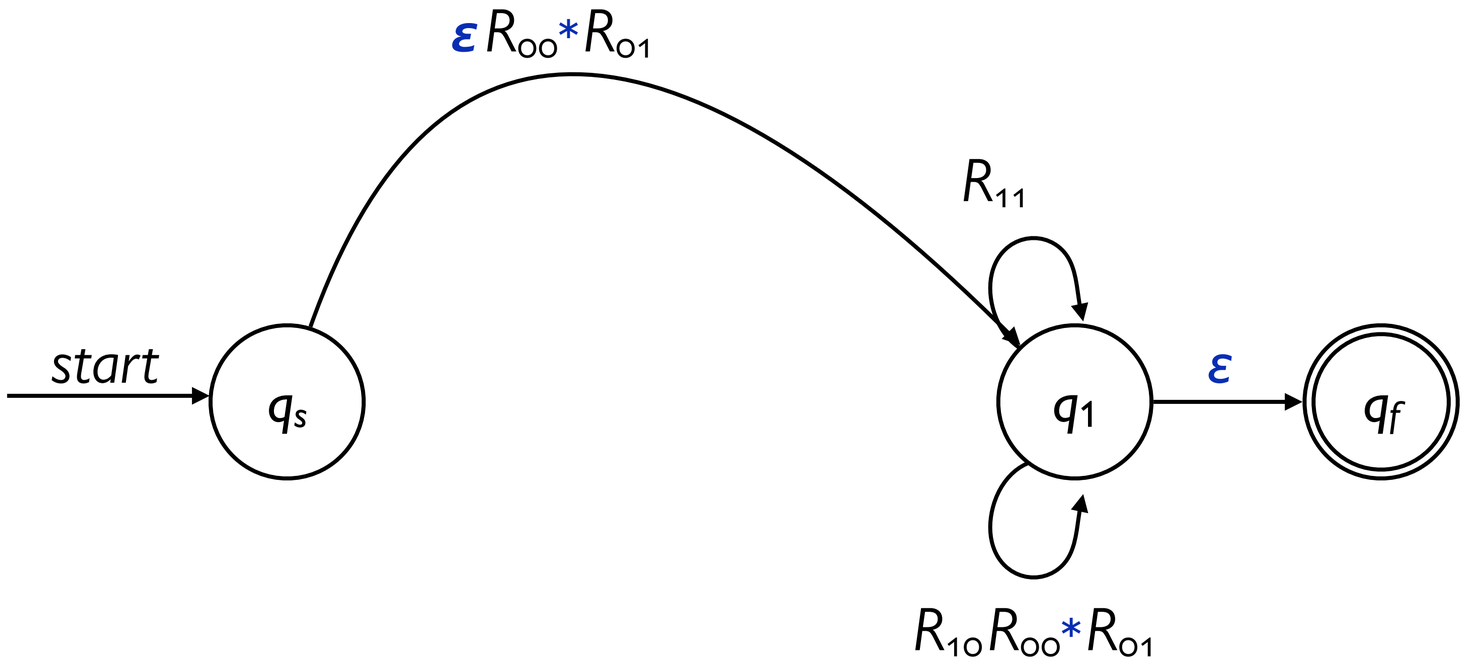
From GNFAs to regular expressions



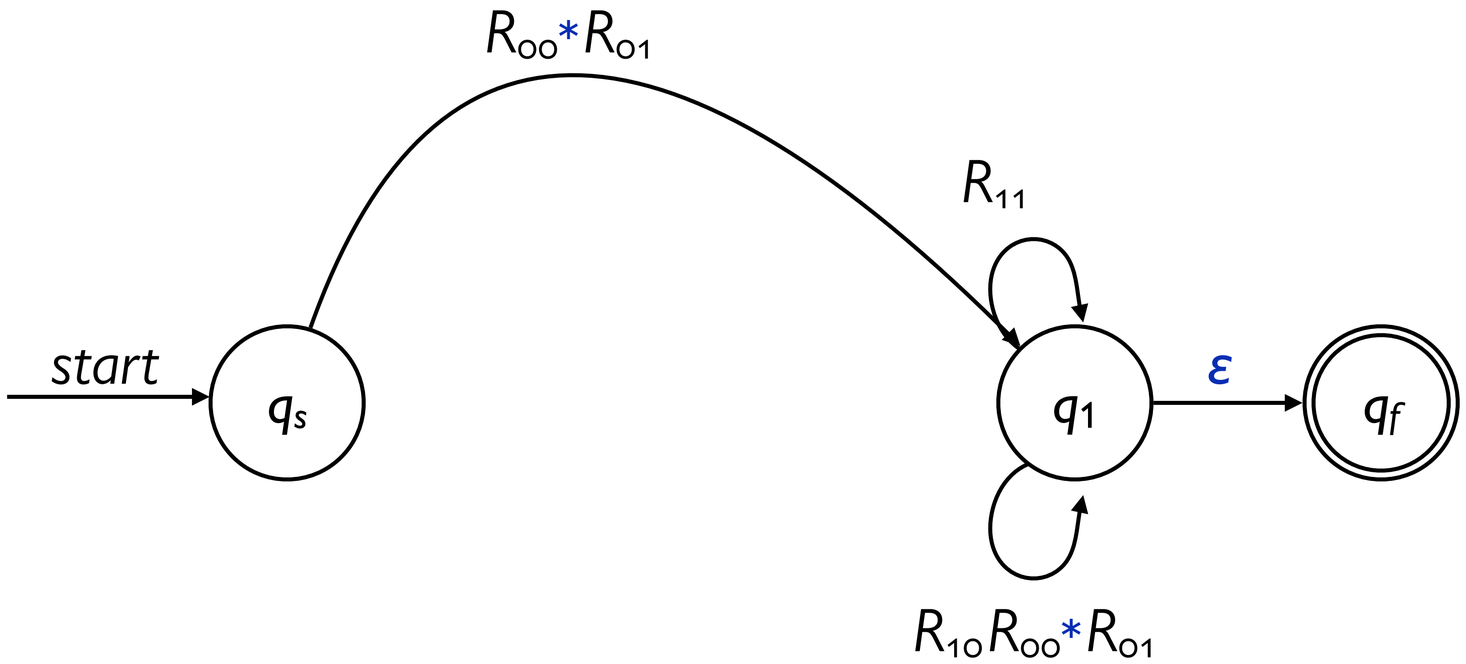
From GNFAs to regular expressions



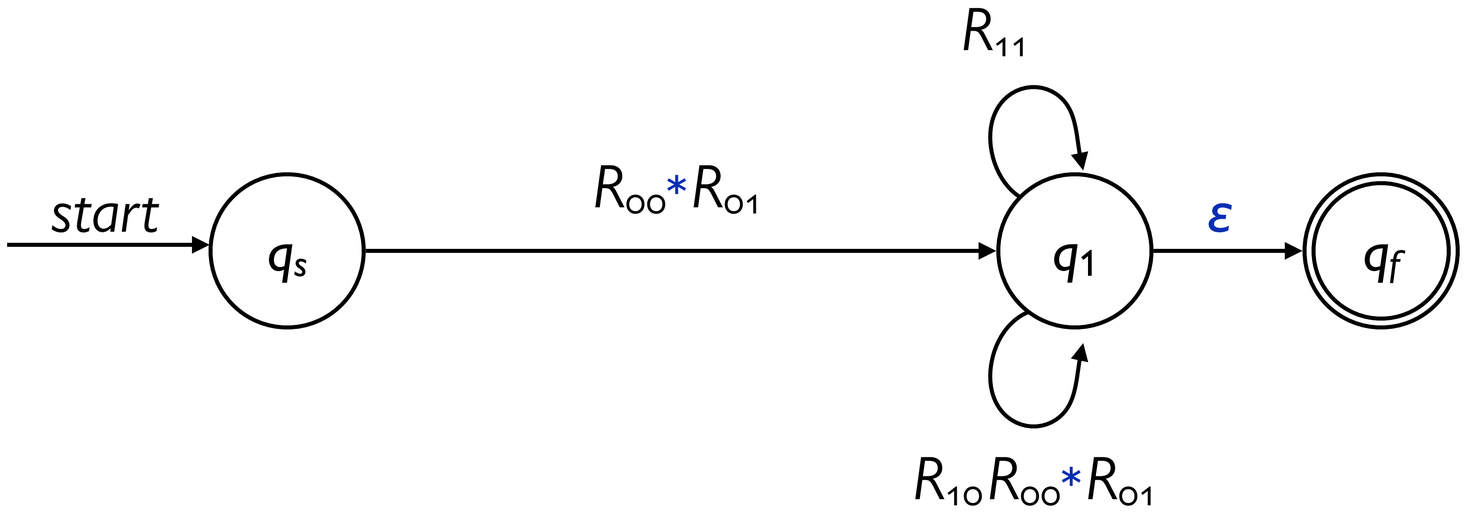
From GNFAs to regular expressions



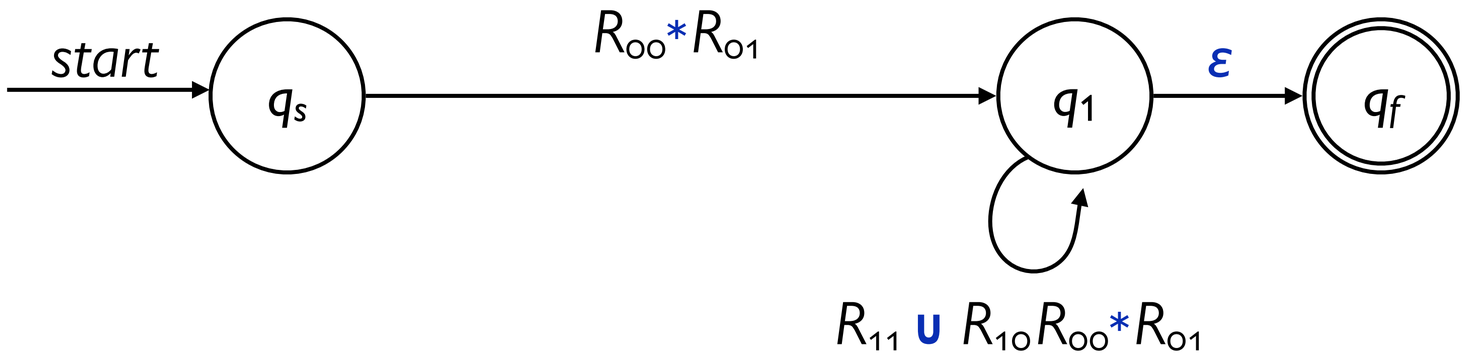
From GNFAs to regular expressions



From GNFAs to regular expressions



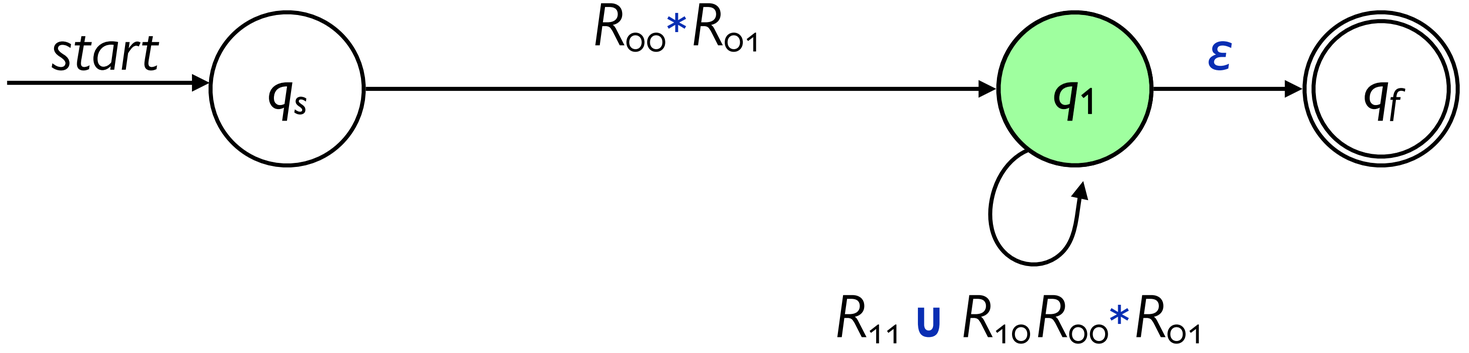
From GNFAs to regular expressions



We can use union to combine these transitions.

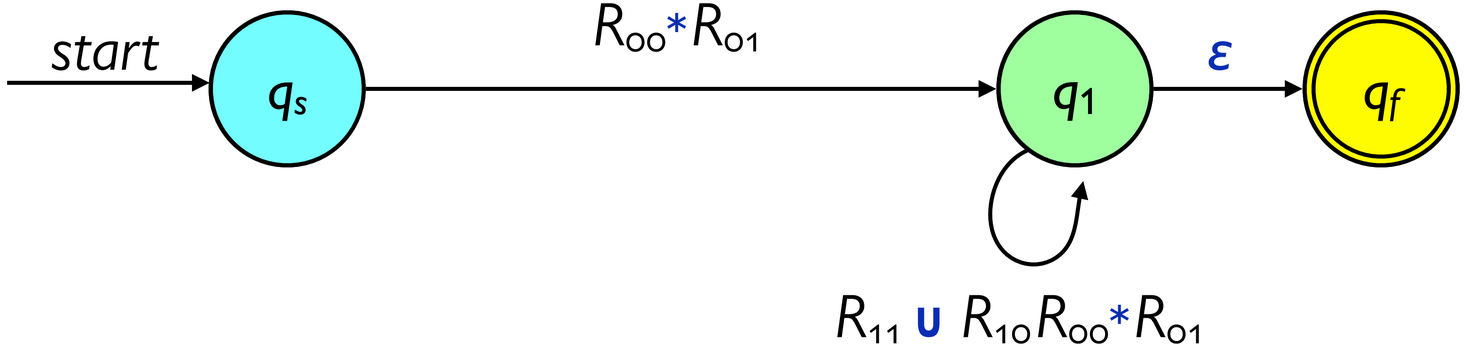
From GNFAs to regular expressions

Could we eliminate this state from the GNFA?

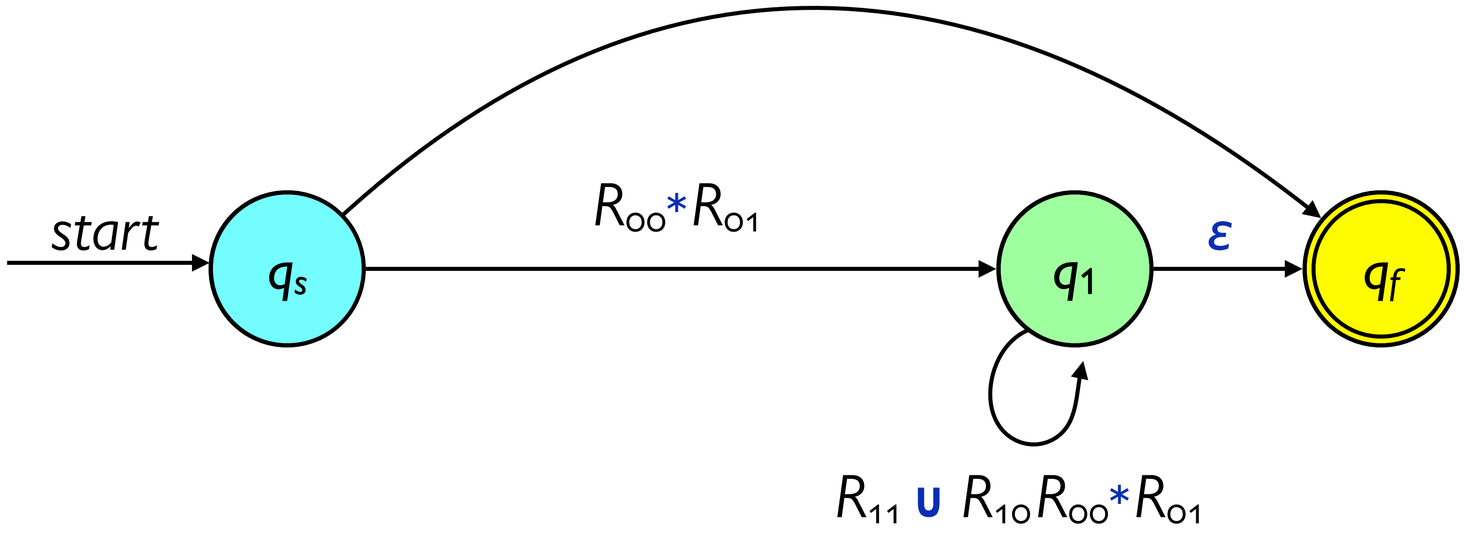


From GNFAs to regular expressions

Could we eliminate this state from the GNFA?

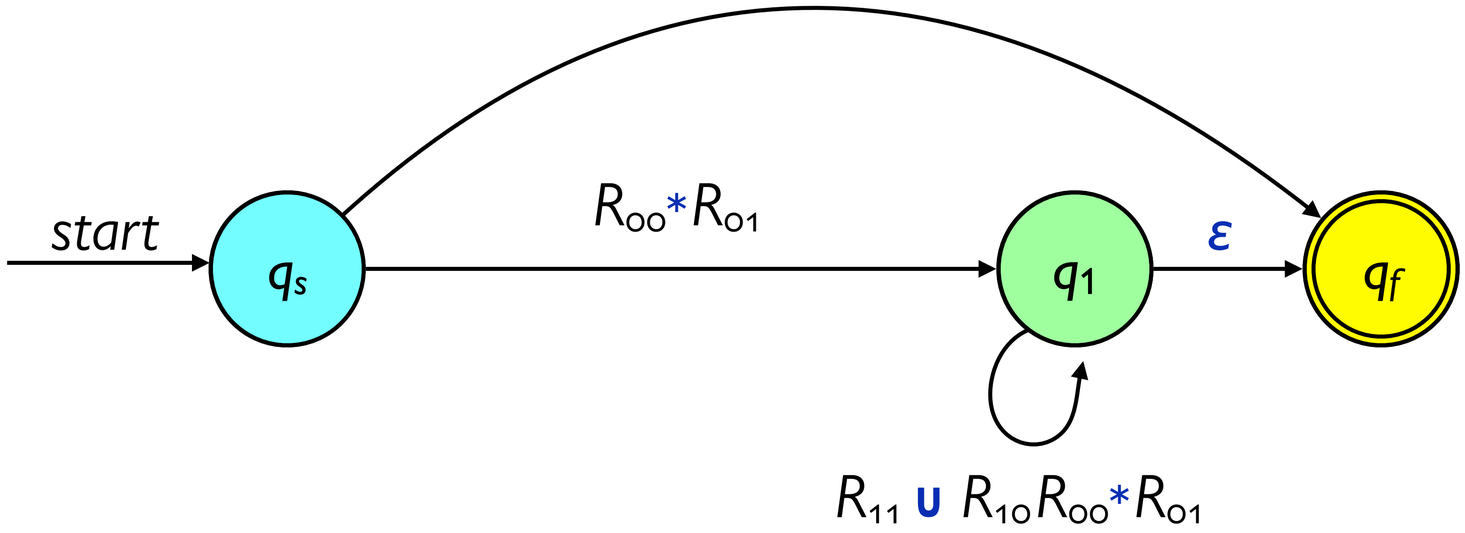


From GNFAs to regular expressions

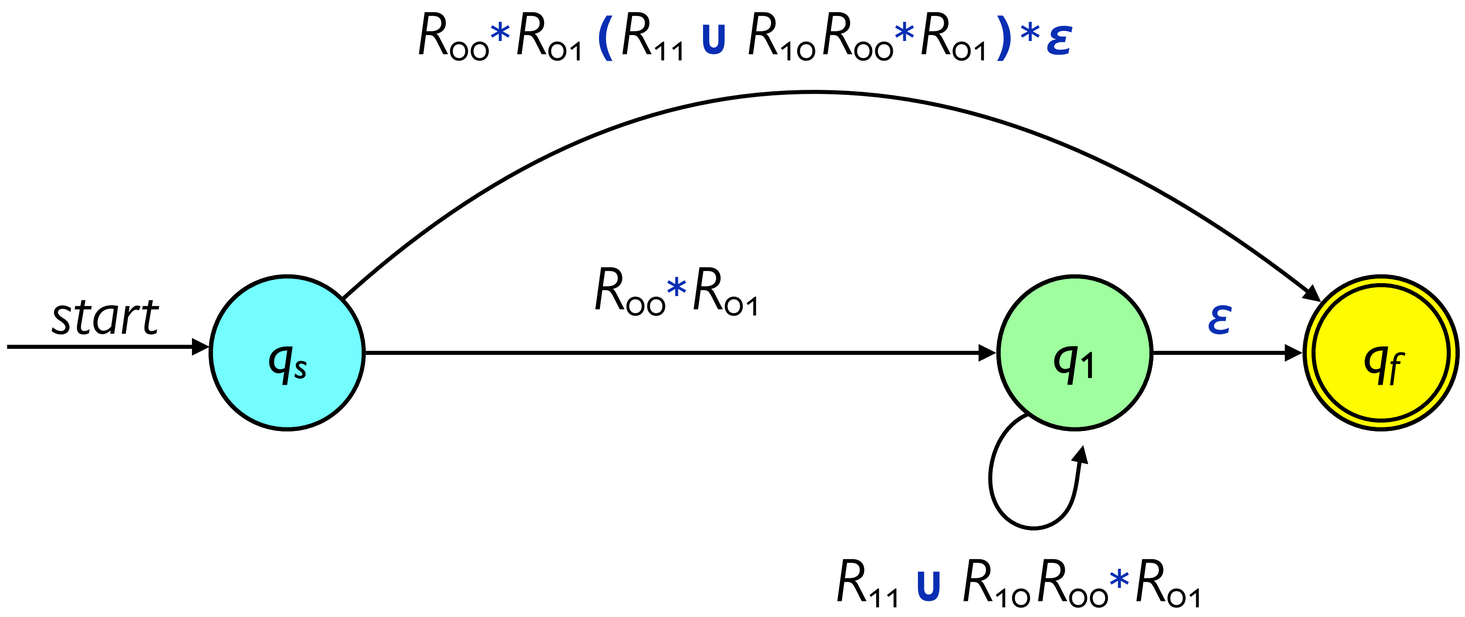


From GNFAs to regular expressions

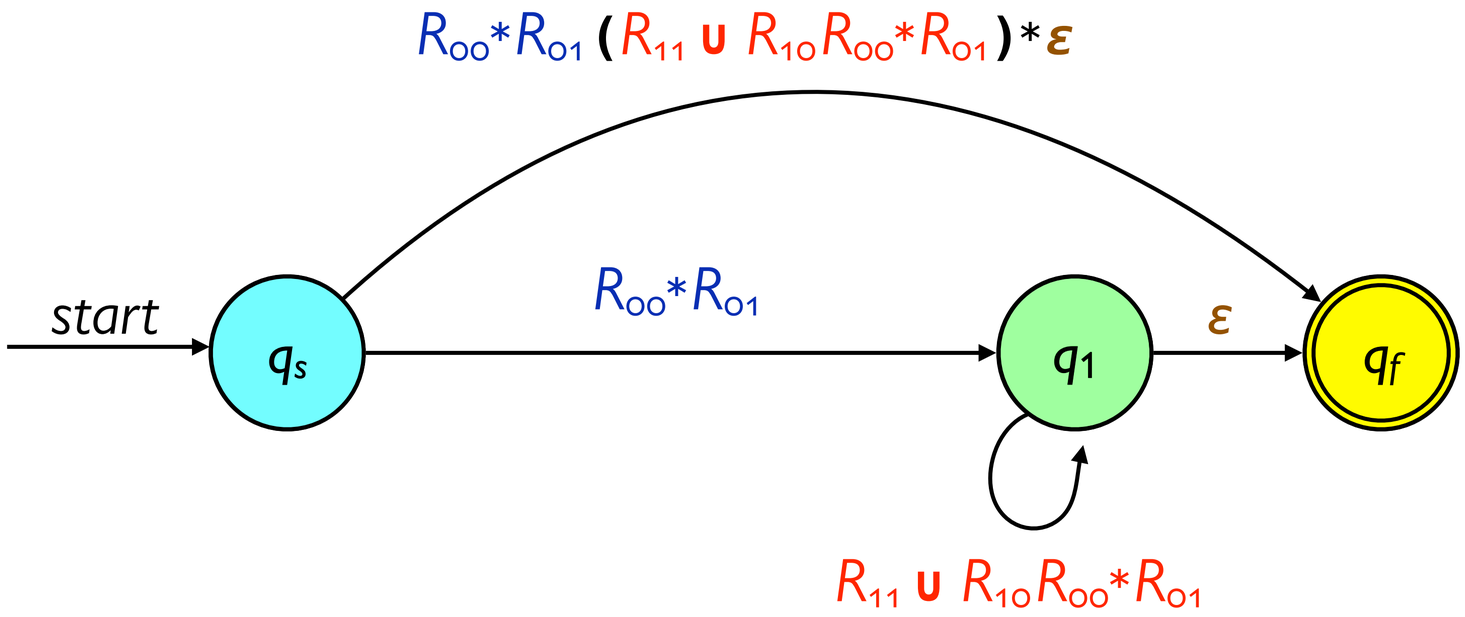
What should we put on this transition?



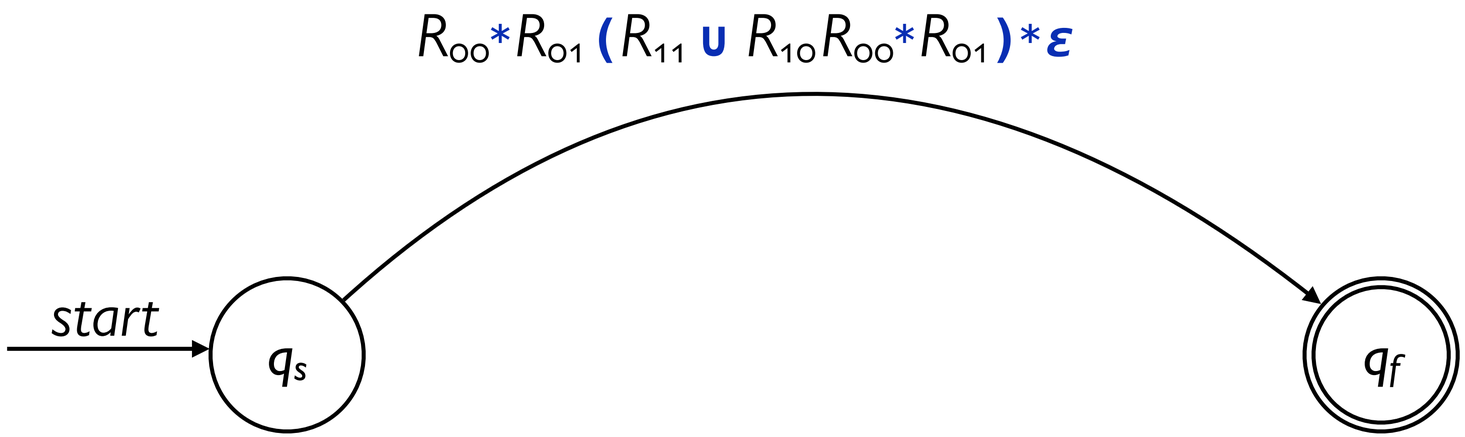
From GNFAs to regular expressions



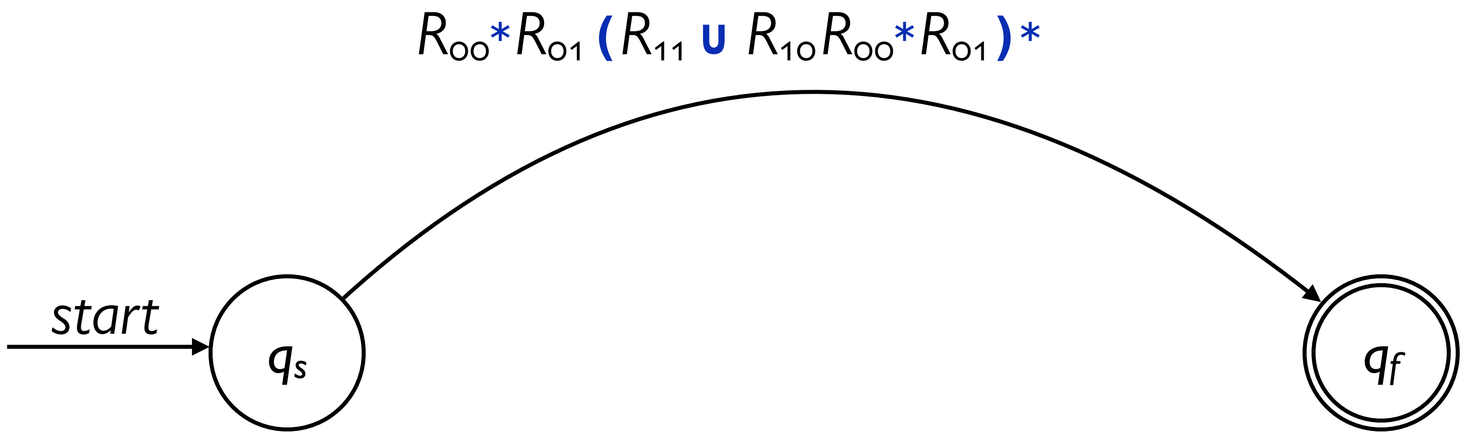
From GNFAs to regular expressions



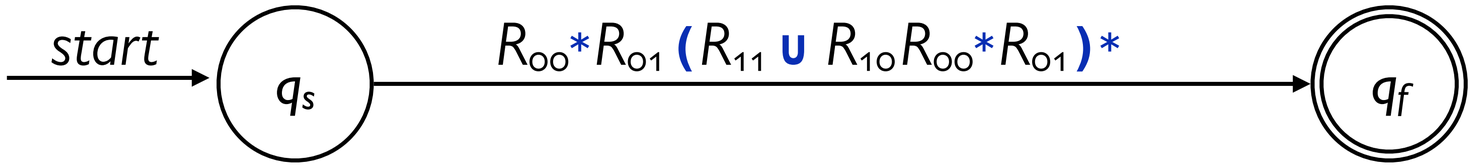
From GNFAs to regular expressions



From GNFAs to regular expressions

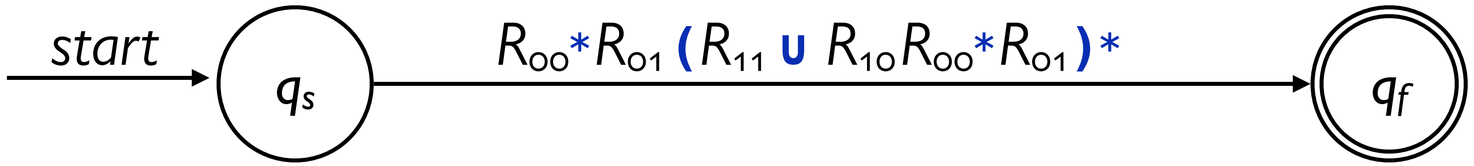


From GNFAs to regular expressions

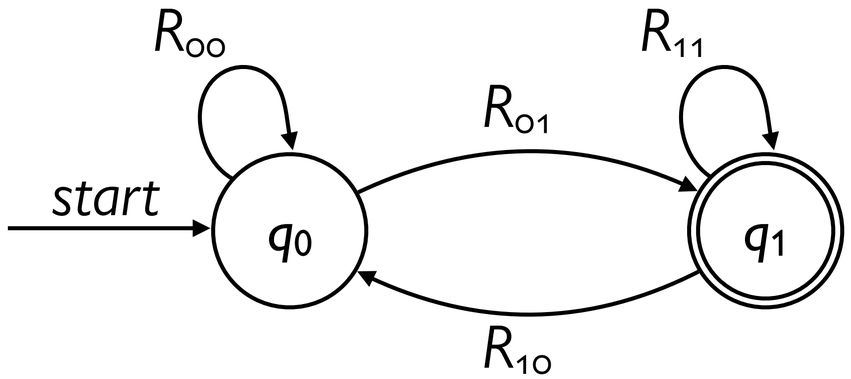


From GNFAs to regular expressions

After:



Before:



The state-elimination algorithm

1. Start with an NFA N for the language L, which we'll use as a generalized NFA (GNFA).

2. Add a new start state q_s and accept state q_f to N.

   Add an ε-transition from q_s to the old start state of N.

   Add ε-transitions from each accept state of N to q_f, then mark them as not accept states.

3. Repeatedly remove states other than q_s and q_f from the NFA by "shortcutting" them until only q_s and q_f remain.

4. The transition from q_s to q_f is now a regular expression equivalent to the original NFA.

To eliminate a state q_{rip} from the automaton, do the following for each pair of states q_i and q_j, where there's a transition from q_i into q_{rip} and a transition from q_{rip} into q_j:

Let R_{in} be the regex. on the transition from q_i to q_{rip}.

Let R_{out} be the regex. on the transition from q_{rip} to q_j.

If there is a regular expression R_{stay} on a transition from q_{rip} to itself,

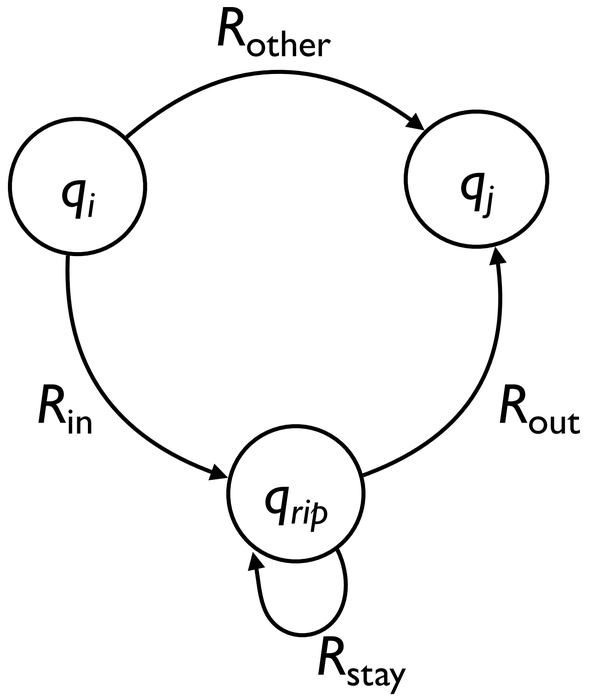
Add a new transition from q_i to q_j labeled $((R_{in})(R_{stay})^*(R_{out}))$.

Otherwise,

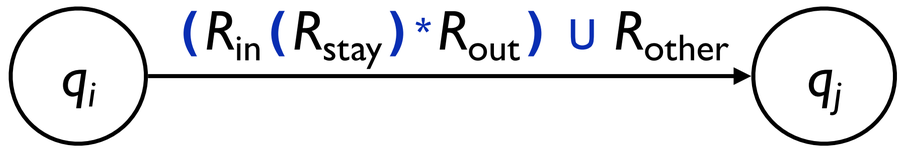
Add a new transition from q_i to q_j labeled $((R_{in})(R_{out}))$.

If a pair of states has multiple transitions between them labeled R_1, R_2, …, R_k, replace them with a single transition labeled $R_1 \cup R_2 \cup \cdots \cup R_k$.

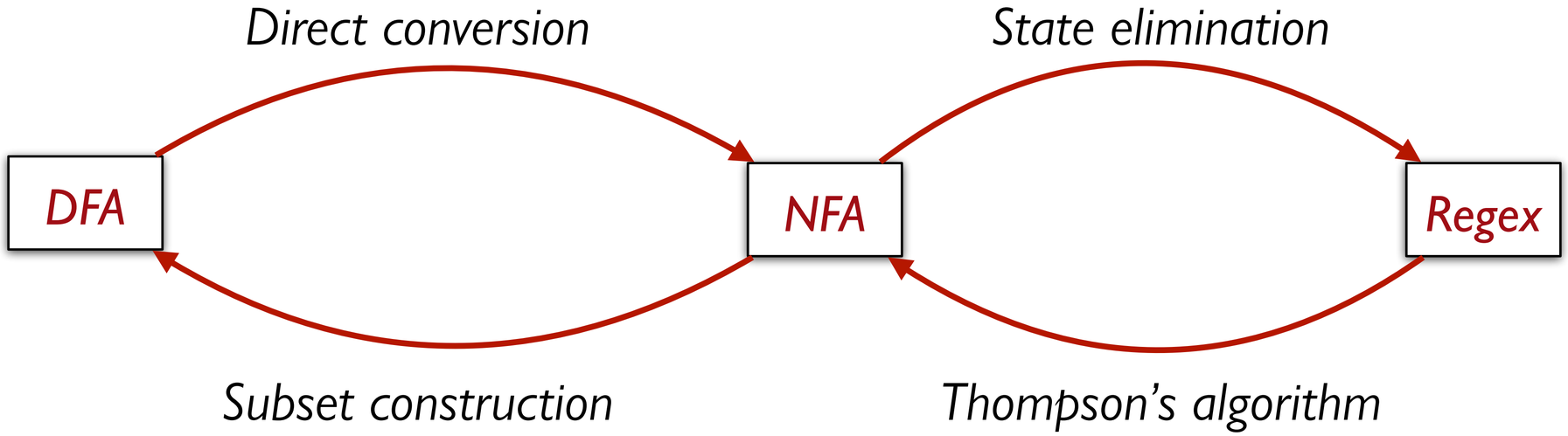
Before



After



Our transformations



The following are all equivalent:

L is a regular language.

There is a DFA *D* such that $L(D) = L$.

There is an NFA *N* such that $L(N) = L$.

There is a regular expression *R* such that $L(R) = L$.

Why this matters

The equivalence of regular expressions and finite automata has *practical* relevance.

> Tools like **`grep`** and **`flex`** that use regular expressions capture all the power available via DFAs and NFAs.

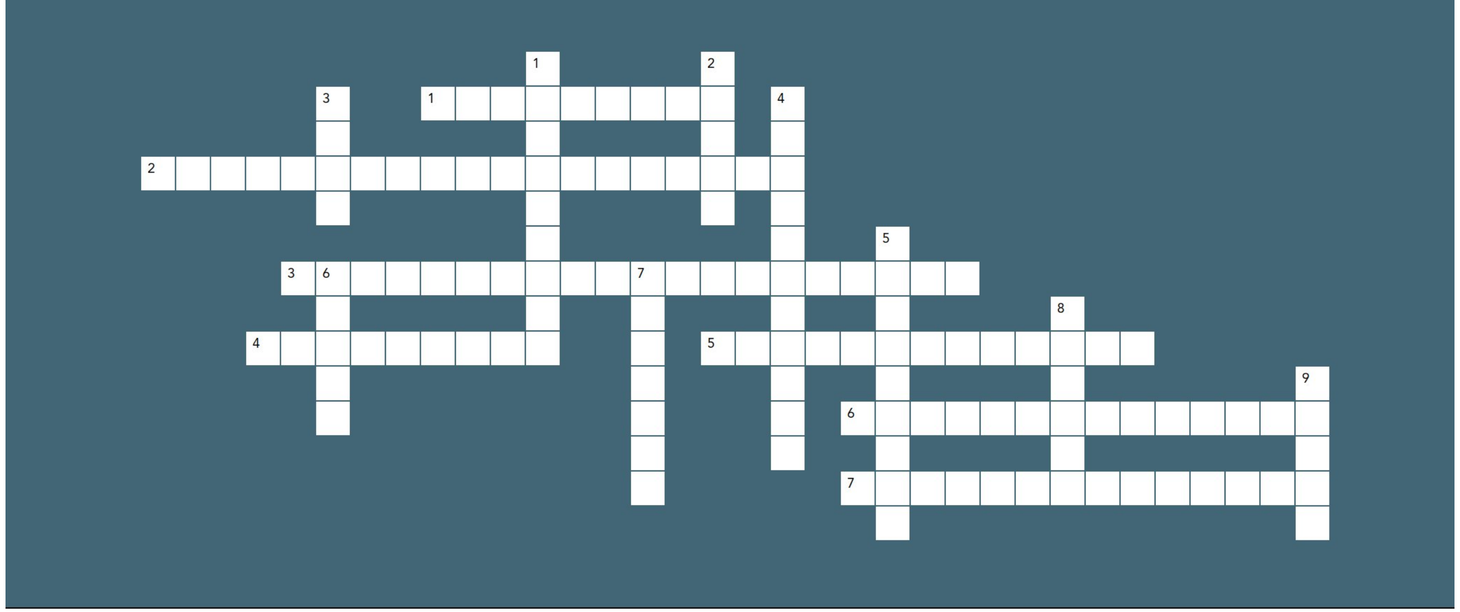
This is also hugely significant *theoretically*:

> The regular languages can be assembled "from scratch" using a small number of operations!

Next time

Is every language regular?

What, if anything, can't we solve with DFAs, NFAs, and regular expressions?



Regular expression crossword puzzle

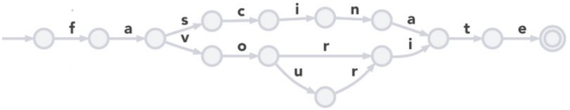
Fill out corresponding regexps in the grid, including operators.
Emoji go in a single square each, except when I say to spell them out as words instead.
Good luck!
— @thingskatedid

Kate

Across

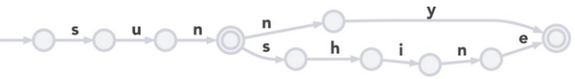
1. 

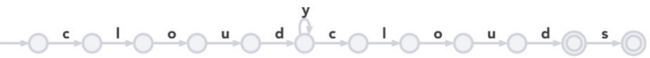
2. 

3. 

4. **/ 🌈+ /** (spell out the emoji, write an equivalent regex)

5. 

6. 

7. 

Down

1. **/ 💧|💧💧+ /** (spell out the emoji, write an equivalent regex)

2. 

3. 

4. 

(get creative!)

5. **/m.n.f.{4}/ ∩ /.{4}f.l.y/** (find a matching string)

6. **/(un)?.l.gn(ed)?/** (find a matching string)

7. **/.all(en|ing)*/** (find a matching string)

8. 

9. 

Acknowledgments

This lecture incorporates material from:

Nancy Ide, Vassar College

Keith Schwarz, Stanford University

Michael Sipser, *Introduction to the Theory of Computation*

Jeffrey Ullman, Stanford University