Tokenization

CSCI 1460: Computational Linguistics
Lecture 4

Ellie Pavlick
Fall 2023

Announcements

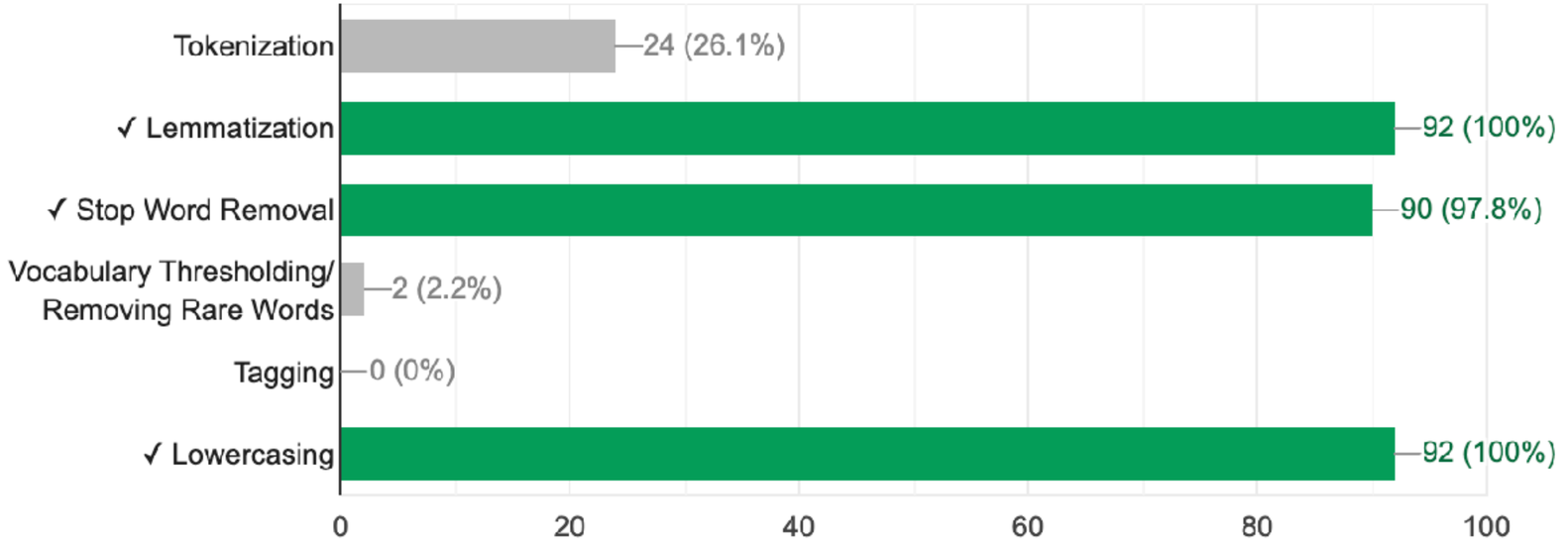
- Assignment 1 is out!
- I’ll be traveling next week. I’ll upload a pre-recorded lecture on Tuesday. Charlie will give a lecture on Thursday.

Quiz 3 Recap

In the below piece of text, which of the following preprocessing steps appear to have been run? Check all that apply.

Copy

65 / 92 correct responses



Quiz 3 Recap

Can you please explain in lecture the correct answer for these two from question #3:

Spelling out contractions (i.e., "I'm" -> "I am")

Spelling out abbreviations (i.e., "RI" -> "Rhode Island")

I selected Fewer for both of them because in the question it states "You can assume you are working with a very large corpus in which all words occur at least once." So I take that to mean that "I'm", "I", "am", "RI", "Rhode", "Island" and "Rhode Island" are all already in the corpus, so when you remove the contractions and abbreviations from the corpus that will result in less features in the matrix.

Is this the correct way to think about this question? Can you please elaborate on this 2 cases in the next lecture? Thank you!

Topics

- (English) Morphology 101
- Finite State Machines
- Subword Tokenization
- [Saving for Later] Supporting all languages!

Tokenization vs. Morphological Analysis

- Tokenization — splitting a string (sentence) into “words”
  - Unoaked chardonnay. -> [“unoaked”, “chardonnay”, “.”]
- Morphological parsing — splitting a “word” into morphemes
  - unoaked -> [“un”, “oak”, “ed”]
- These processes are not really different! Rather, points along a continuum

Tokenization

- Input: String of characters
- Output: list of (hopefully meaningful) units
- The Rhode Island-based baby food producer didn’t comment.
- “The” “Rhode Island” “based” “baby” “food” “produc” “er” “did” “n’t” “comment”

https://courses.grainger.illinois.edu/cs447/fa2019/Slides/Lecture02.pdf

Topics

- **(English) Morphology 101**
- Finite State Machines
- Subword Tokenization
- [Saving for Later] Supporting all languages!

Can’t we just split on white space?

https://courses.grainger.illinois.edu/cs447/fa2019/Slides/Lecture02.pdf

Can’t we just split on white space?

Problem 1: Compounding

- We can form new semantically meaningful units by combining existing words
- These compounds may or may not contain whitespace and punctuation
- E.g.,
  - “ice cream”
  - “website”, “web site”
  - “Rhode Island-based”

Can’t we just split o

Problem 1: Compounding

- We can form new semantically existing words
- These compounds may or ma punctuation
- E.g.,
  - “ice cream”
  - “website”, “web site”
  - “Rhode Island-based”

Can’t we just split on white space?

Problem 2: Many writing systems don’t use whitespace

- Chinese:
  - 我开始写小说 =
  - 我 (I) 开始 (start(ed)) 写 (writing) 小说 (novel(s))
- Turkish:
  - uygarlaştıramadıklarımızdanmışsınızcasına
  - uygar (civilized) laş (become) tır (cause to X) ama (not able) dık (past participle) lar (plural) ımız (first person plural) dan (from/among) mış (past tense) sınız (second person plural) casına (adverb form)
  - “behaving as if you are among those we could not civilize”

https://courses.grainger.illinois.edu/cs447/fa2019/Slides/Lecture02.pdf

Can’t we just split on white space?

Problem 3: Clitics

- Clitic: like a word, but only every occurs with other words; needs a “host”
- English:
  - “doesn’t” , “I’m”
- Italian:
  - “L’ho veduta ieri” (I saw her yesterday)

Can’t we just split on white space?

Problem 4: Word formation is productive!

- Can combine things in infinitely many ways
- New word enters the language: Zoom
  - All inflections are instantly valid: zoomed, zooming, will have zoomed
  - Particles: zoom in, zoom it, zoomed out
  - Compounds: zoom meeting, zoom party, zoom fatigue
  - …

Topics

- (English) Morphology 101
- **Finite State Machines**
- Subword Tokenization
- [Saving for Later] Supporting all languages!

Finite State Machines

- Key idea from Theory of Computation! (Take 1010 for more!)
- Main computational tool for morphological parsing in NLP
- Two related tools:
  - Finite State Automata (FSA): Used to “accept” a string—i.e., tells me whether or not a string is “in the language” defined by the FSA
  - Finite State Transducer (FST): Used to “translate” one string into another, i.e., output a new string for each input

Finite State Automata

- Model/rules which describes a language
- Can say whether or not a given string is “in” the language or not
- E.g., computational procedure for telling us:
  - dog, dogs, goose, geese, mouse, mice —> Good!
  - gooese, mouses, deg, meuse, dig, gice —> Bad!

Finite State Automata

Simple Example

“sheep” language: contains “b” followed by at least two “a”s followed by “!”

baa!	ba!
baaa!	baba!
baaaaaaaa!	abaaa!

Finite State Automata

Simple Example

“sheep” language: contains “b” followed by at least two “a”s followed by “!”

baa!	ba!
baaa!	baba!
baaaaaaaa!	abaaa!

Note: NOT machine learning!* We are going to write down a rule-based procedure for identifying strings in the language. No training. No statistics.

Finite State Automata

Simple Example

“sheep” language: contains “b” followed by at least two “a”s followed by “!”

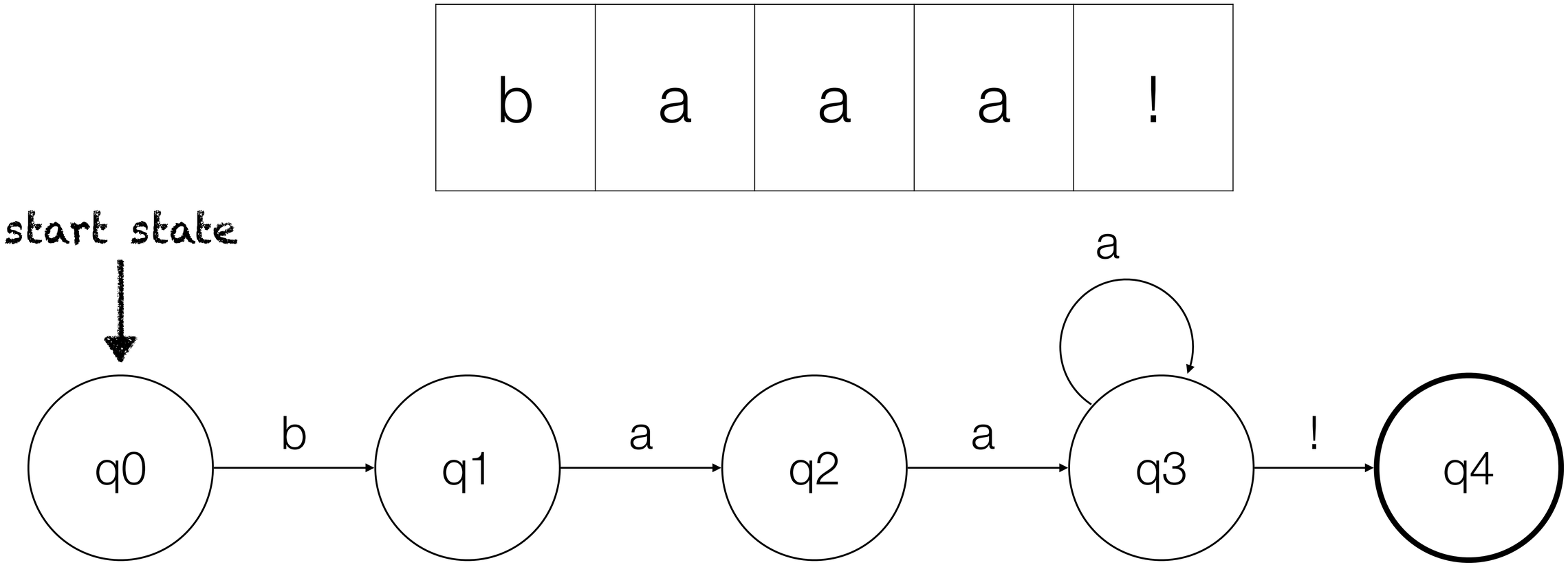
baa!	ba!
baaa!	baba!
baaaaaaaa!	abaaa!

Note: NOT machine learning!* We are going to write down a rule-based procedure for identifying strings in the language. No training. No statistics.

*Though, today, most morphological analysis would use machine learning

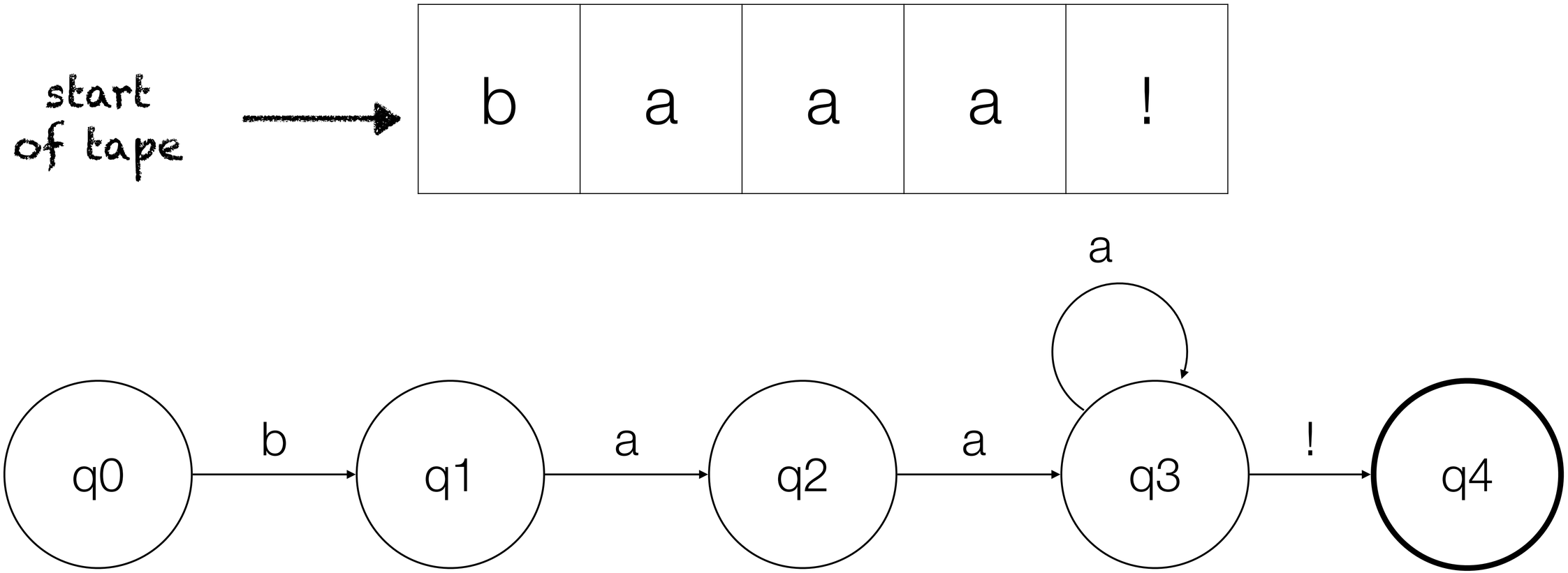
Finite State Automata

Simple Example



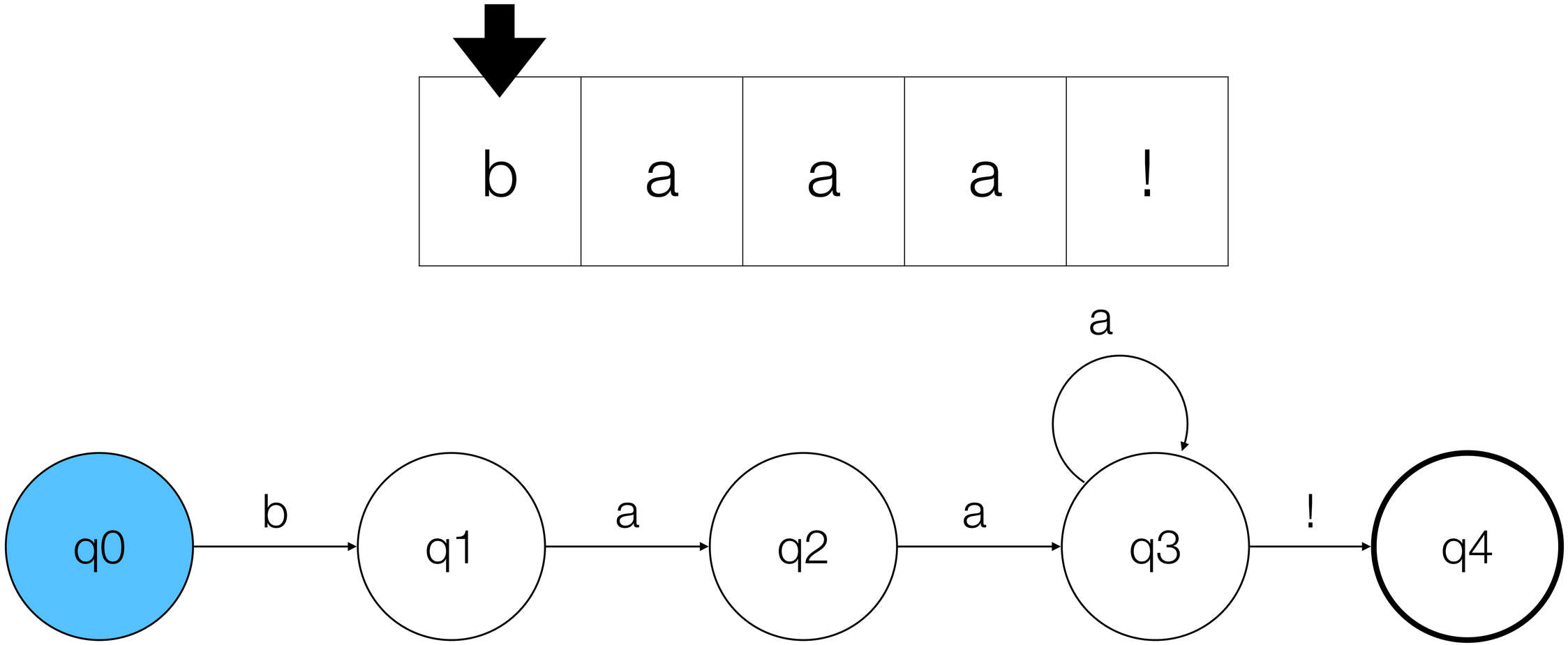
Finite State Automata

Simple Example



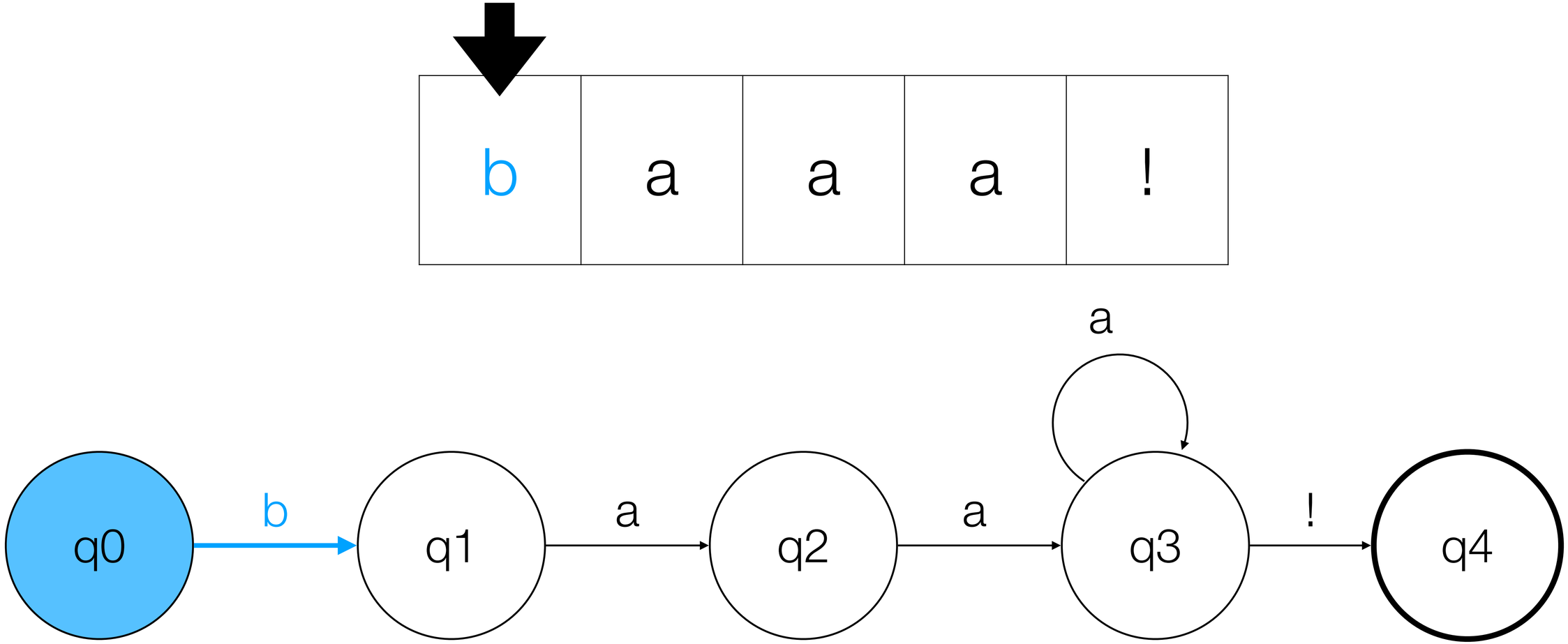
Finite State Automata

Simple Example



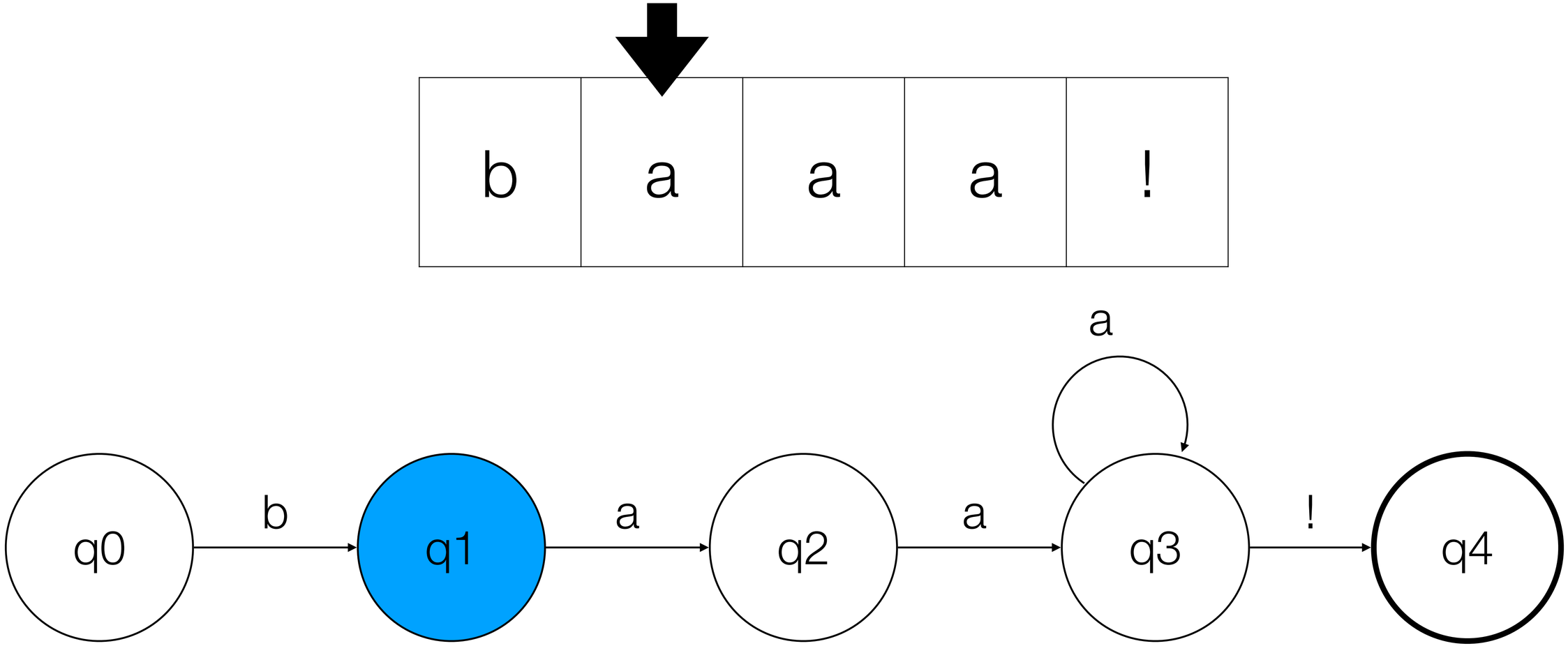
Finite State Automata

Simple Example



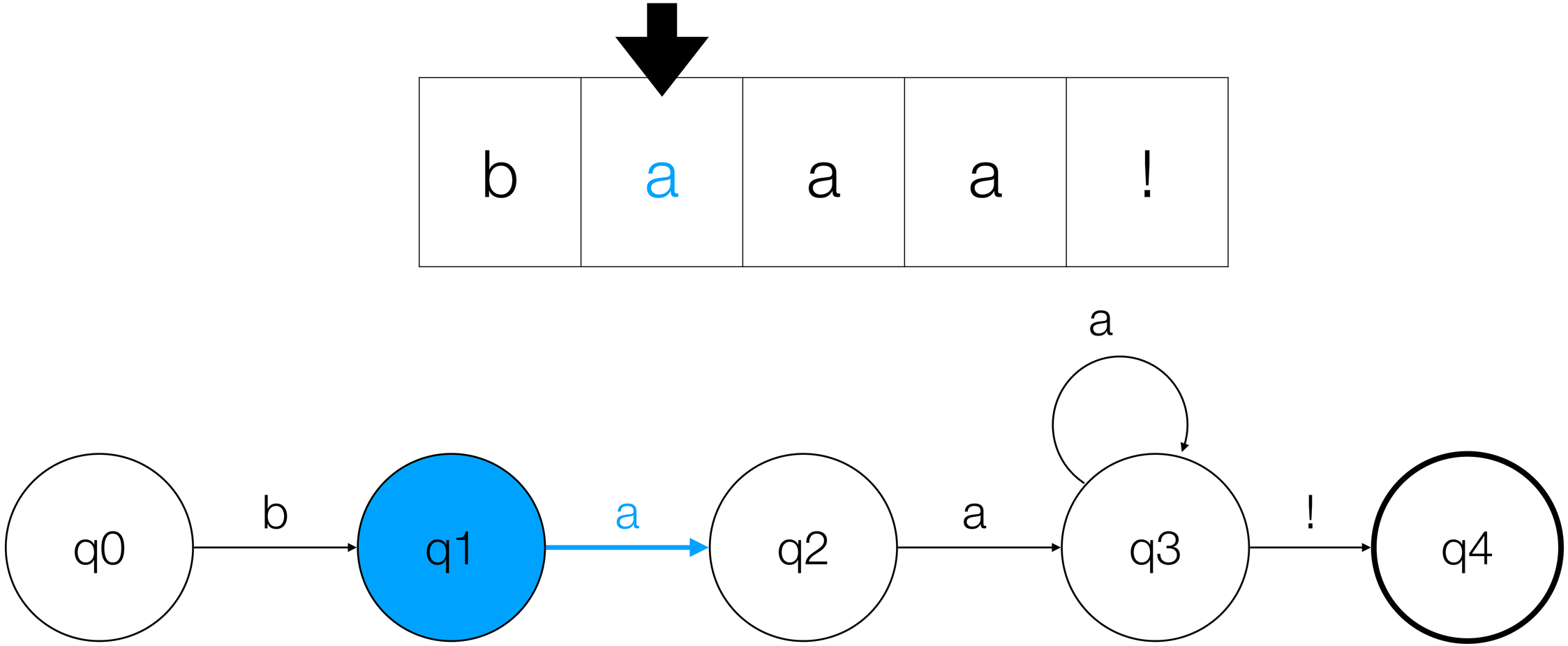
Finite State Automata

Simple Example



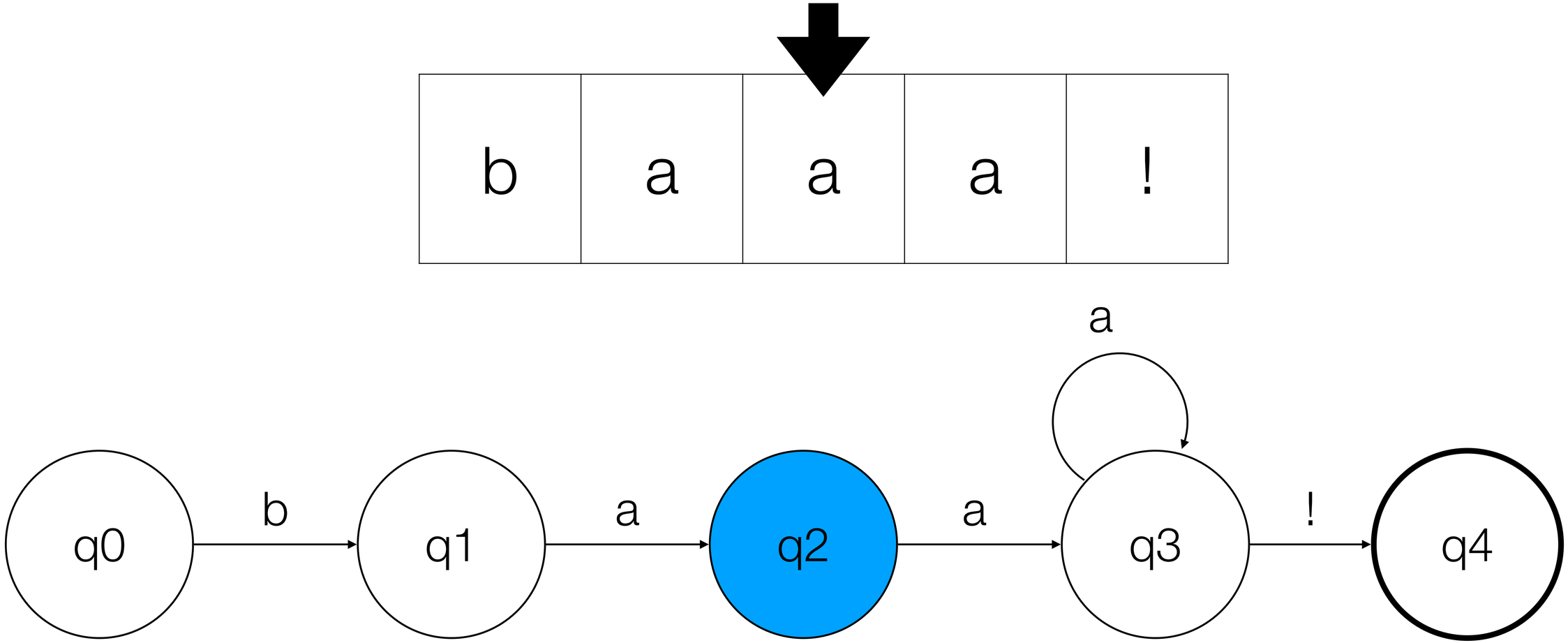
Finite State Automata

Simple Example



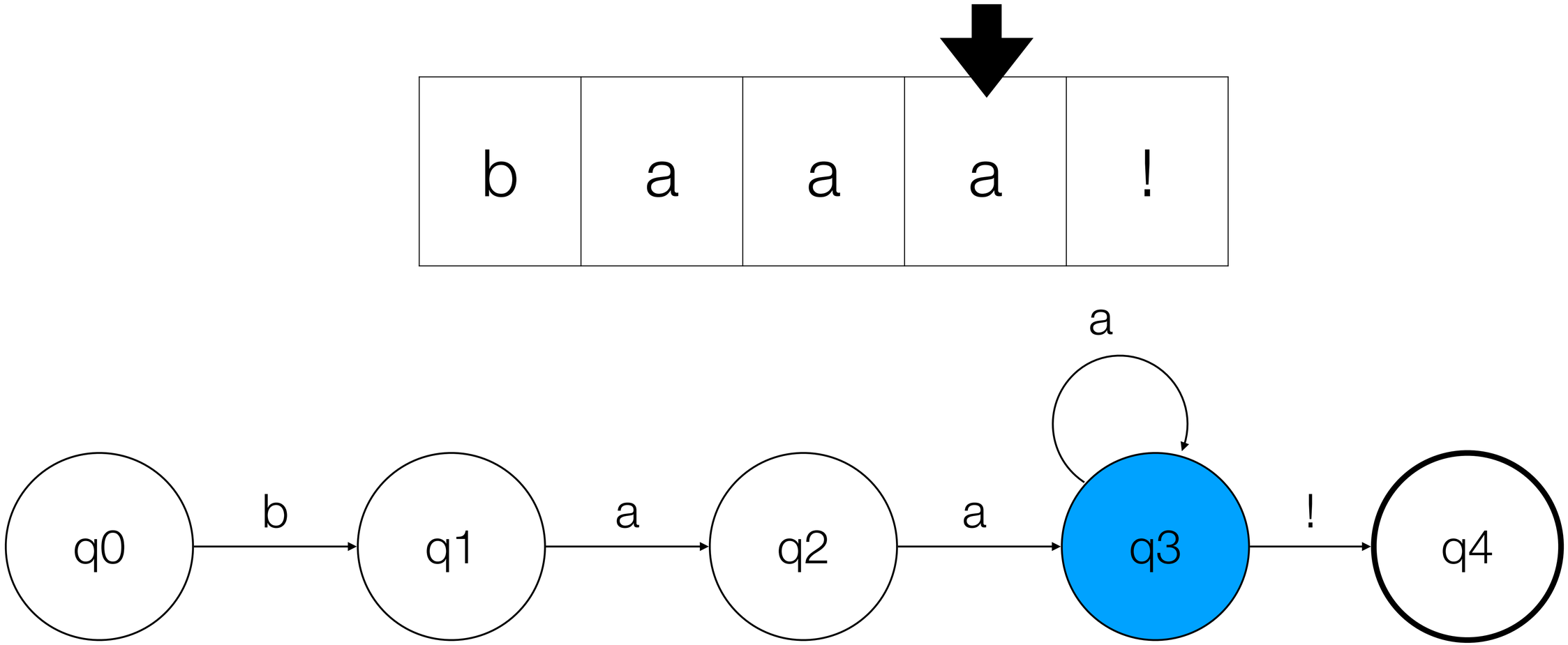
Finite State Automata

Simple Example



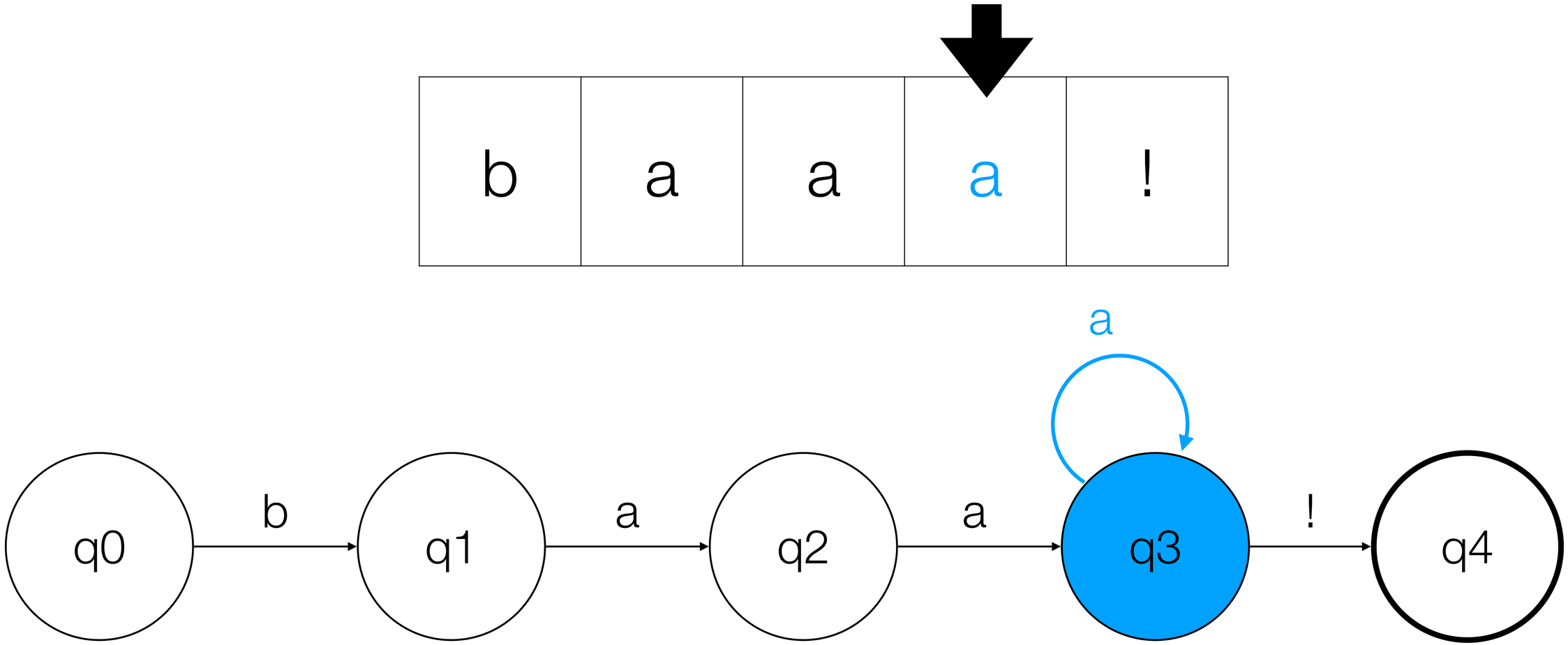
Finite State Automata

Simple Example



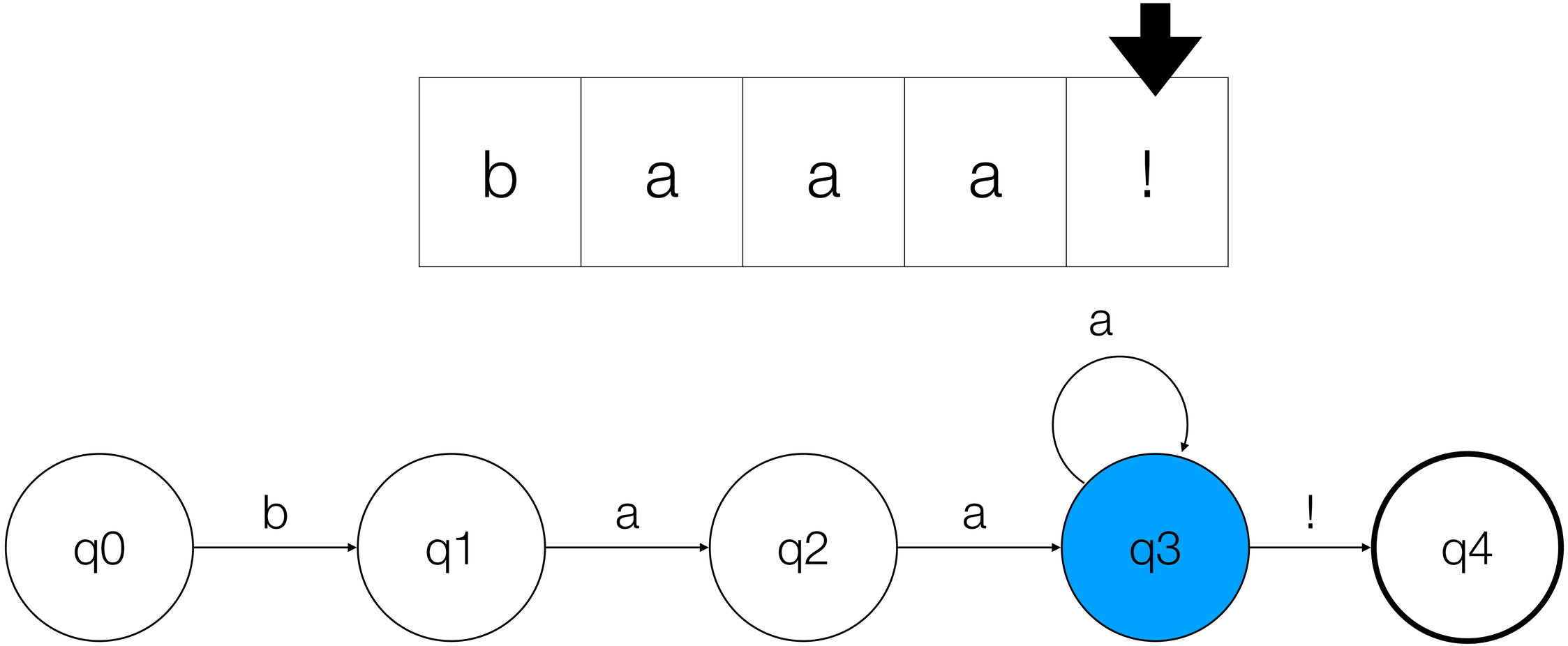
Finite State Automata

Simple Example



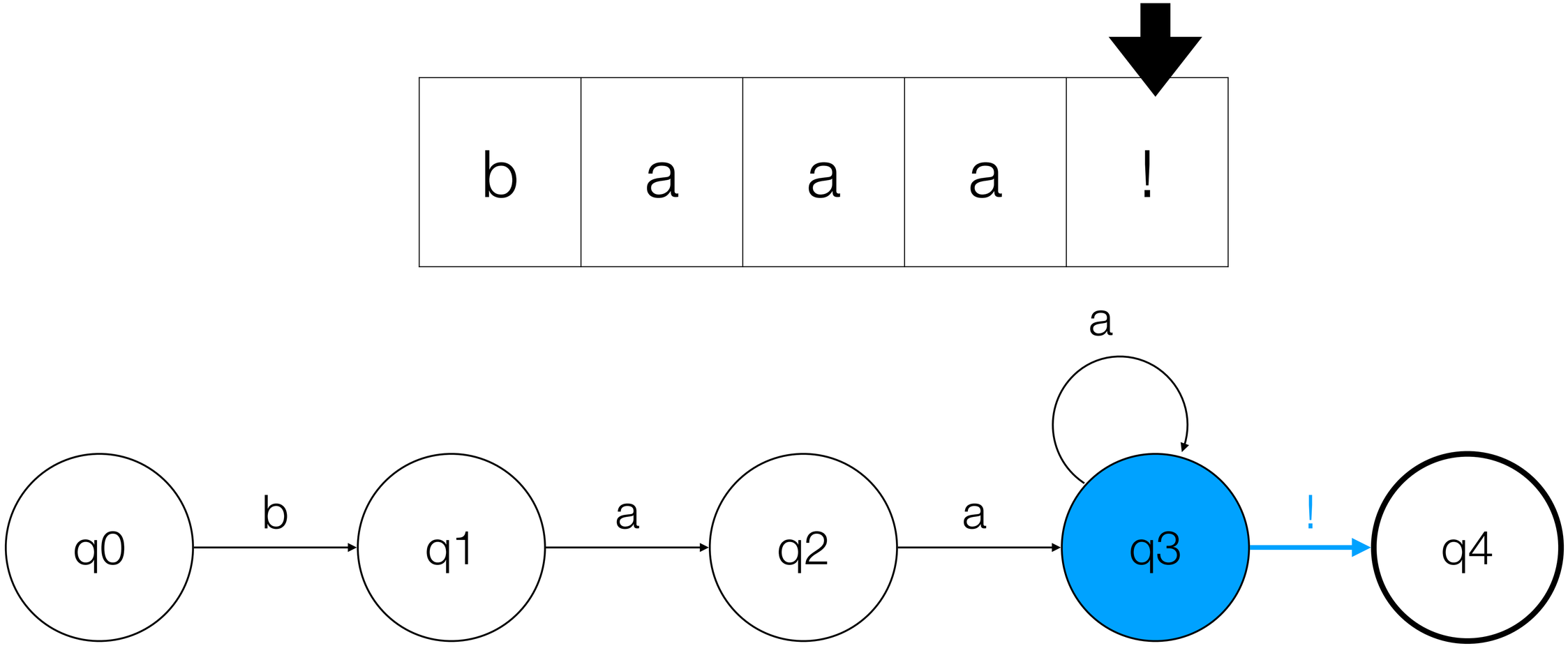
Finite State Automata

Simple Example



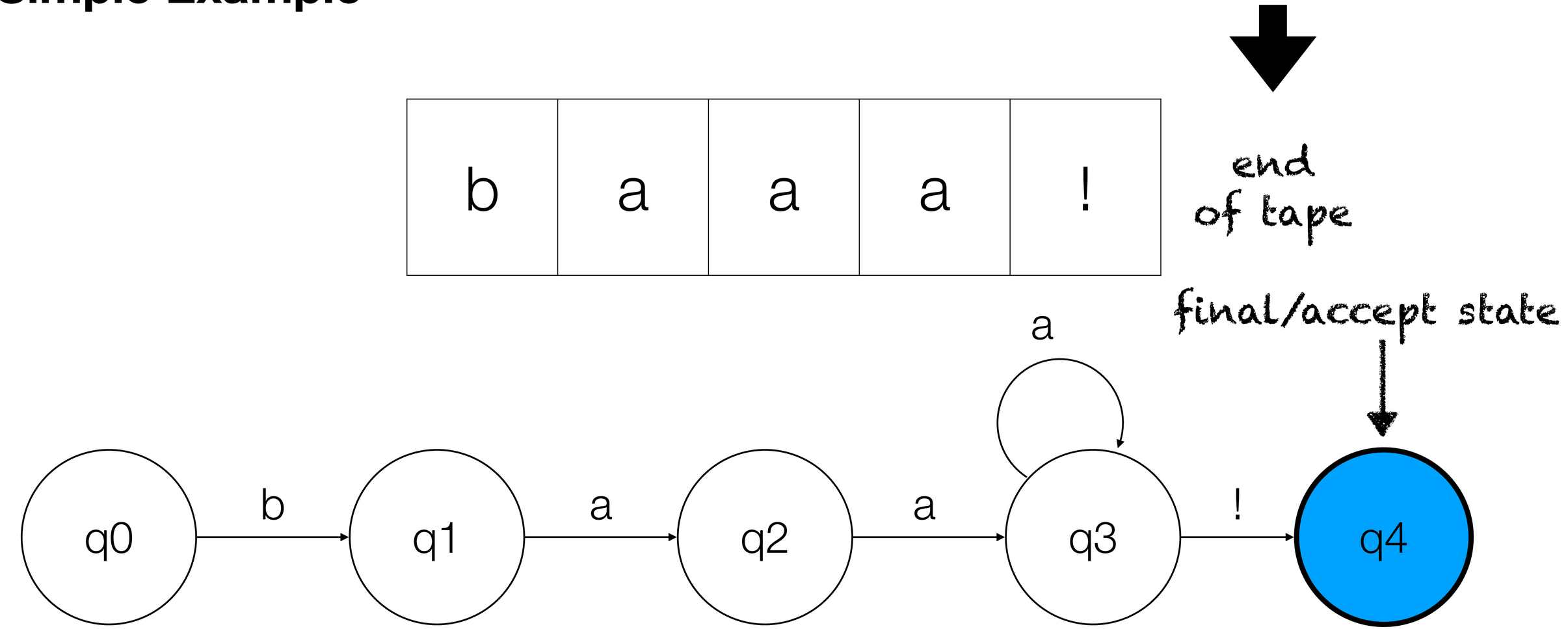
Finite State Automata

Simple Example



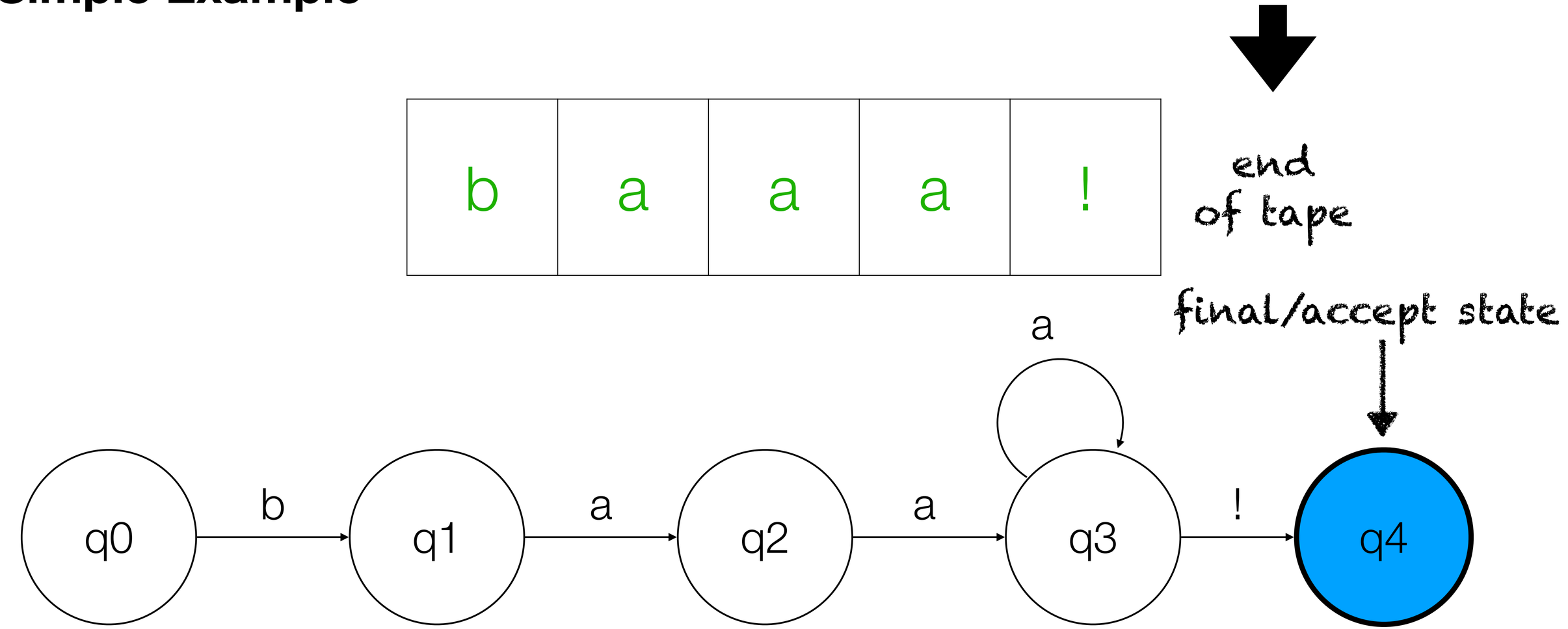
Finite State Automata

Simple Example



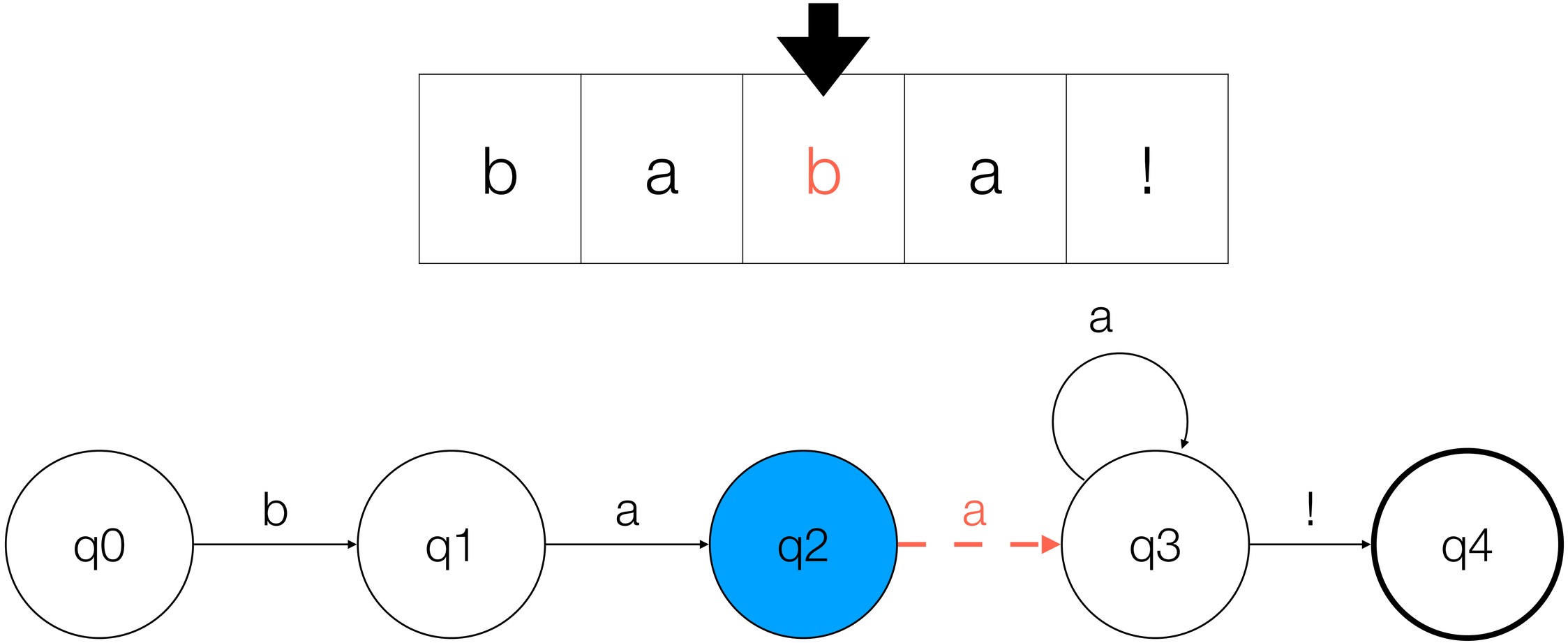
Finite State Automata

Simple Example



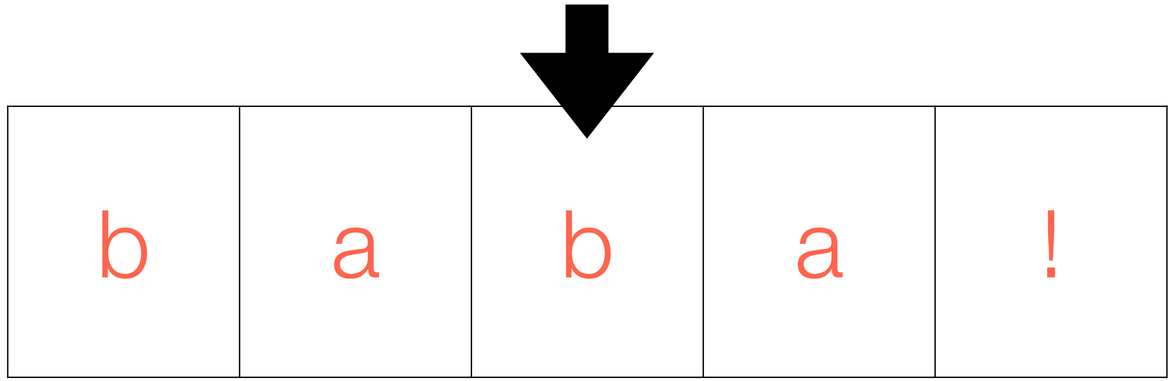
Finite State Automata

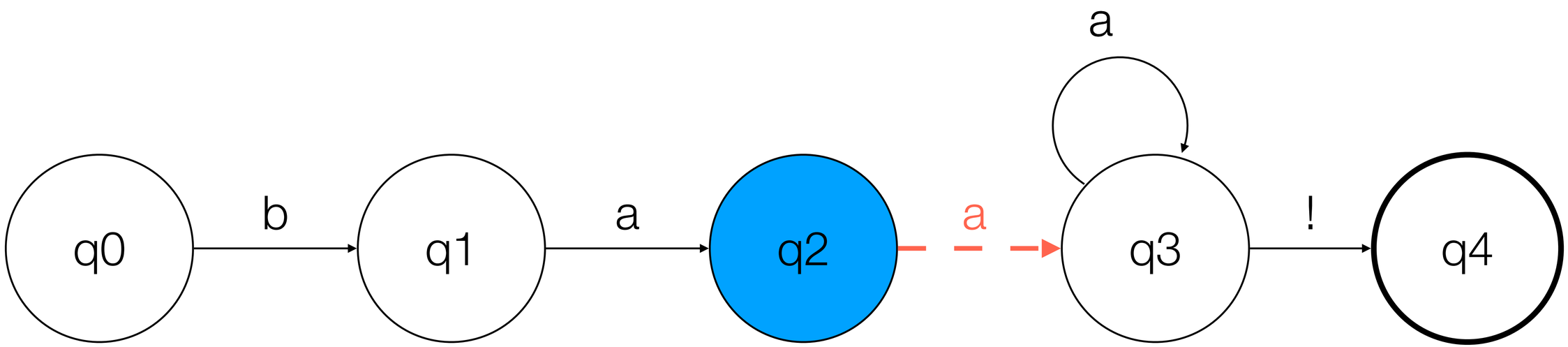
Simple Example



Finite State Automata

Simple Example





Finite State Automata

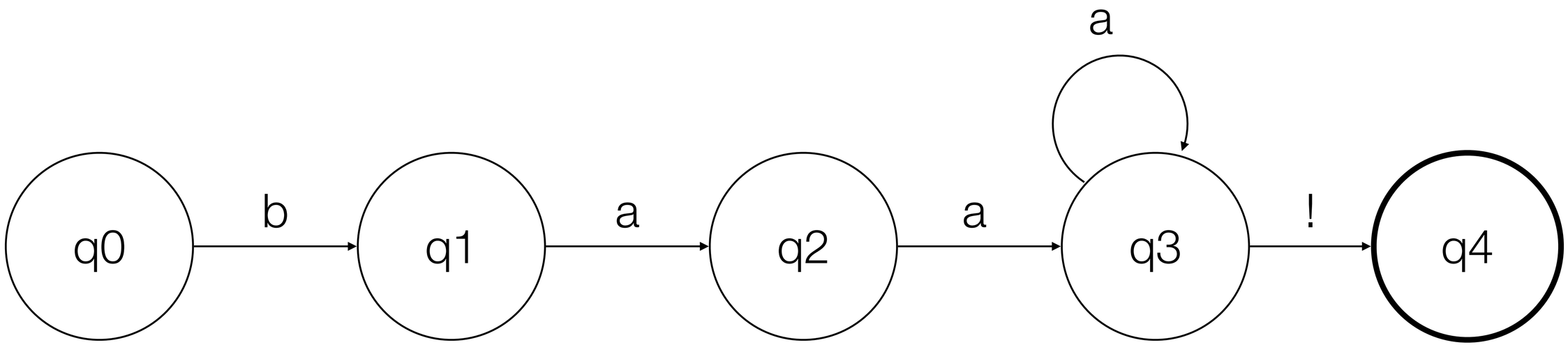
Formal Definition

- $Q = \{q_0, q_1, \ldots q_{N-1}\}$ = a finite set of **states**
- $\Sigma = \{i_0, i_1, \ldots i_{M-1}\}$ = an **alphabet**, i.e., a finite set of symbols
- q_0 = a designated **start state**
- $F \subseteq Q$ = a designated set of **final states**
- $\delta(q,i)$ = a **transition function** that, given a state and a symbol from the alphabet, returns the new state

Finite State Automata

Implementation

	a	b	!
0	–	1	–
1	2	–	–
2	3	–	–
3	3	–	4
4*	–	–	–

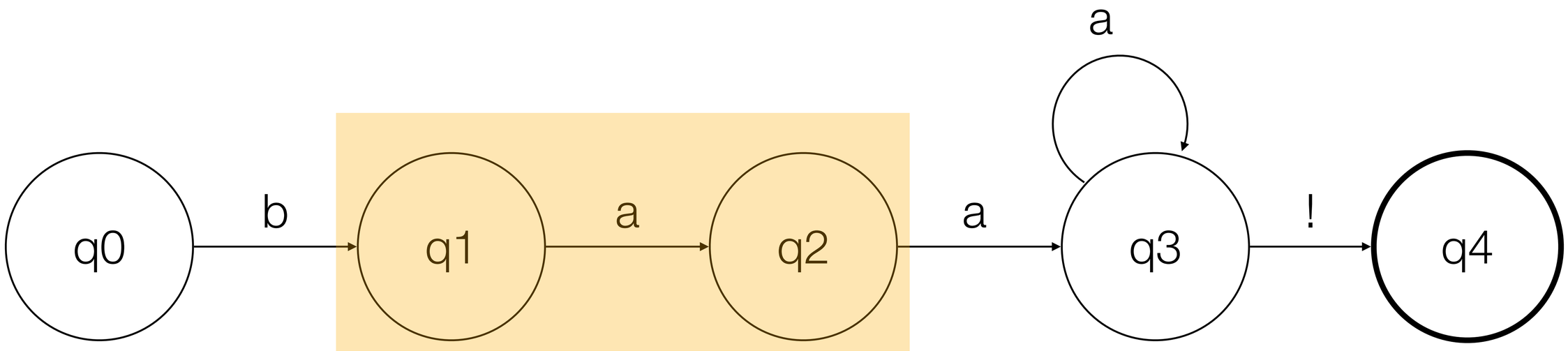


Finite State Automata

Implementation

from state {row},
on seeing symbol {column},
go do state {cell}

	a	b	!
0	—	1	—
1	2	—	—
2	3	—	—
3	3	—	4
4*	—	—	—

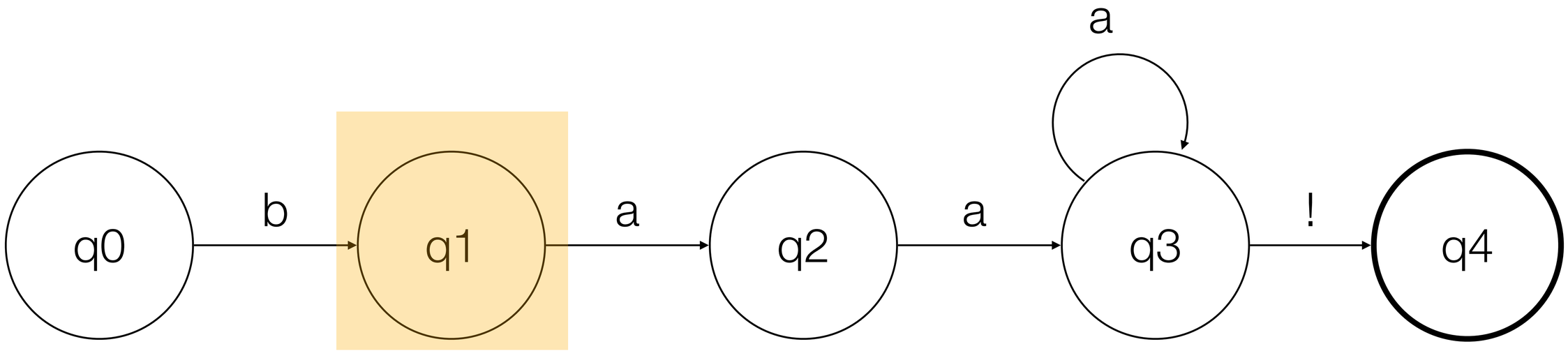


Finite State Automata

Implementation

from state {row},
on seeing symbol {column},
go do state {cell}

	a	b	!
0	—	1	—
1	2	—	—
2	3	—	—
3	3	—	4
4*	—	—	—

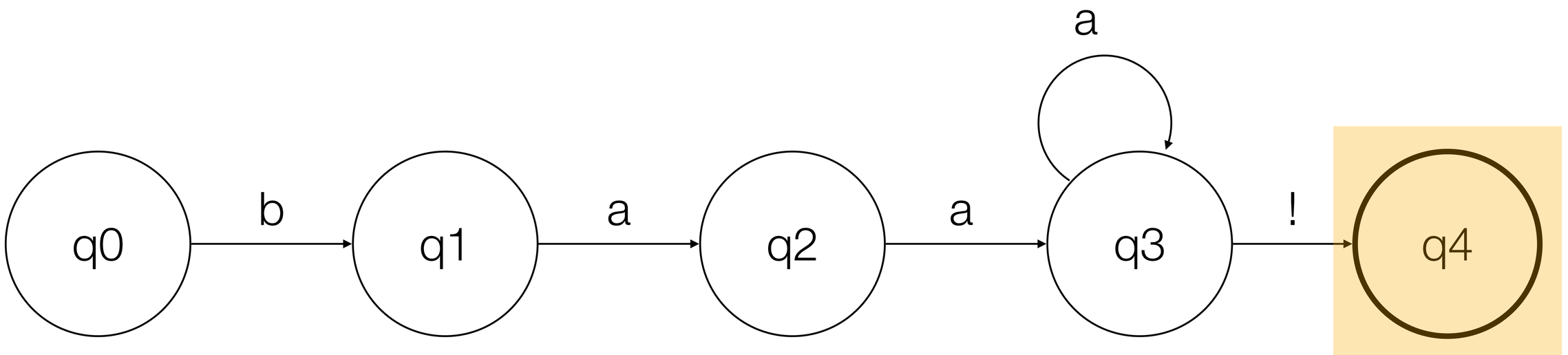


Finite State Automata

Implementation

designate subset of states as "accept states"

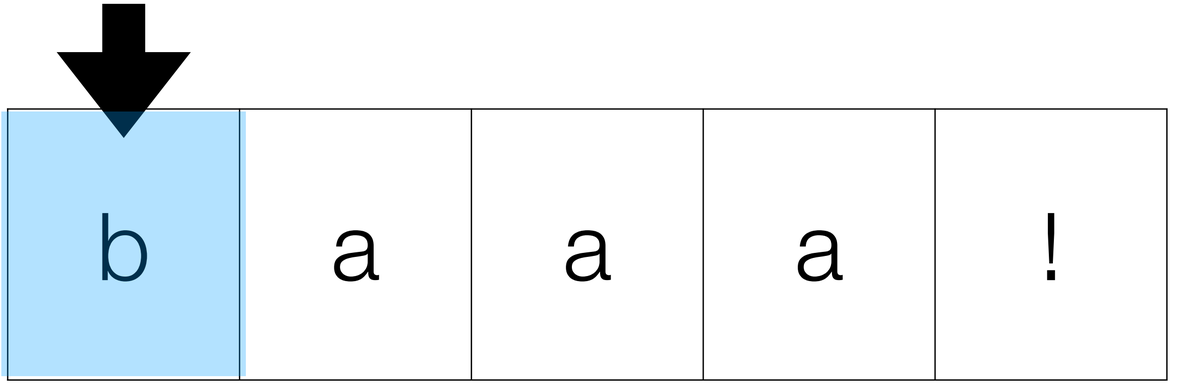
	a	b	!
0	—	1	—
1	2	—	—
2	3	—	—
3	3	—	4
4*	—	—	—



Finite State Automata

Implementation

```
recognize(tape, machine) -> accept or reject
  index <- beginning of tape
  cur_state <- initial state of machine
  loop
    if end of input reached then
      if cur_state in FinalStates then
        return Accept
      else
        return Reject
    elif Transition[cur_state, tape[index]] is empty then
      return Reject
    else
      cur_state <- Transition[cur_state, tape[index]]
      index += 1
  end
```

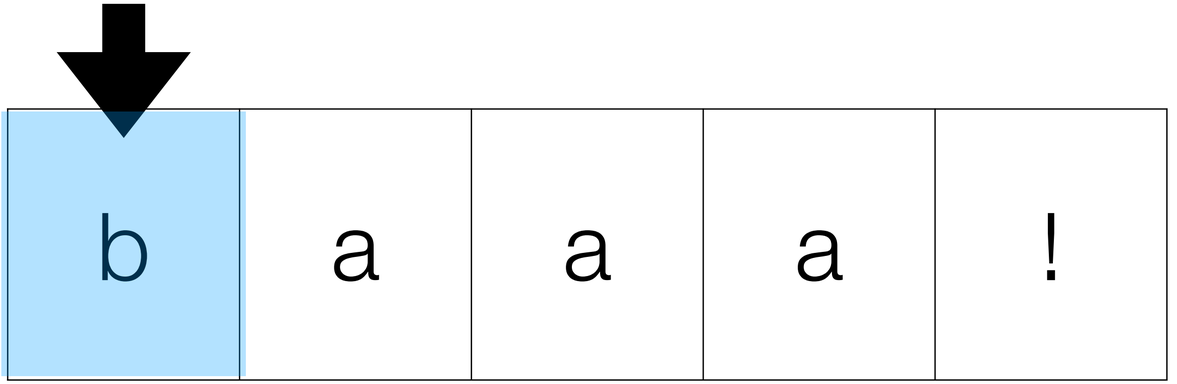


	a	b	!
0	–	1	–
1	2	–	–
2	3	–	–
3	3	–	4
4*	–	–	–

Finite State Automata

Implementation

```
recognize(tape, machine) -> accept or reject
  index <- beginning of tape
  cur_state <- initial state of machine
  loop
    if end of input reached then
      if cur_state in FinalStates then
        return Accept
      else
        return Reject
    elif Transition[cur_state, tape[index]] is empty then
      return Reject
    else
      cur_state <- Transition[cur_state, tape[index]]
      index += 1
  end
```

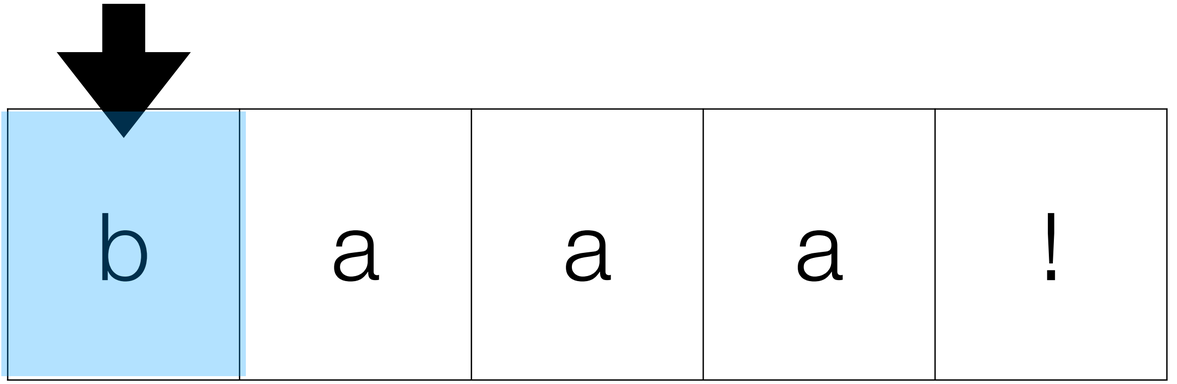


	a	b	!
0	–	1	–
1	2	–	–
2	3	–	–
3	3	–	4
4*	–	–	–

Finite State Automata

Implementation

```
recognize(tape, machine) -> accept or reject
  index <- beginning of tape
  cur_state <- initial state of machine
  loop
    if end of input reached then
      if cur_state in FinalStates then
        return Accept
      else
        return Reject
    elif Transition[cur_state, tape[index]] is empty then
      return Reject
    else
      cur_state <- Transition[cur_state, tape[index]]
      index += 1
  end
```

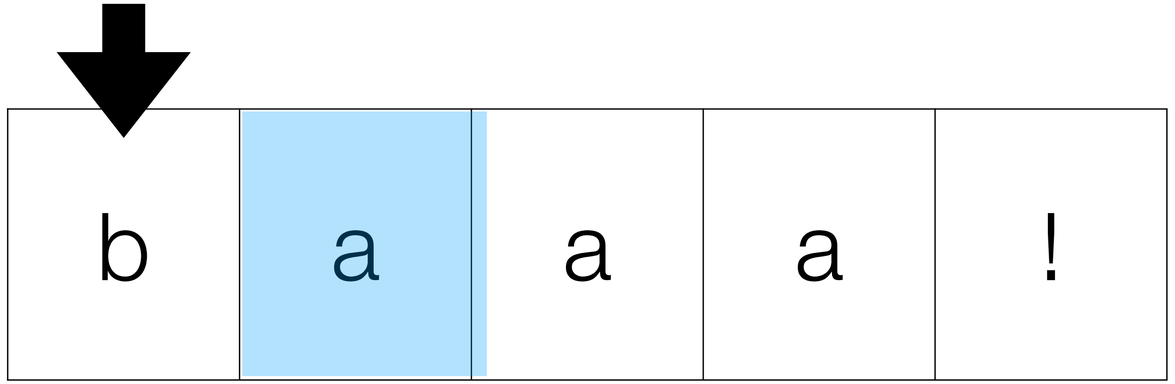


	a	b	!
0	–	1	–
1	2	–	–
2	3	–	–
3	3	–	4
4*	–	–	–

Finite State Automata

Implementation

```
recognize(tape, machine) -> accept or reject
  index <- beginning of tape
  cur_state <- initial state of machine
  loop
    if end of input reached then
      if cur_state in FinalStates then
        return Accept
      else
        return Reject
    elif Transition[cur_state, tape[index]] is empty then
      return Reject
    else
      cur_state <- Transition[cur_state, tape[index]]
      index += 1
  end
```



	a	b	!
0	—	1	—
1	2	—	—
2	3	—	—
3	3	—	4
4*	—	—	—

Finite State Automata

Non-Determinism

Finite State Automata

Non-Determinism

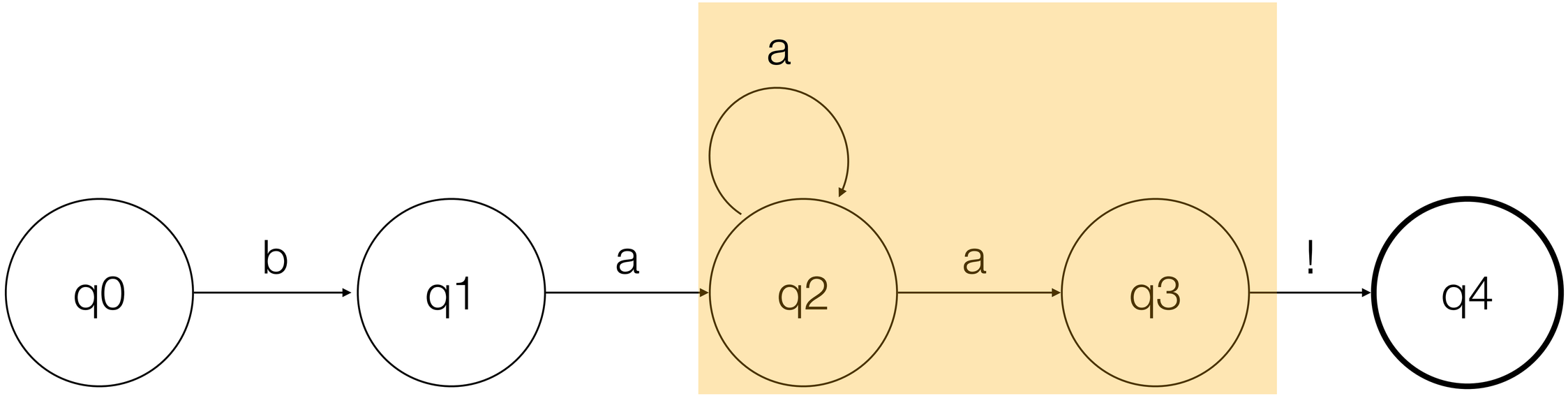
- Two ways of representing:

Finite State Automata

Non-Determinism

- Two ways of representing:
  - Multiple links out of the same node

	a	b	!
0	—	1	—
1	2	—	—
2	2,3	—	—
3	3	—	4
4*	—	—	—

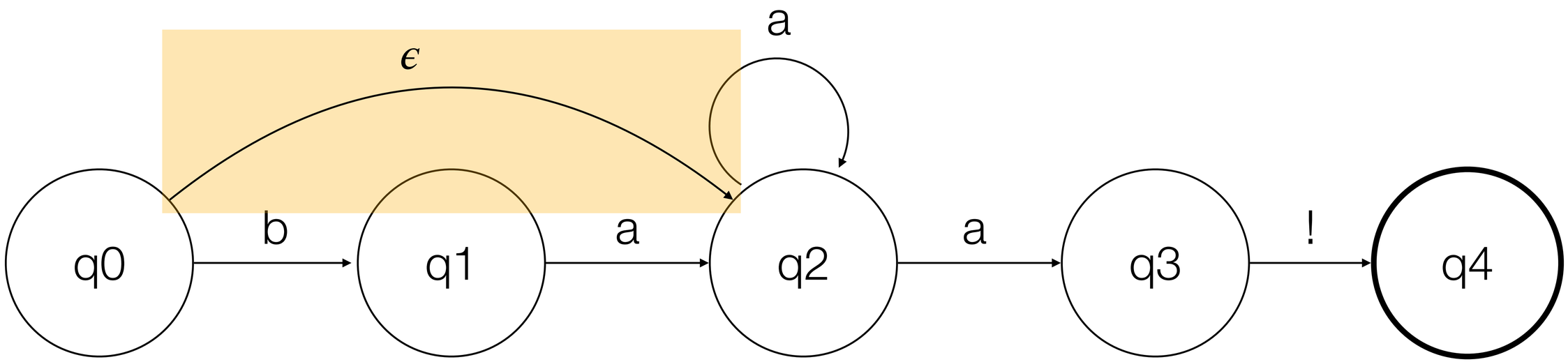


Finite State Automata

Non-Determinism

- Two ways of representing:
  - Multiple links out of the same node
  - Epsilon transitions

	ϵ	a	b	!
0	2	—	1	—
1	—	2	—	—
2	—	2,3	—	—
3	—	3	—	4
4*	—	—	—	—

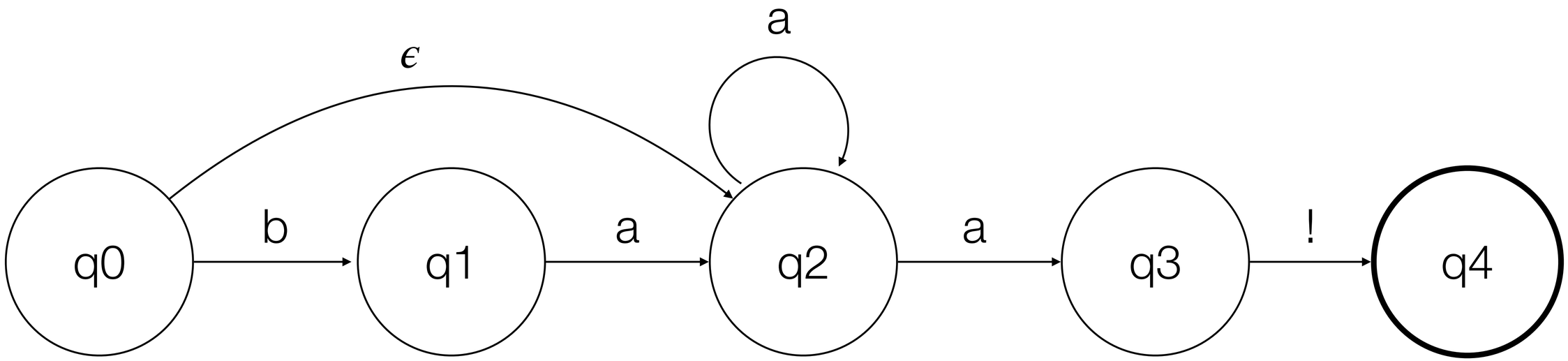


Finite State Automata

Non-Determinism

- Two ways of representing:
  - Multiple links out of the same node
  - Epsilon transitions
- Solvable using e.g., parallelism

	ϵ	a	b	!
0	2	—	1	—
1	—	2	—	—
2	—	2,3	—	—
3	—	3	—	4
4*	—	—	—	—



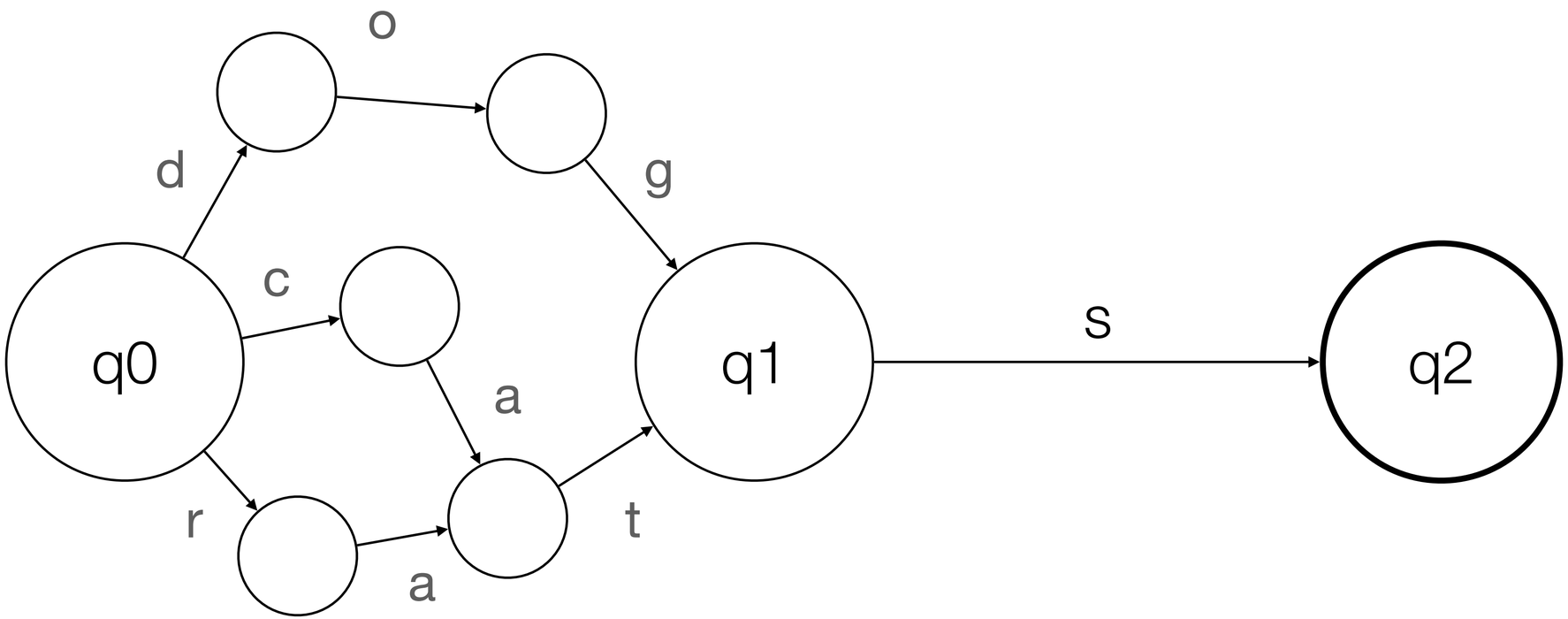
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

Finite State Automata

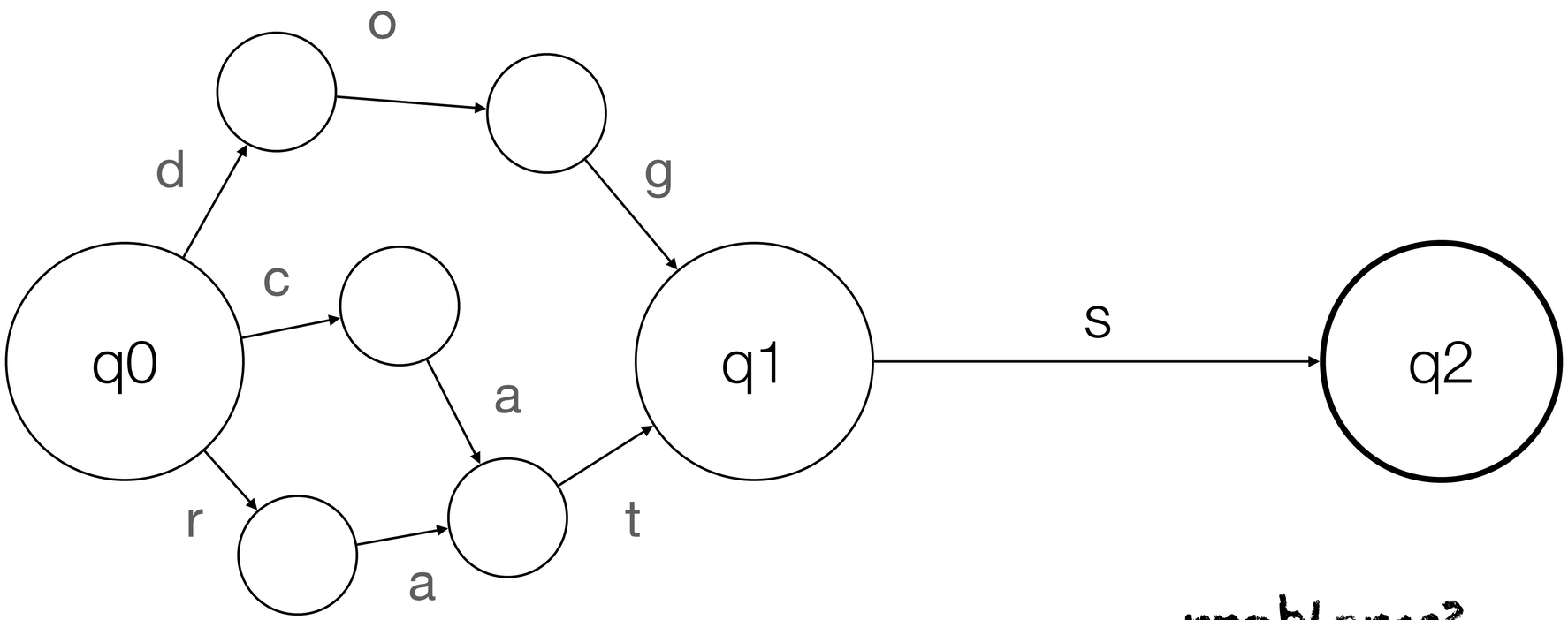
English plurals



Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

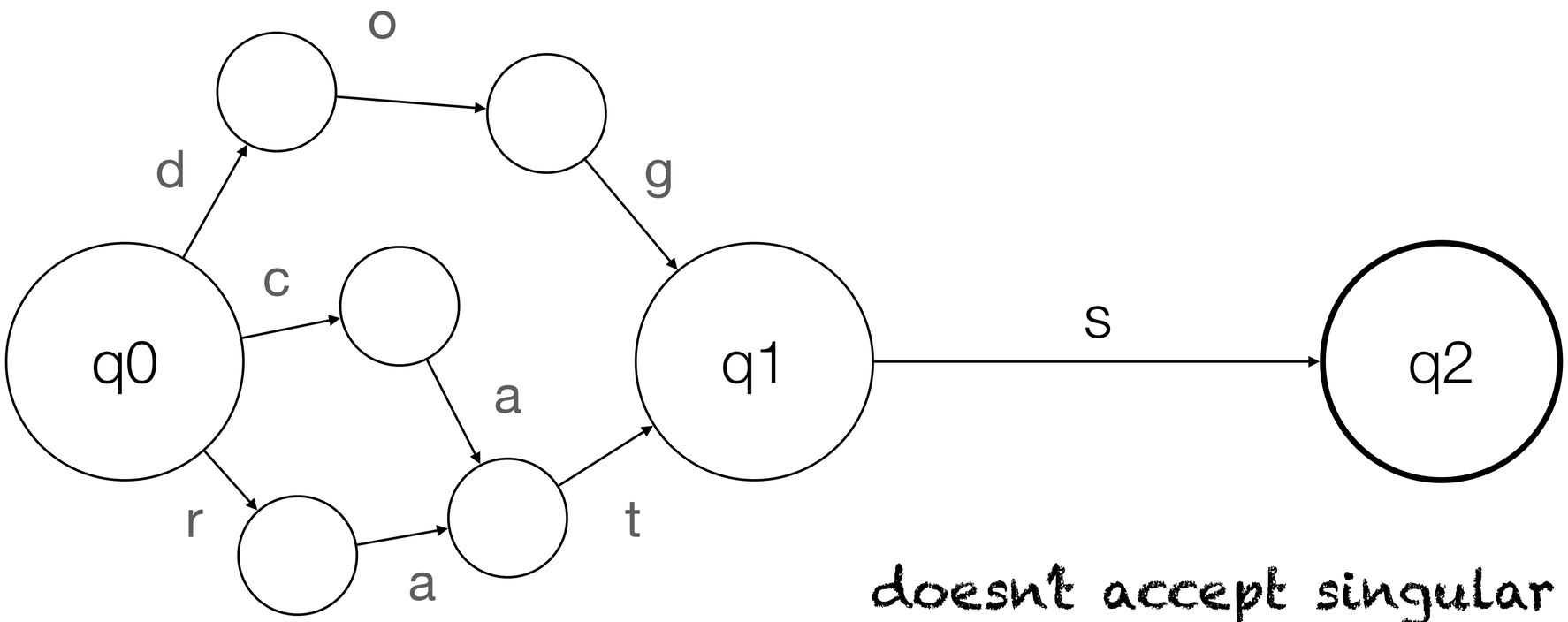


problems?

Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

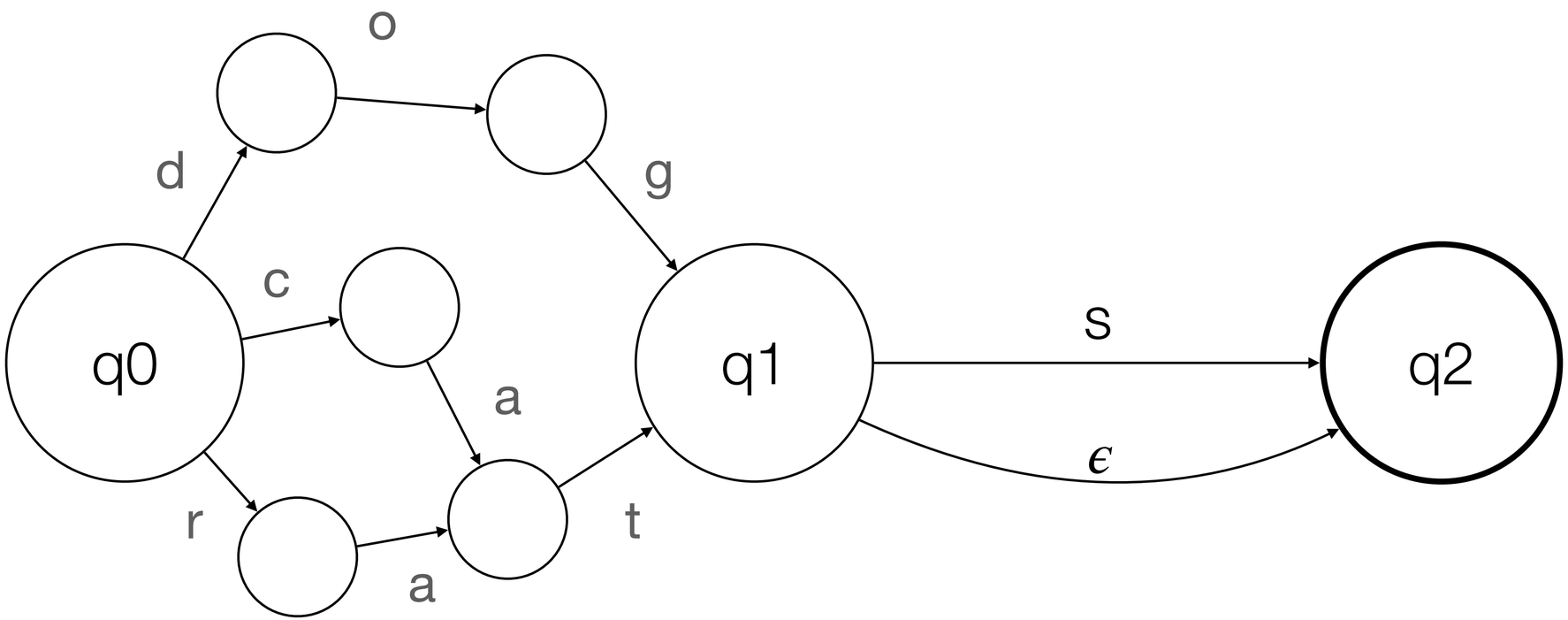


doesn't accept singular nouns!
(cat, dog, rat)

Goal: Accept valid English nouns (singular or plural)

Finite State Automata

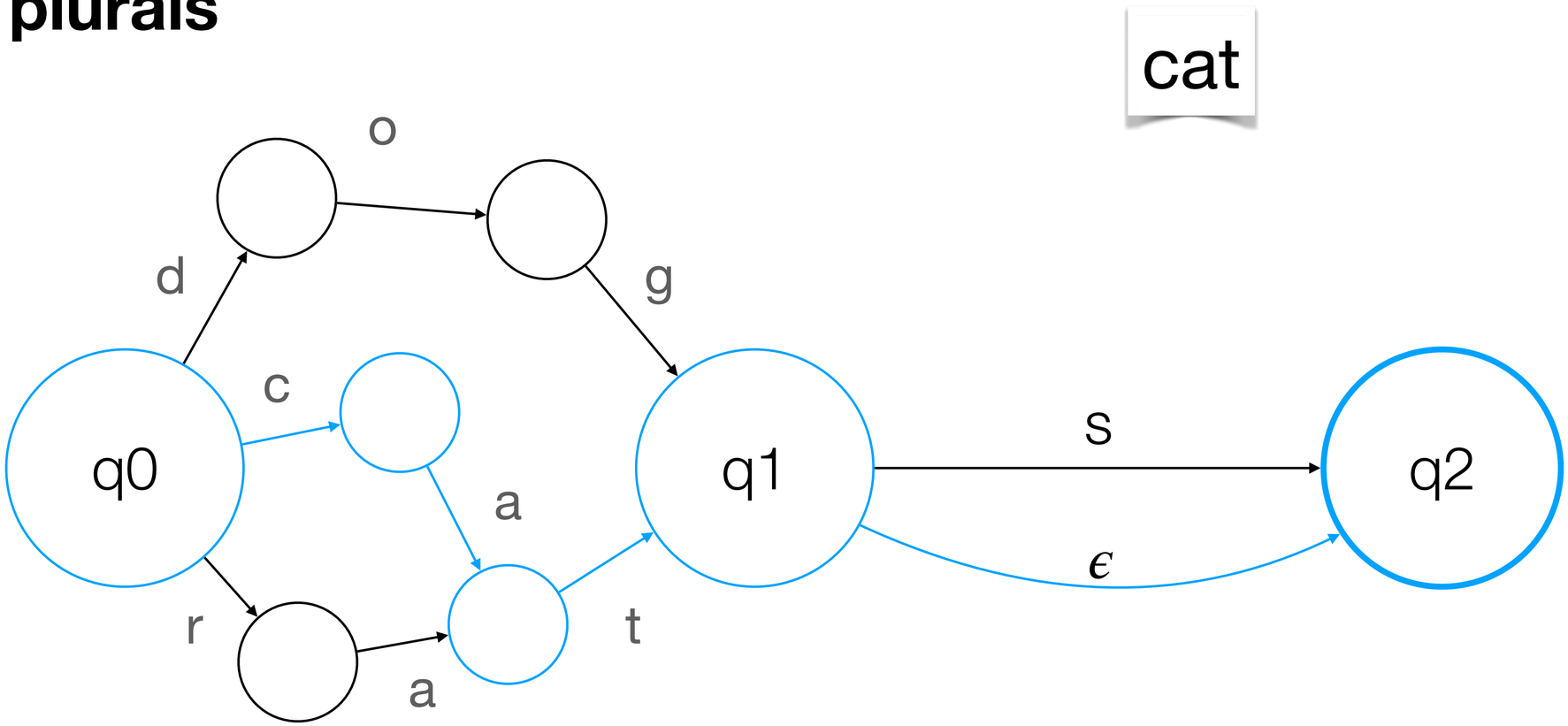
English plurals



Goal: Accept valid English nouns (singular or plural)

Finite State Automata

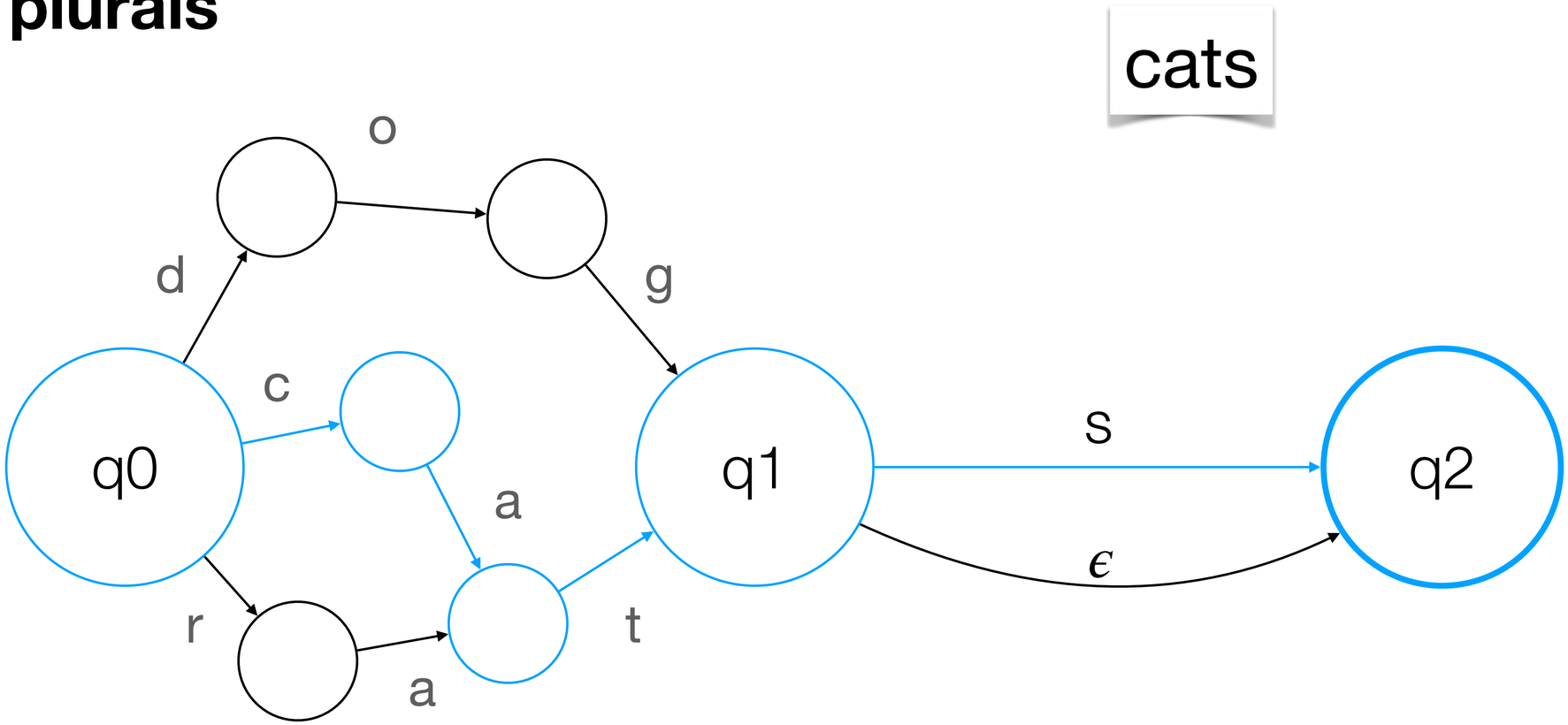
English plurals



Goal: Accept valid English nouns (singular or plural)

Finite State Automata

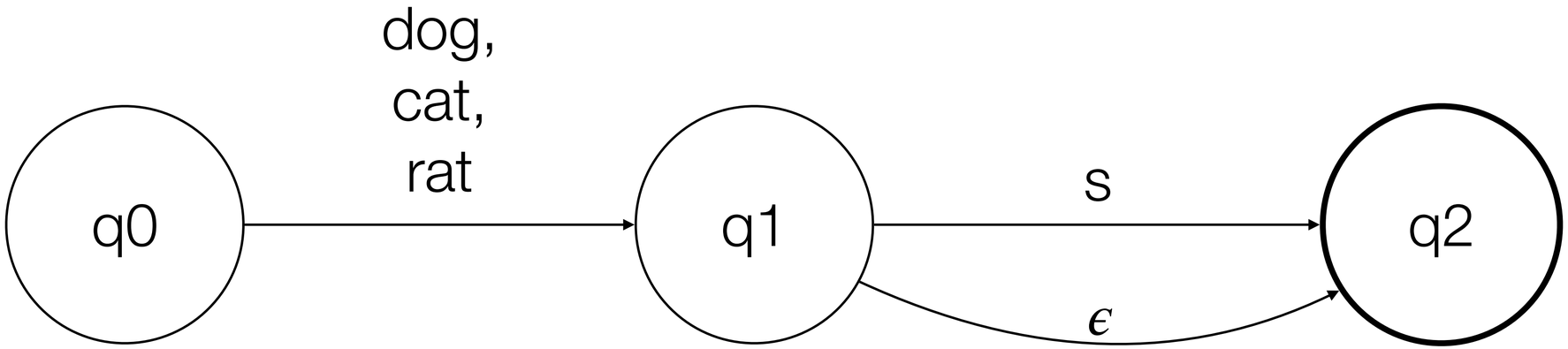
English plurals



Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

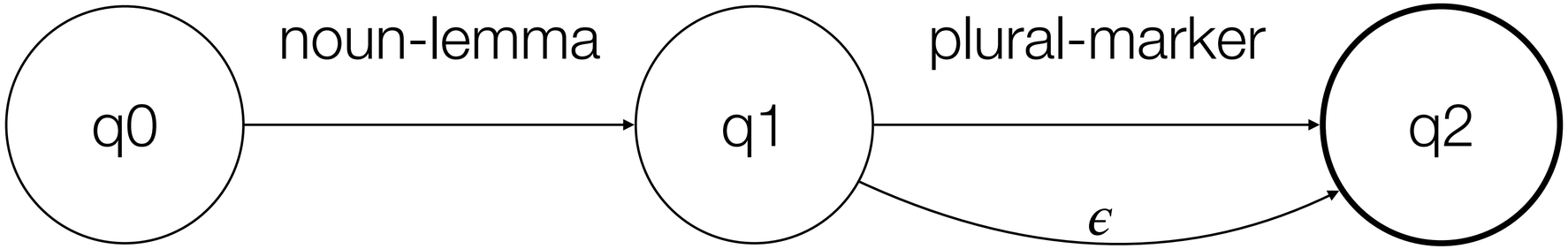


Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

noun-lemma: {cat, dog, rat}
plural-marker: {s}

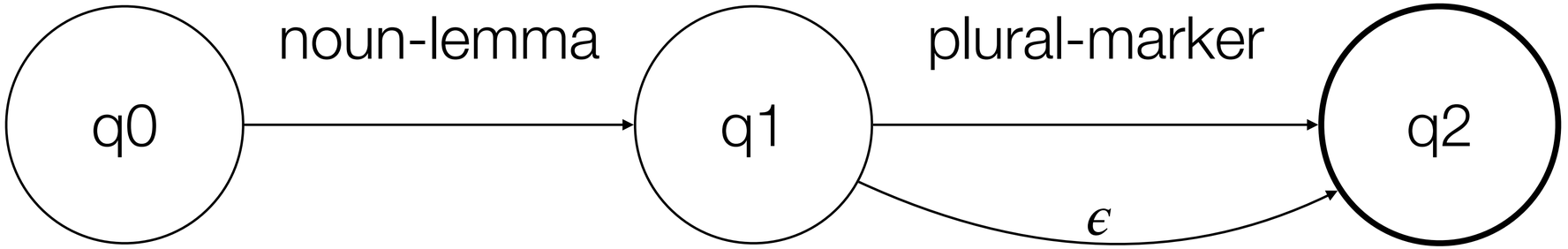


Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

noun-lemma: {cat, dog, rat, mouse}
plural-marker: {s}



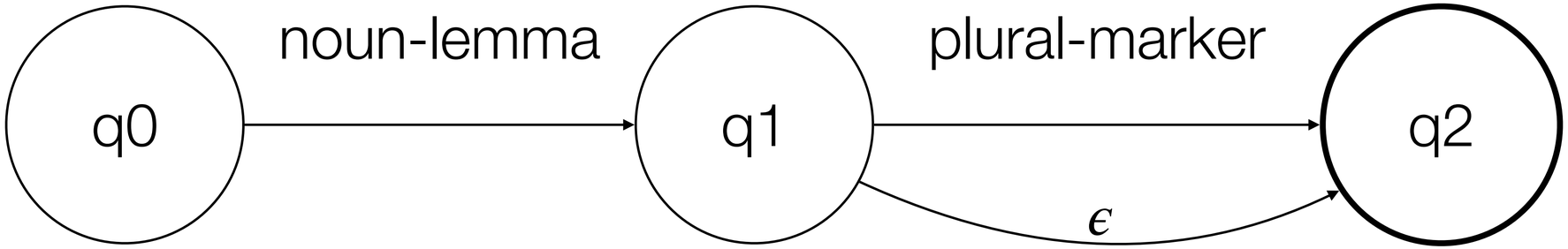
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

noun-lemma: {cat, dog, rat, mouse}
plural-marker: {s}

problems?



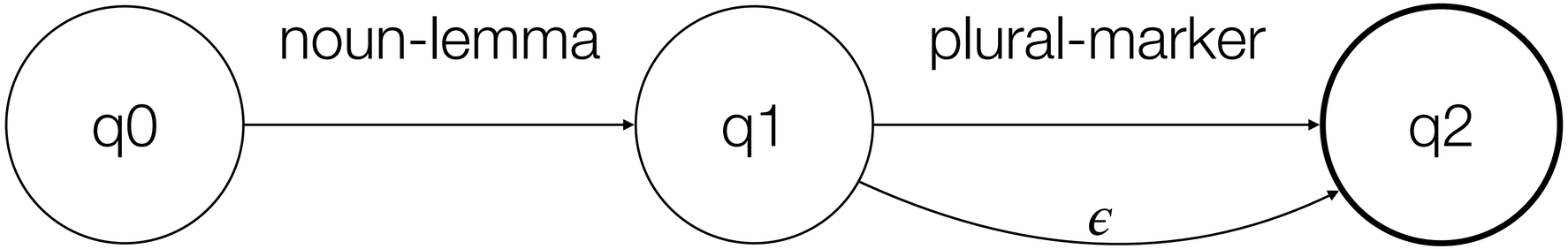
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

noun-lemma: {cat, dog, rat, mouse}
plural-marker: {s}

accepts "mouses"
doesn't accept "mice"



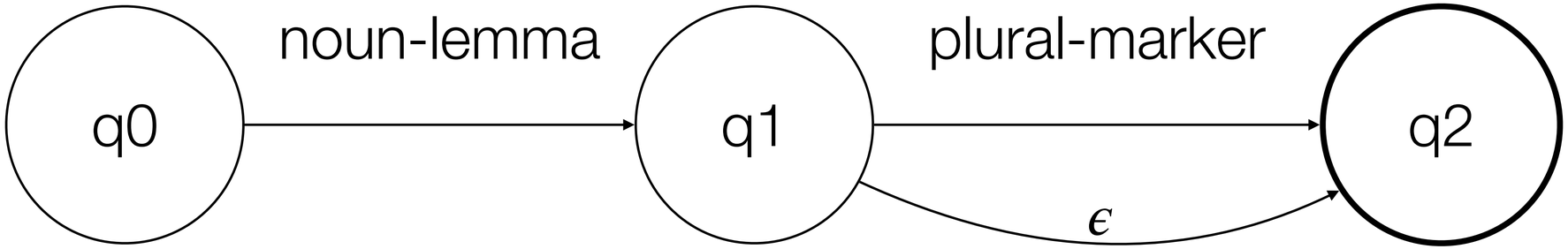
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

noun-lemma: {cat, dog, rat, mouse}
plural-marker: {s}

accepts "mouses"
doesn't accept "mice"



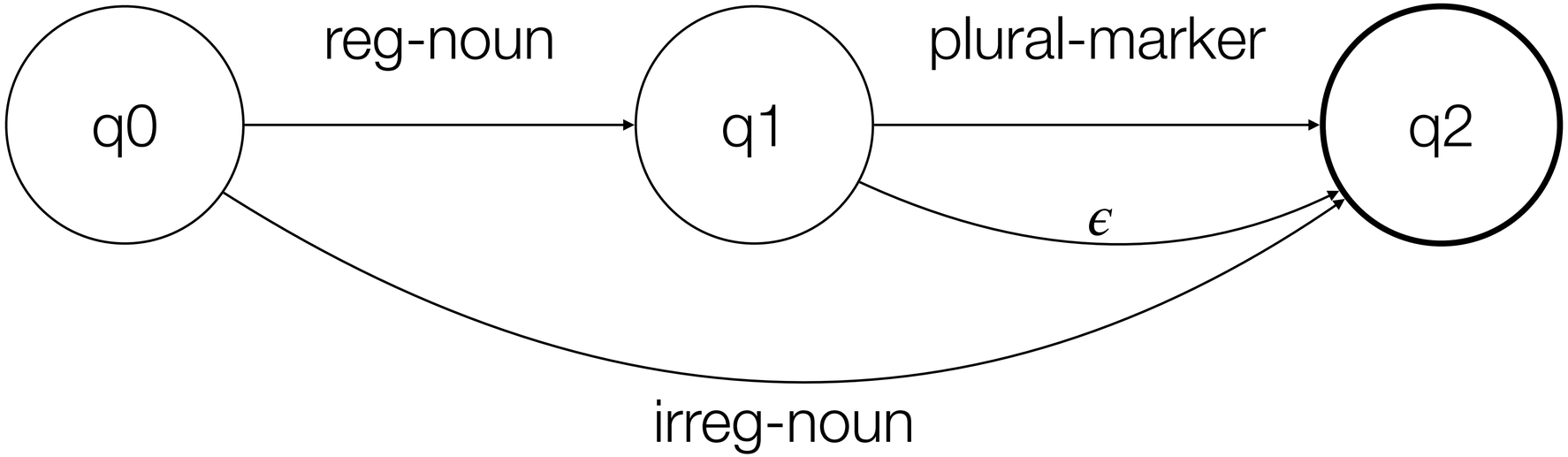
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-noun: {mouse}
plural-marker: {s}

accepts "mouses"
doesn't accept "mice"



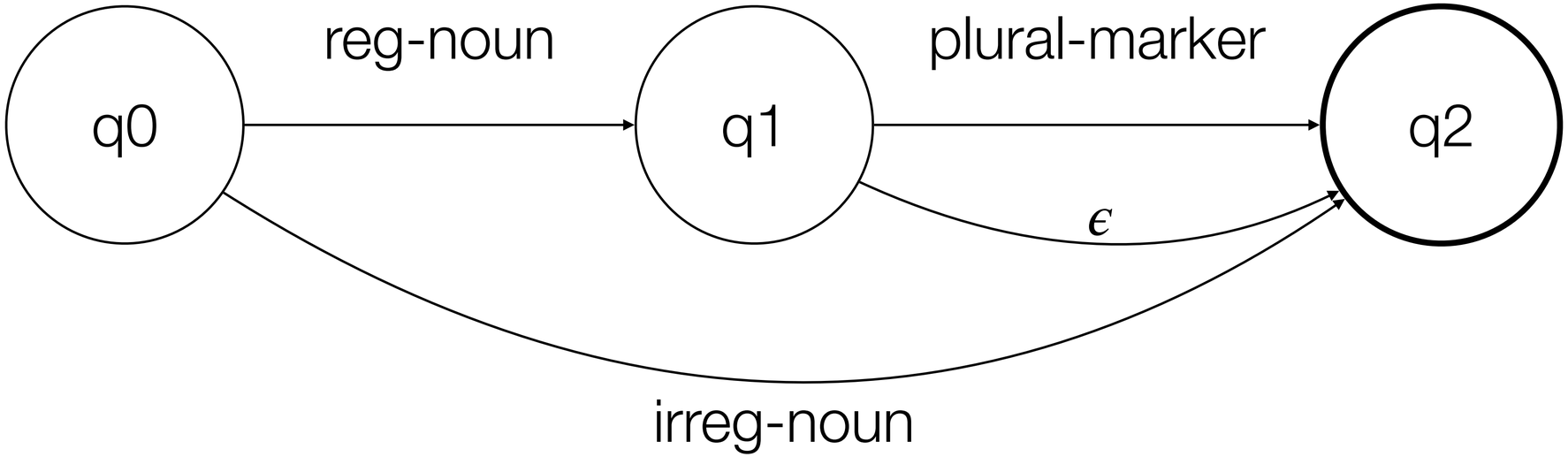
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-noun: {mouse}
plural-marker: {s}

~~accepts "mouses"~~
doesn't accept "mice"



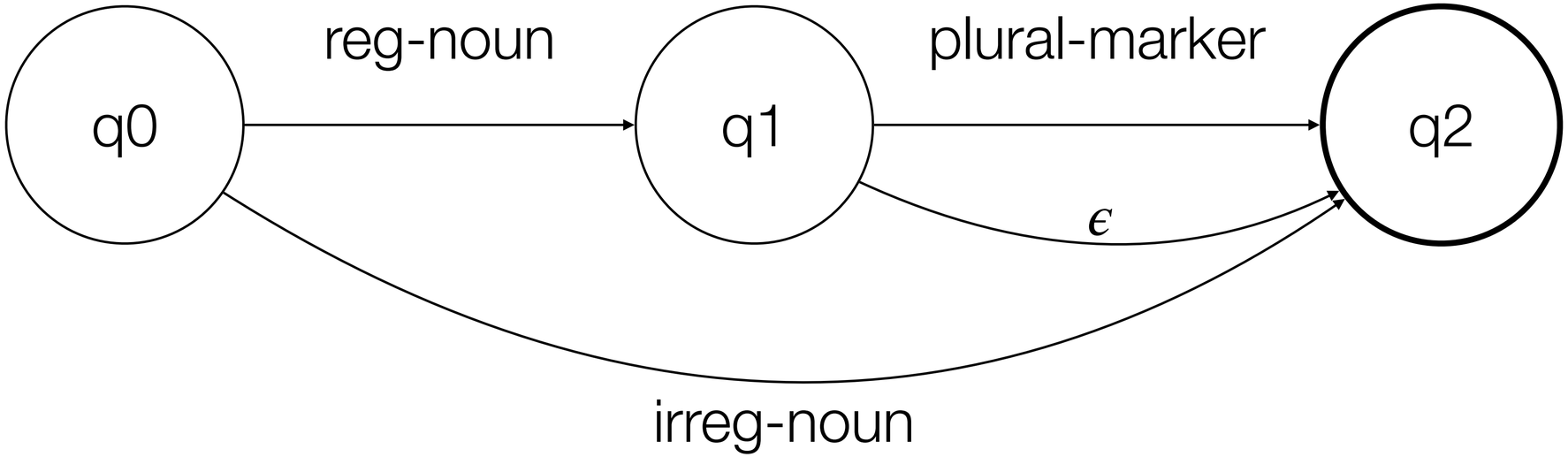
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-noun: {mouse, mice}
plural-marker: {s}

~~accepts "mouses"~~
doesn't accept "mice"



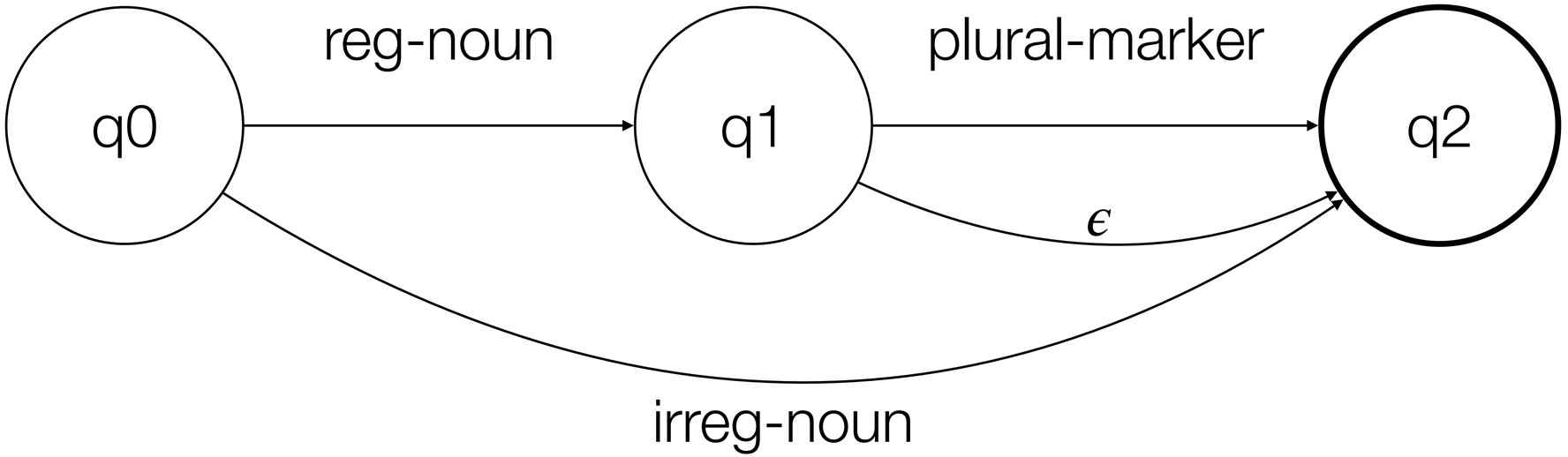
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-noun: {mouse, mice}
plural-marker: {s}

problems?



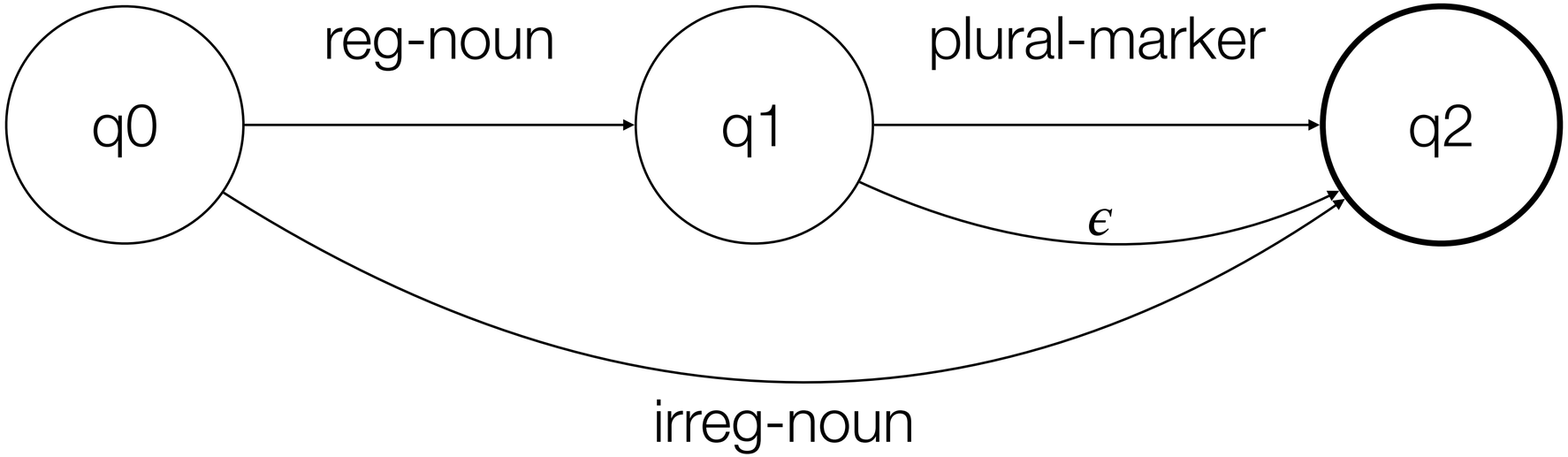
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-noun: {mouse, mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

dogmania, mouseathon, cataholic...



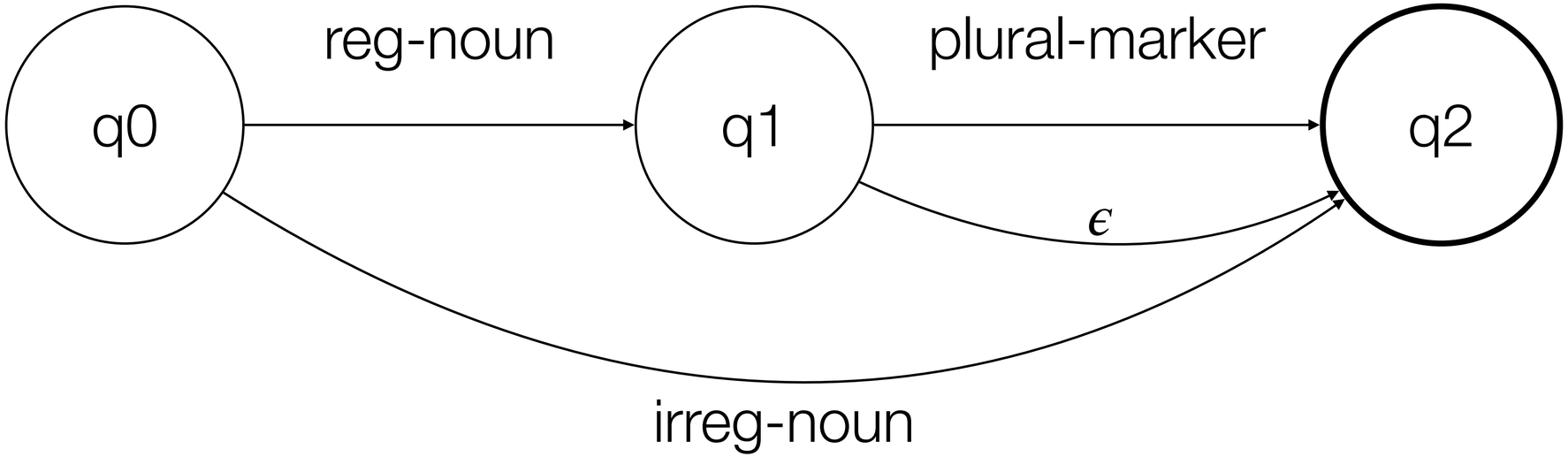
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-sing: {mouse}
irreg-plur: {mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

differentiate singular and plural...



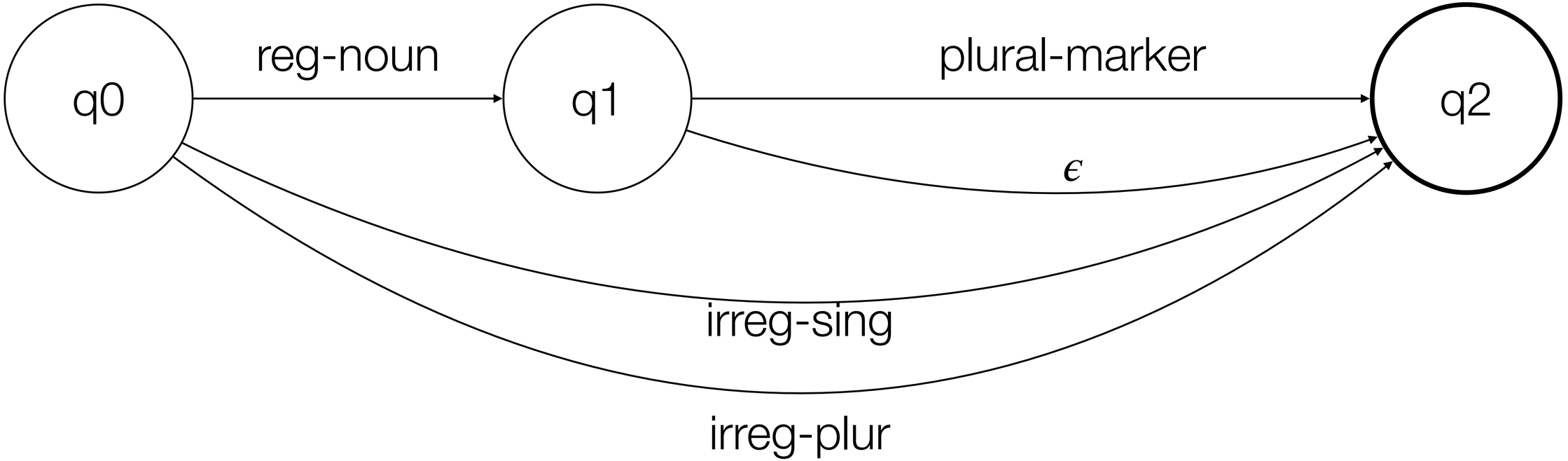
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-sing: {mouse}
irreg-plur: {mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

anyone can go straight to accept



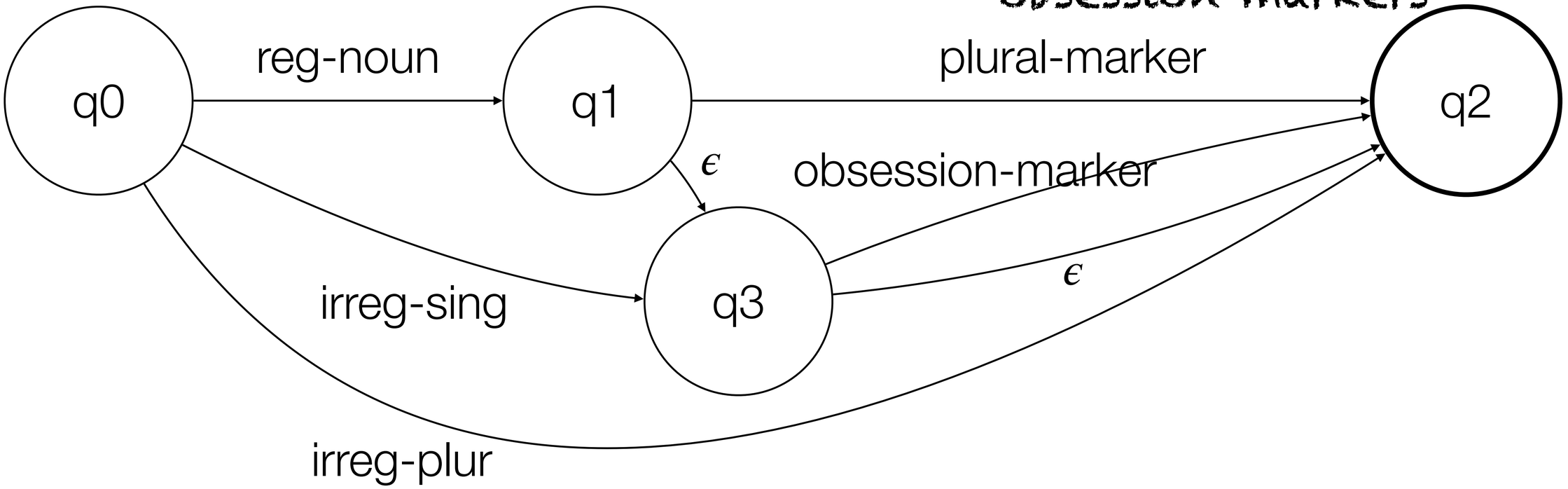
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-sing: {mouse}
irreg-plur: {mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

singular nouns can optionally add "obsession markers"



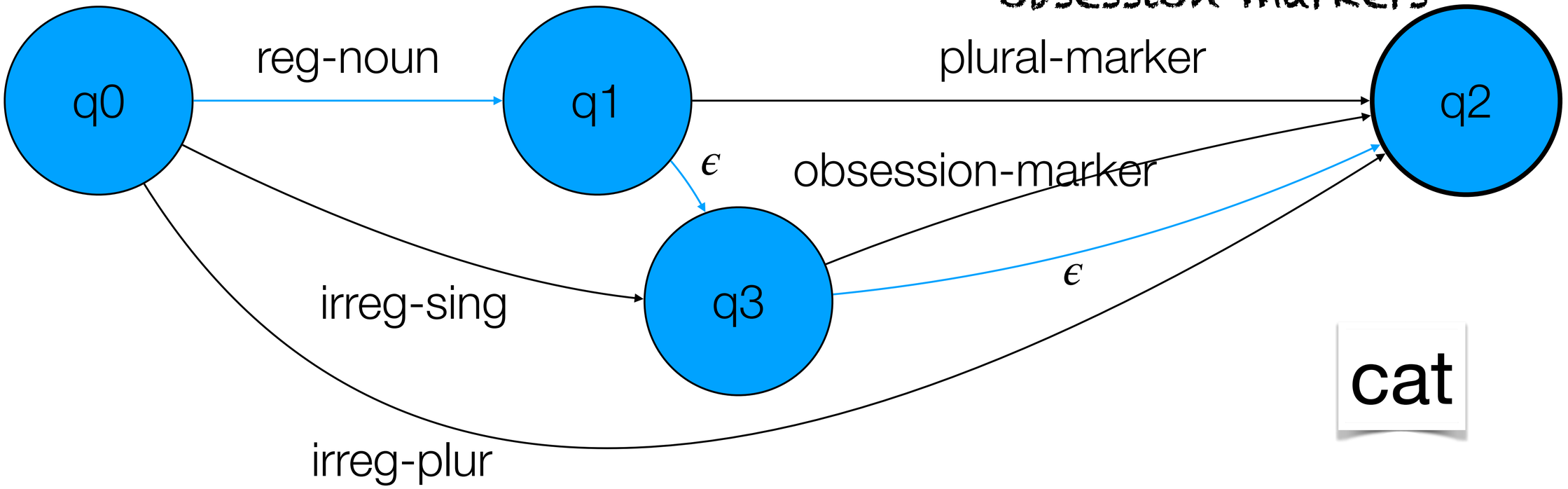
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-sing: {mouse}
irreg-plur: {mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

singular nouns can optionally add "obsession markers"



cat

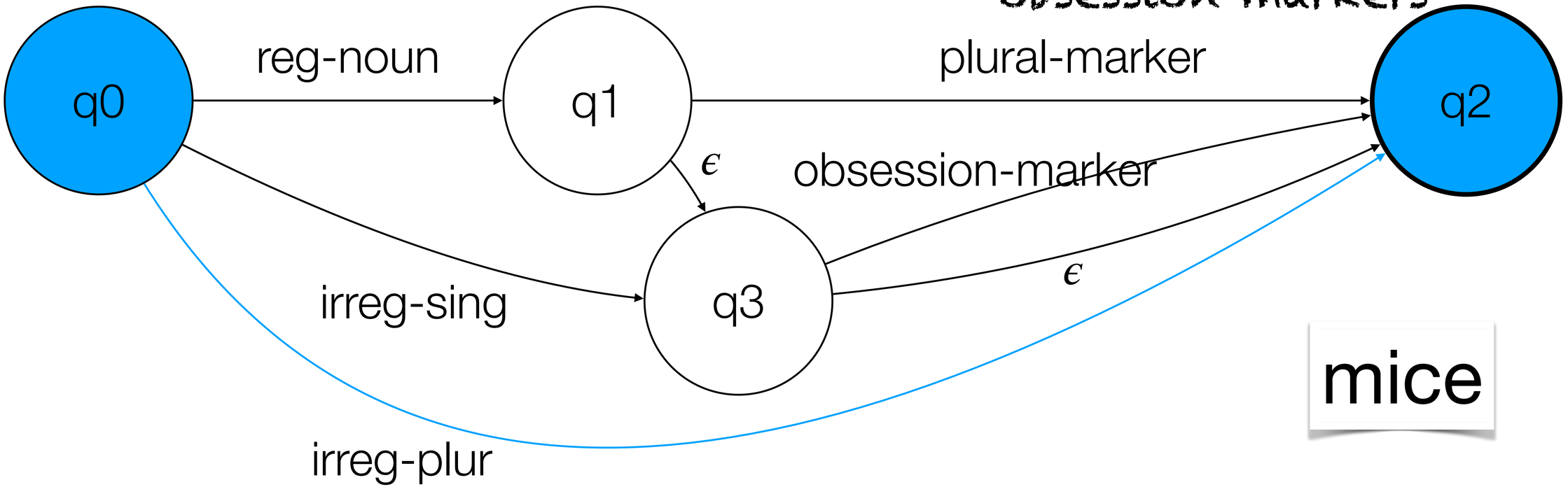
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-sing: {mouse}
irreg-plur: {mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

singular nouns can optionally add "obsession markers"



mice

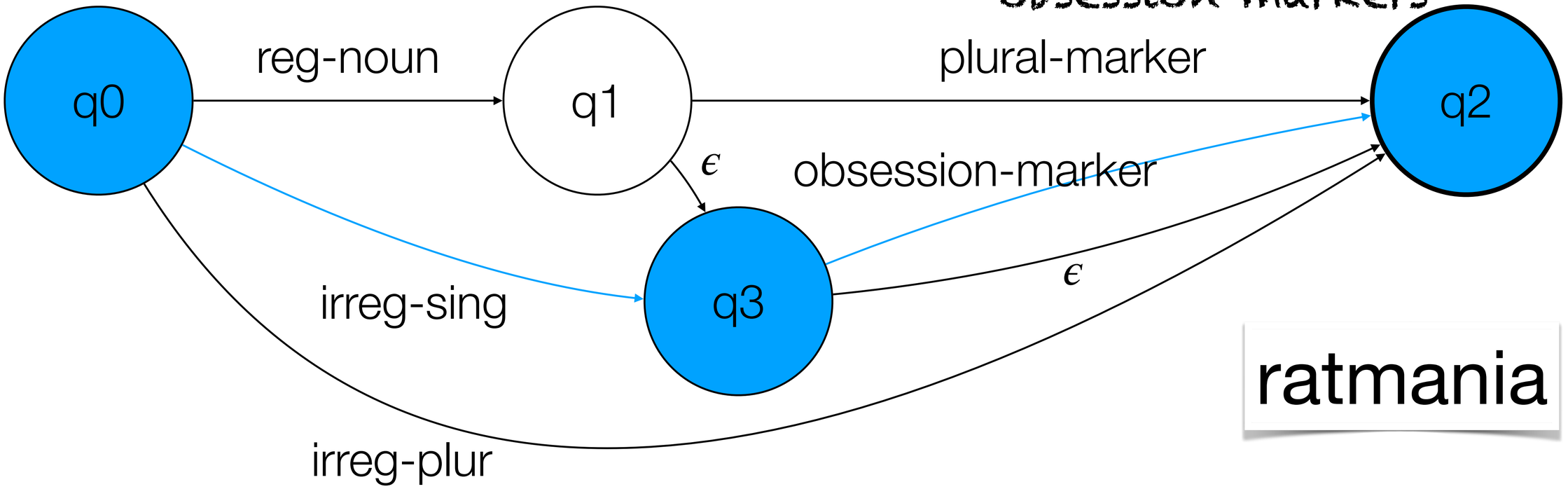
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-sing: {mouse}
irreg-plur: {mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

singular nouns can optionally add "obsession markers"



ratmania

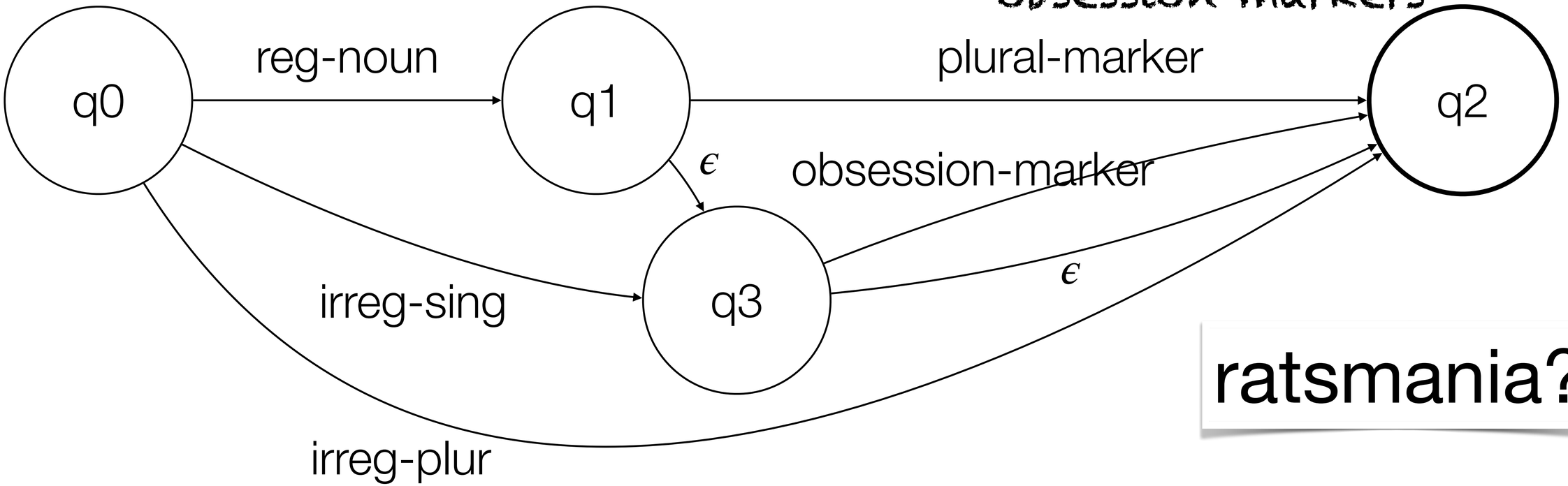
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-sing: {mouse}
irreg-plur: {mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

singular nouns can optionally add "obsession markers"



ratsmania?

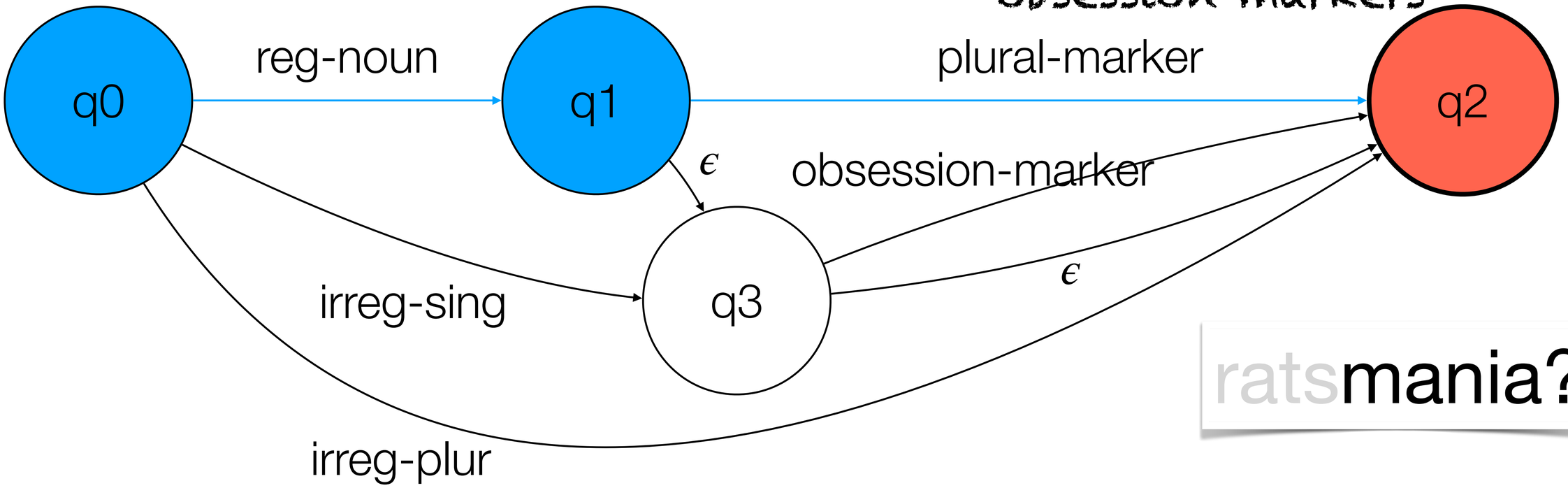
Goal: Accept valid English nouns (singular or plural)

Finite State Automata

English plurals

reg-noun: {cat, dog, rat}
irreg-sing: {mouse}
irreg-plur: {mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

singular nouns can optionally add "obsession markers"



ratsmania?

Goal: Accept valid English nouns (singular or plural)

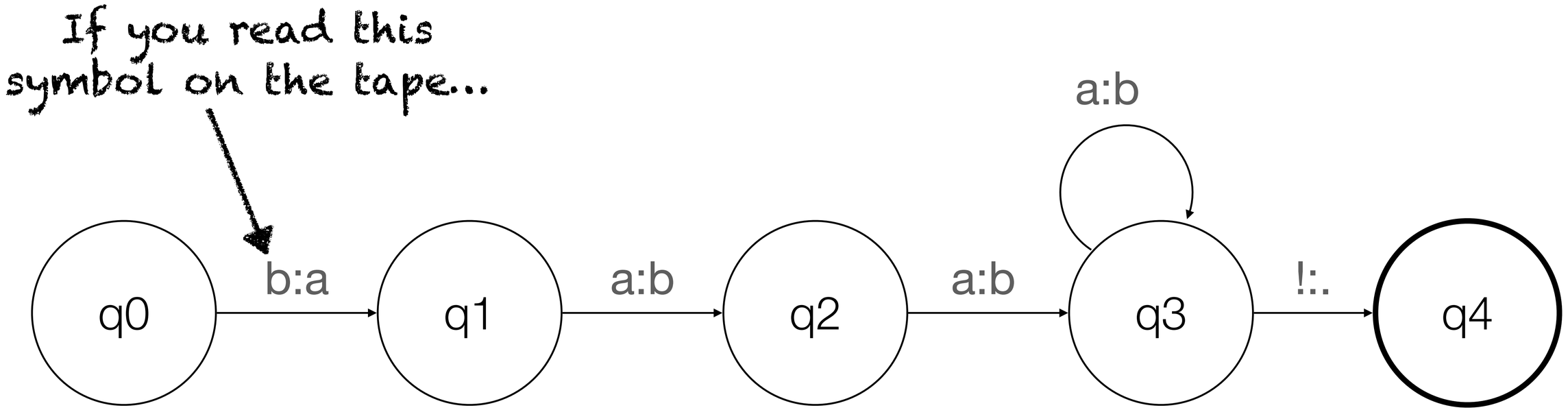
Finite State Transducers

- FSAs can tell you if a string follows the rules or not
- FSAs don’t give you the **parse**: i.e., record of the rules that a particular input uses
- FSTs: simple extension of FSAs which **produces a parse as output**

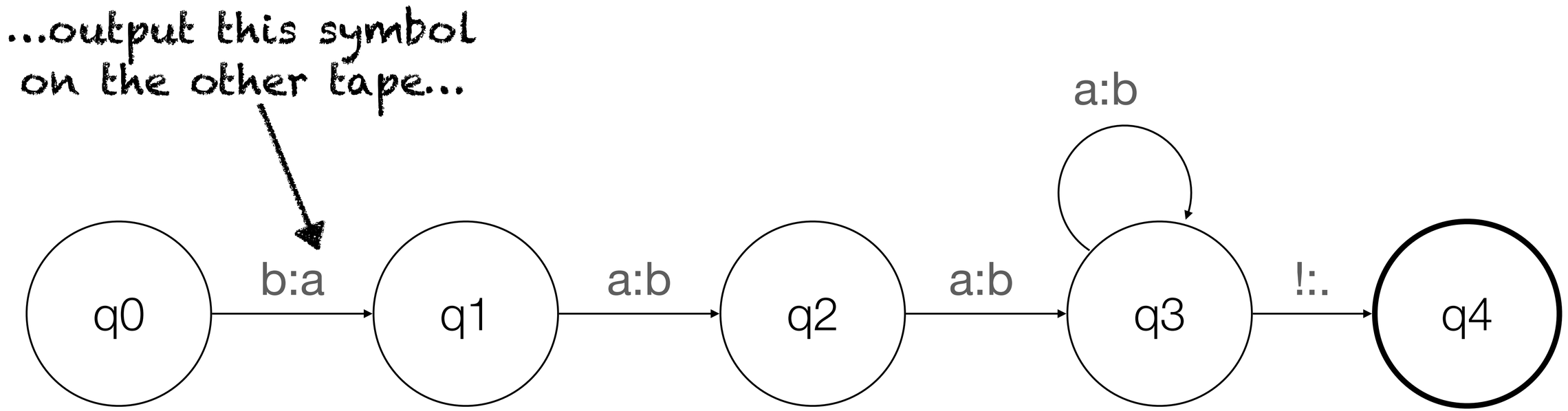
Finite State Transducers

- Two tapes: an input tape and an output tape
- Used to “translate” one language to another
- E.g., “translate” a word (cats) into a morphological parse (cat +N +pl)

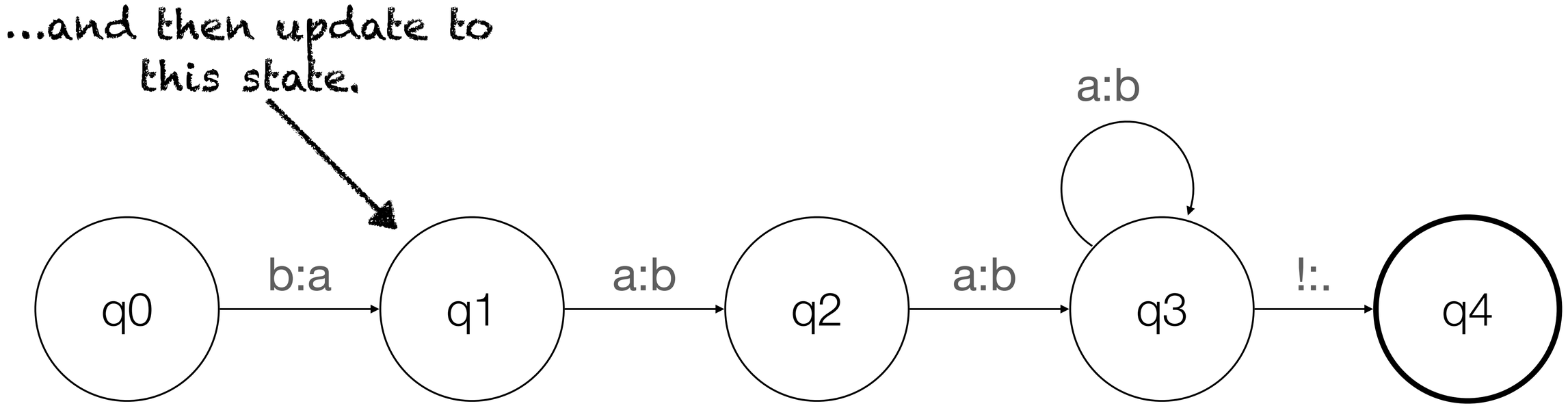
Finite State Transducers



Finite State Transducers

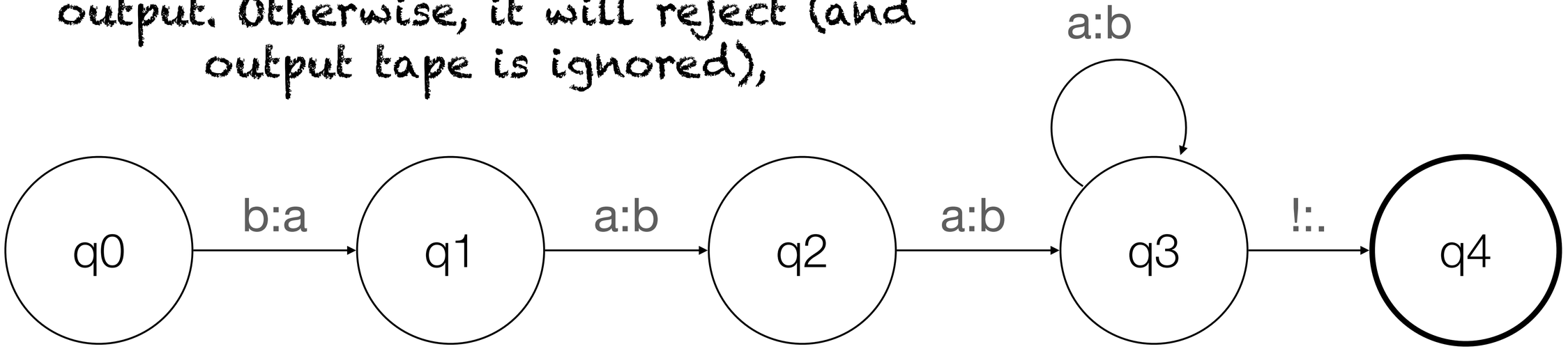


Finite State Transducers



Finite State Transducer

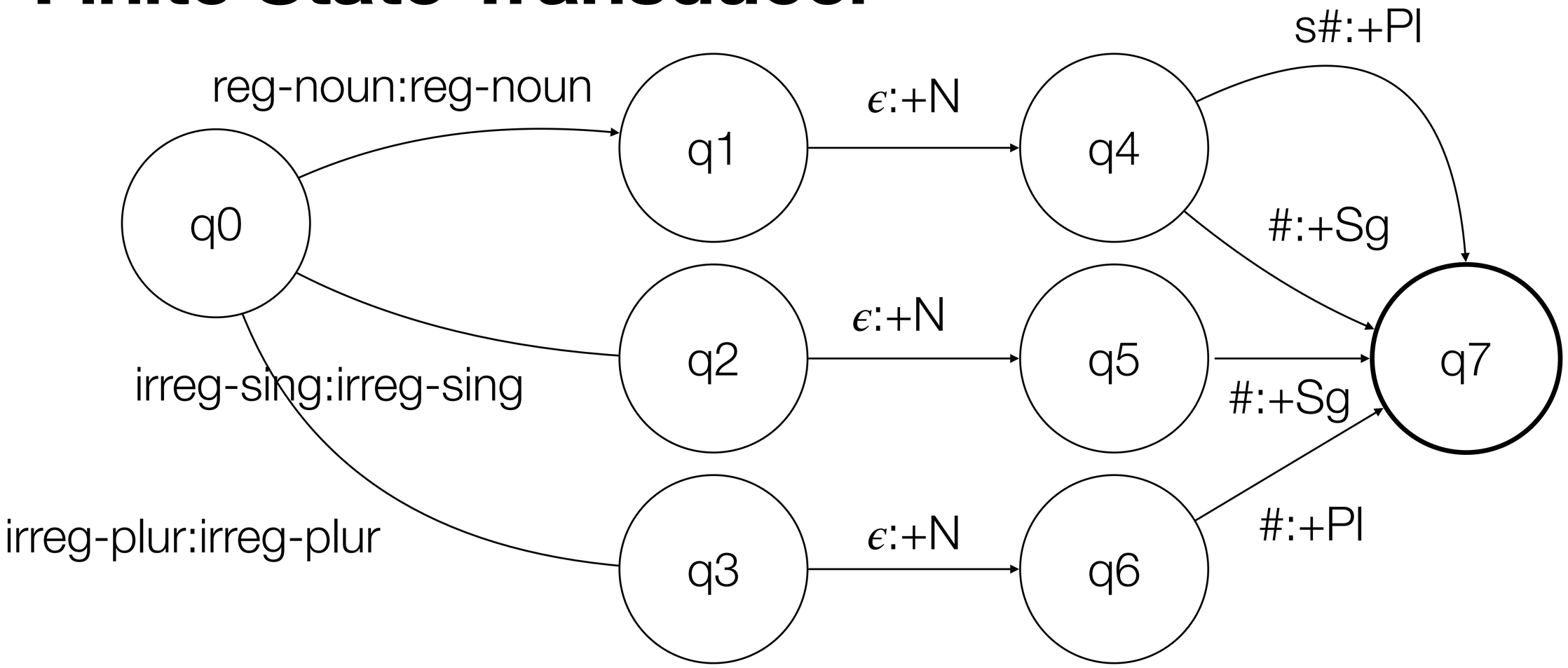
If input string is "accepted", the FST will end with a valid translation on the output. Otherwise, it will reject (and output tape is ignored),



b	a	a	a	!
a	b	b	b	.

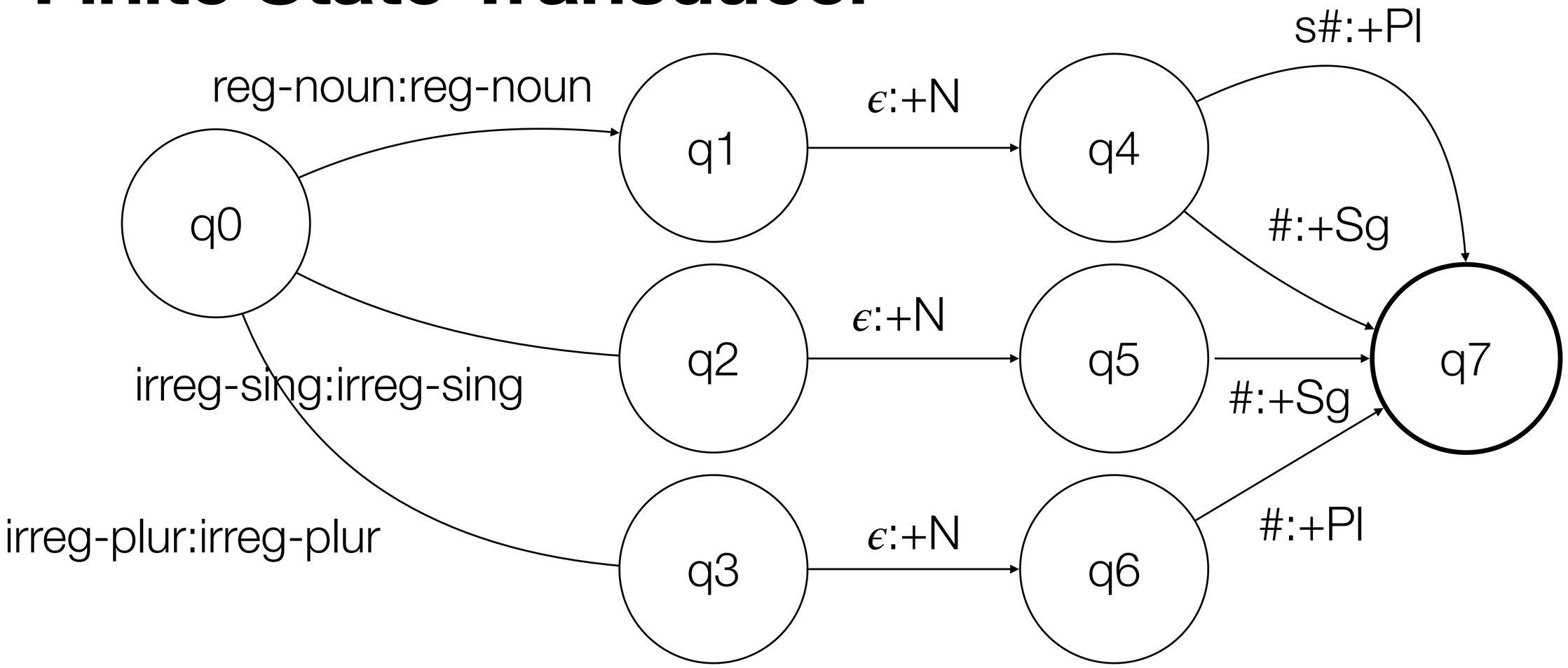
Finite State Transducer

= end of token



Finite State Transducer

= end of token

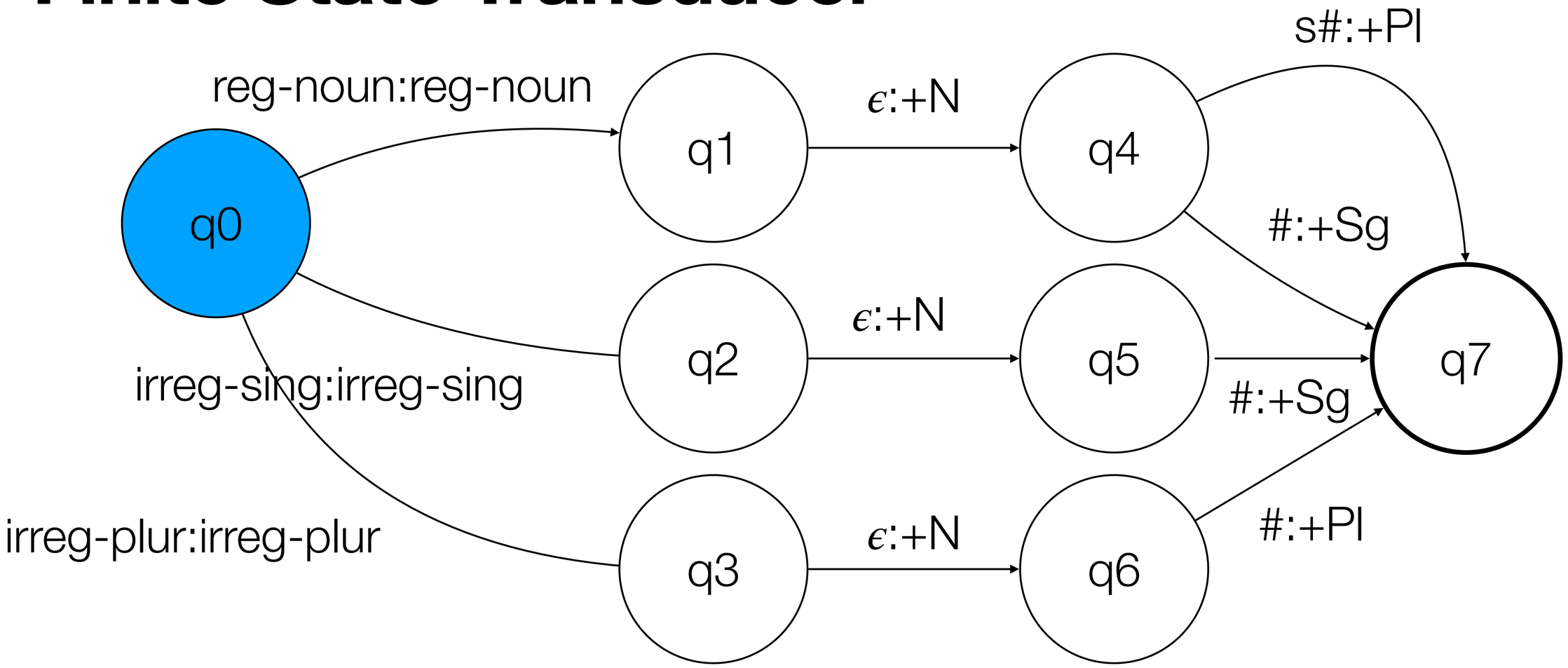


input tape: c a t s #

output tape:

Finite State Transducer

= end of token

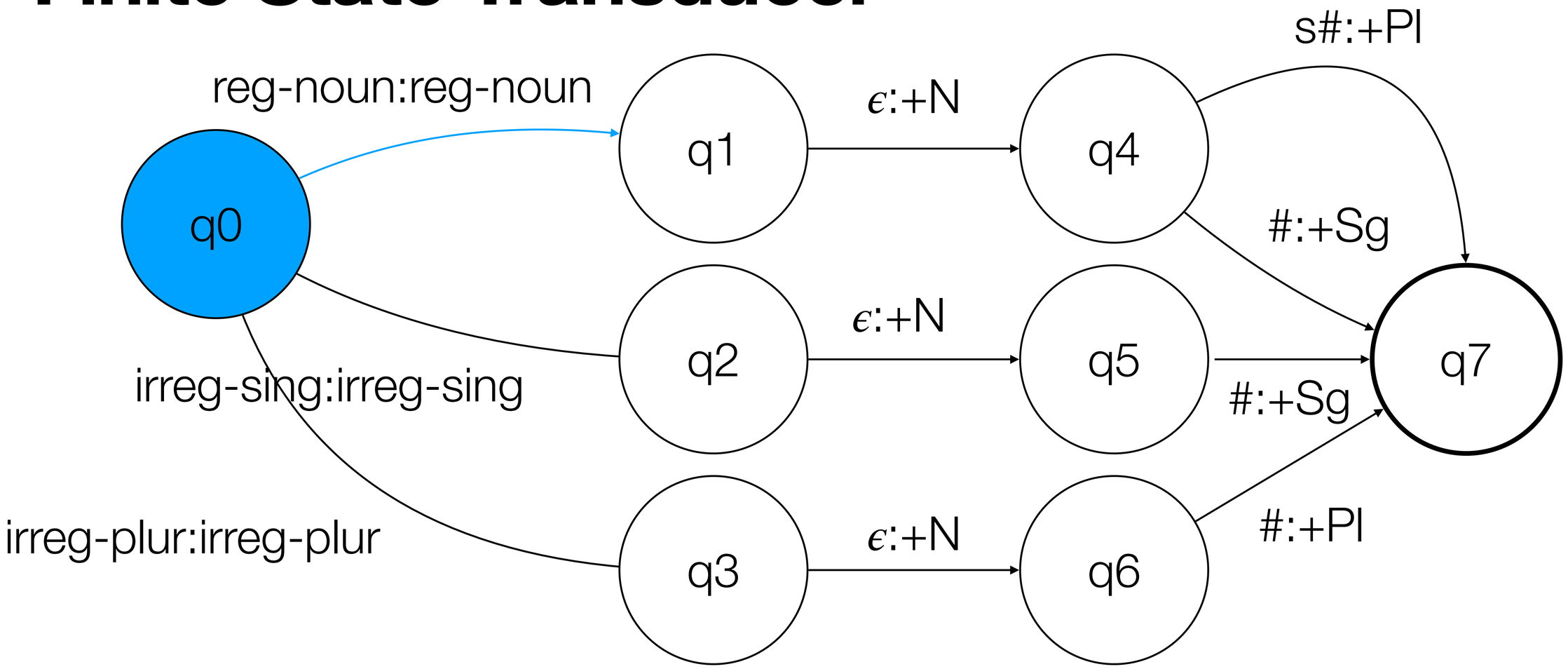


input tape: c a t s #

output tape:

Finite State Transducer

= end of token

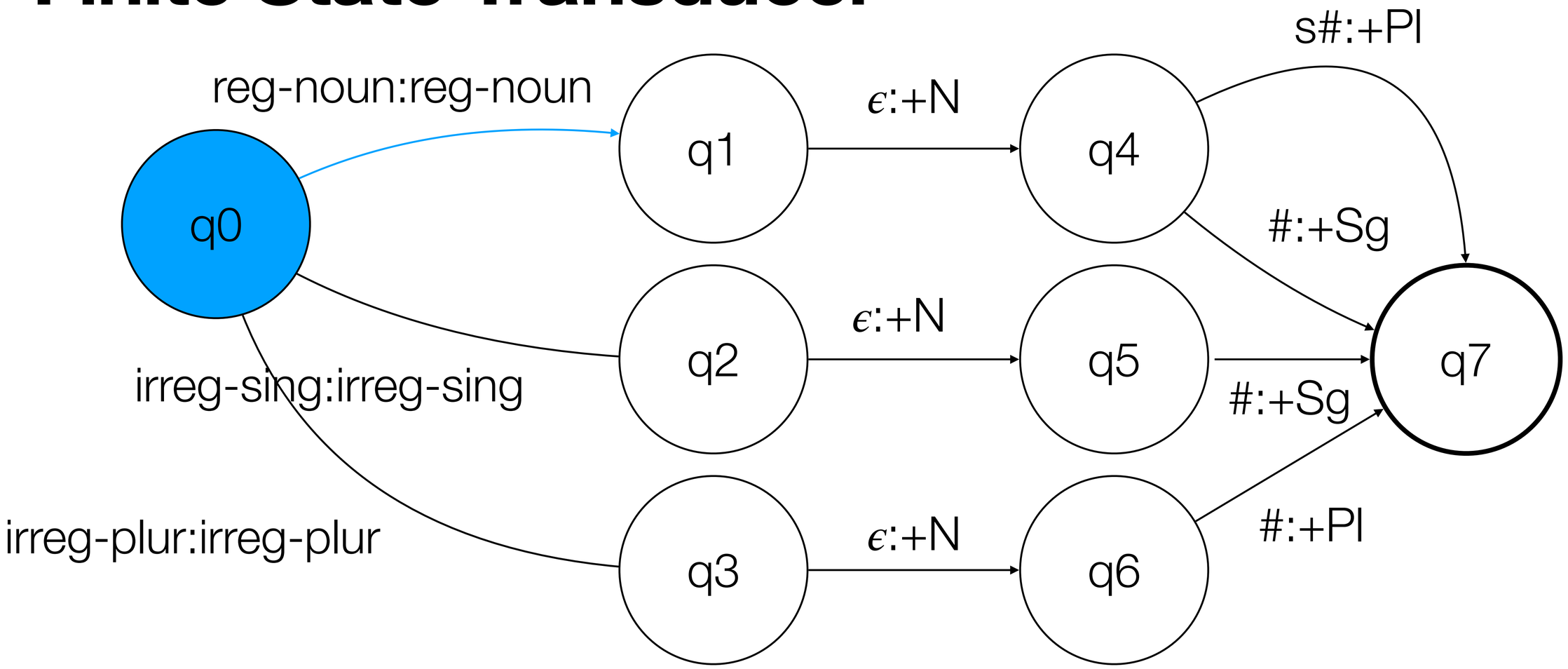


input tape: c a t s #

output tape:

Finite State Transducer

= end of token

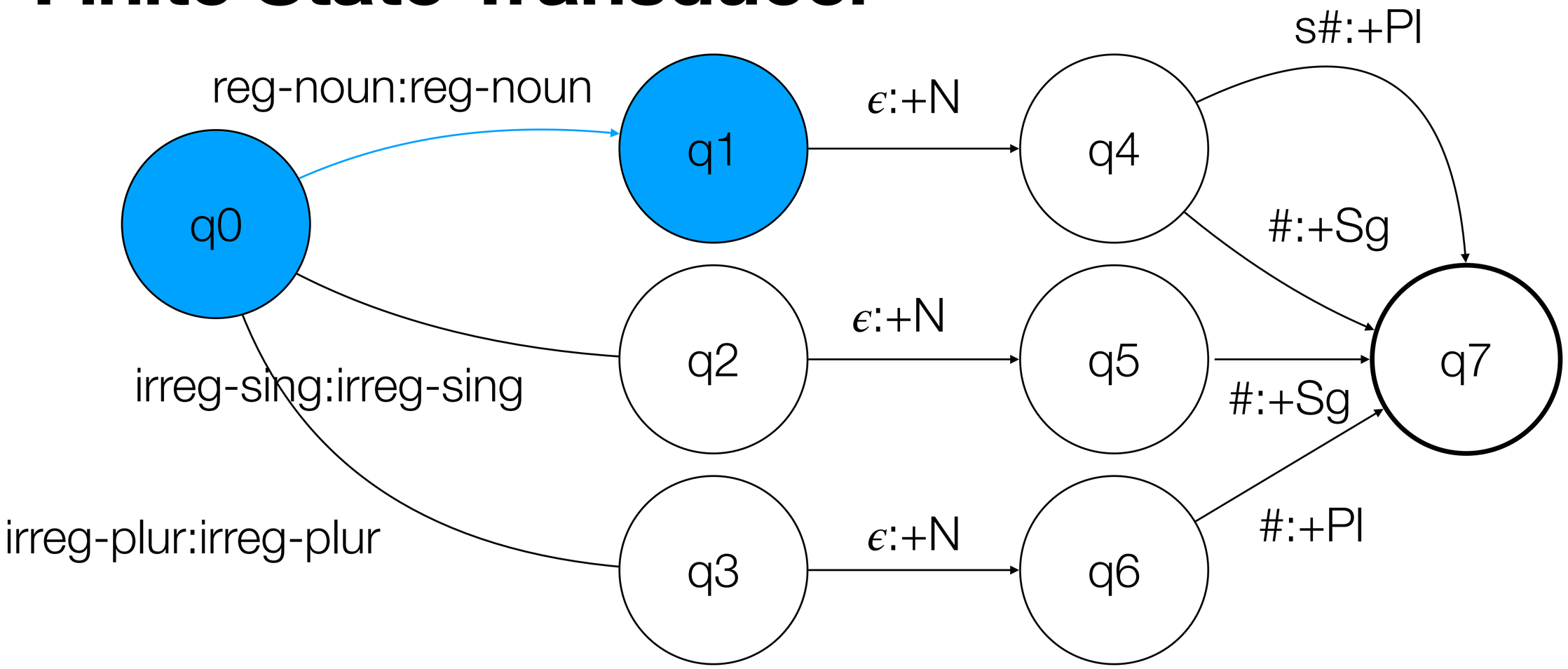


input tape: c a t s #

output tape: c a t

Finite State Transducer

= end of token

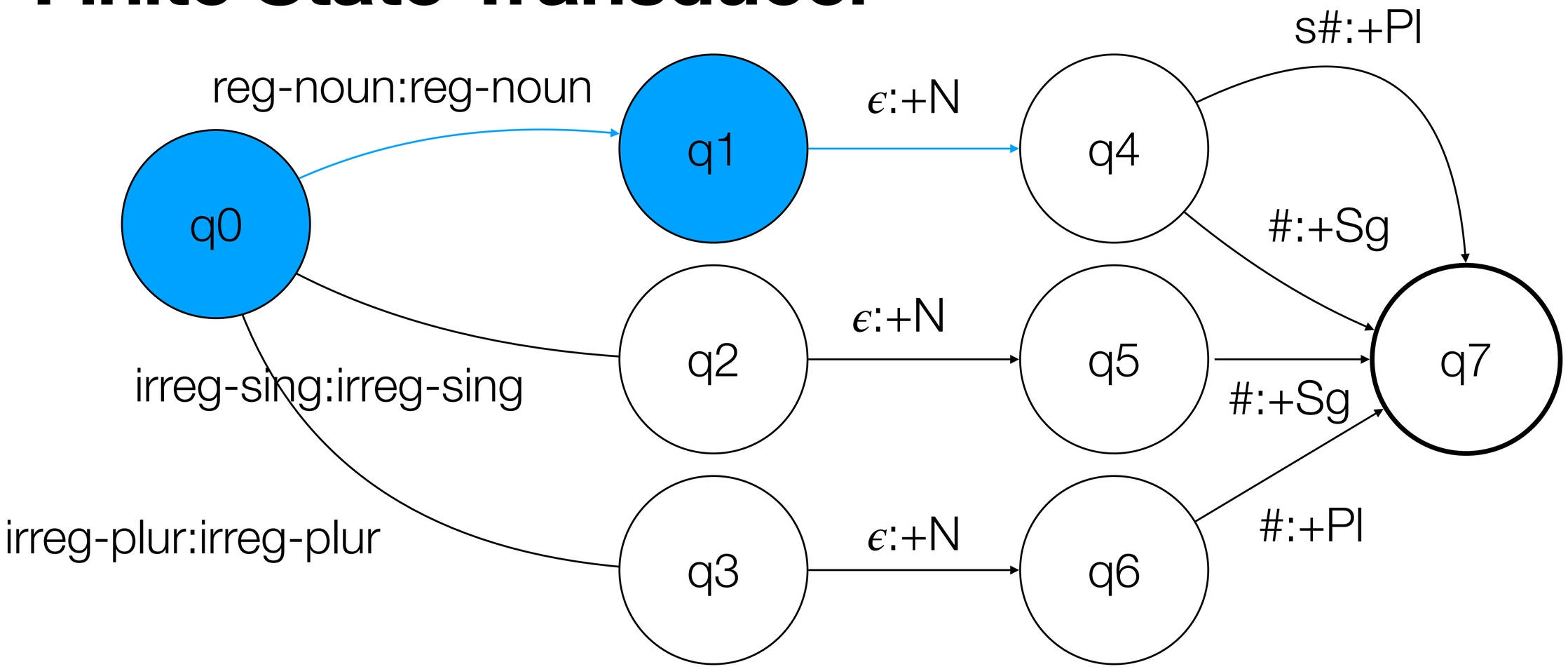


input tape: c a t s #

output tape: c a t

Finite State Transducer

= end of token

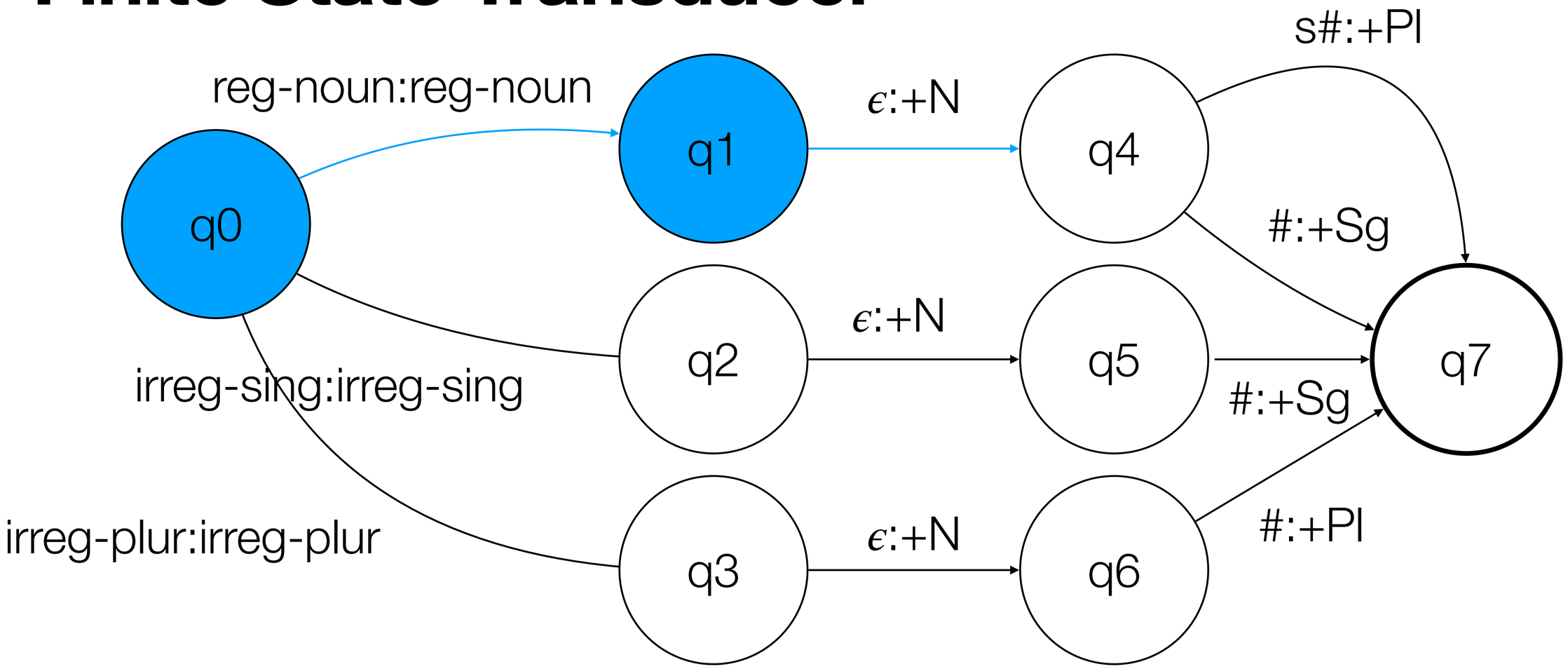


input tape: c a t s #

output tape: c a t

Finite State Transducer

= end of token

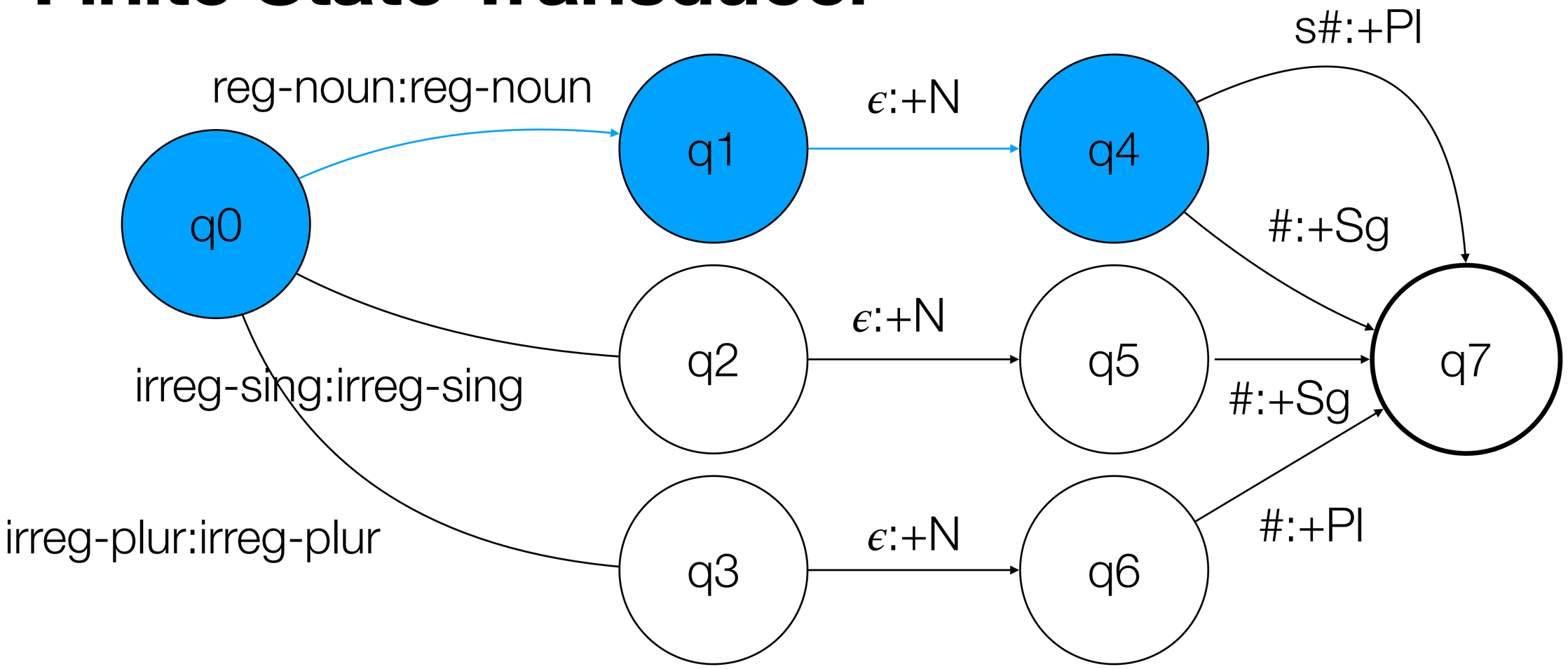


input tape: c a t s #

output tape: c a t +N

Finite State Transducer

= end of token

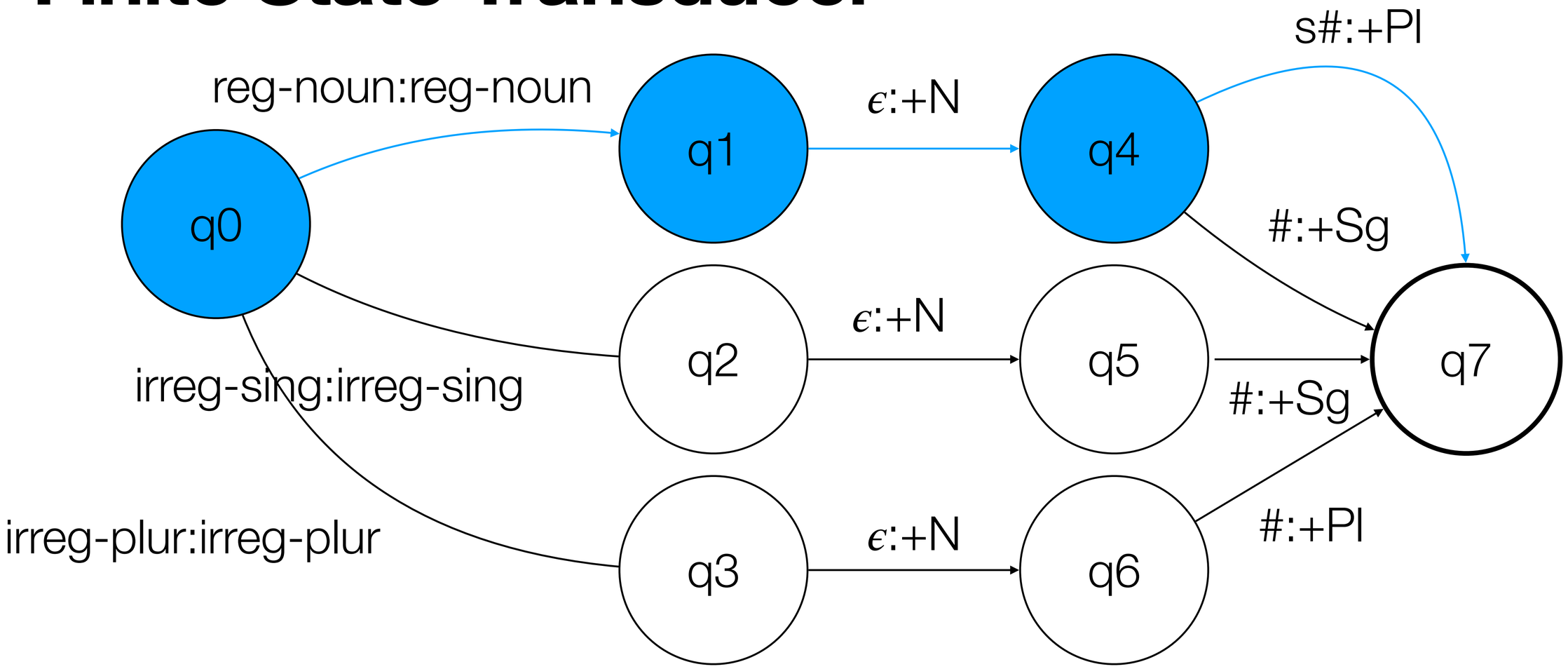


input tape: c a t s #

output tape: c a t +N

Finite State Transducer

= end of token

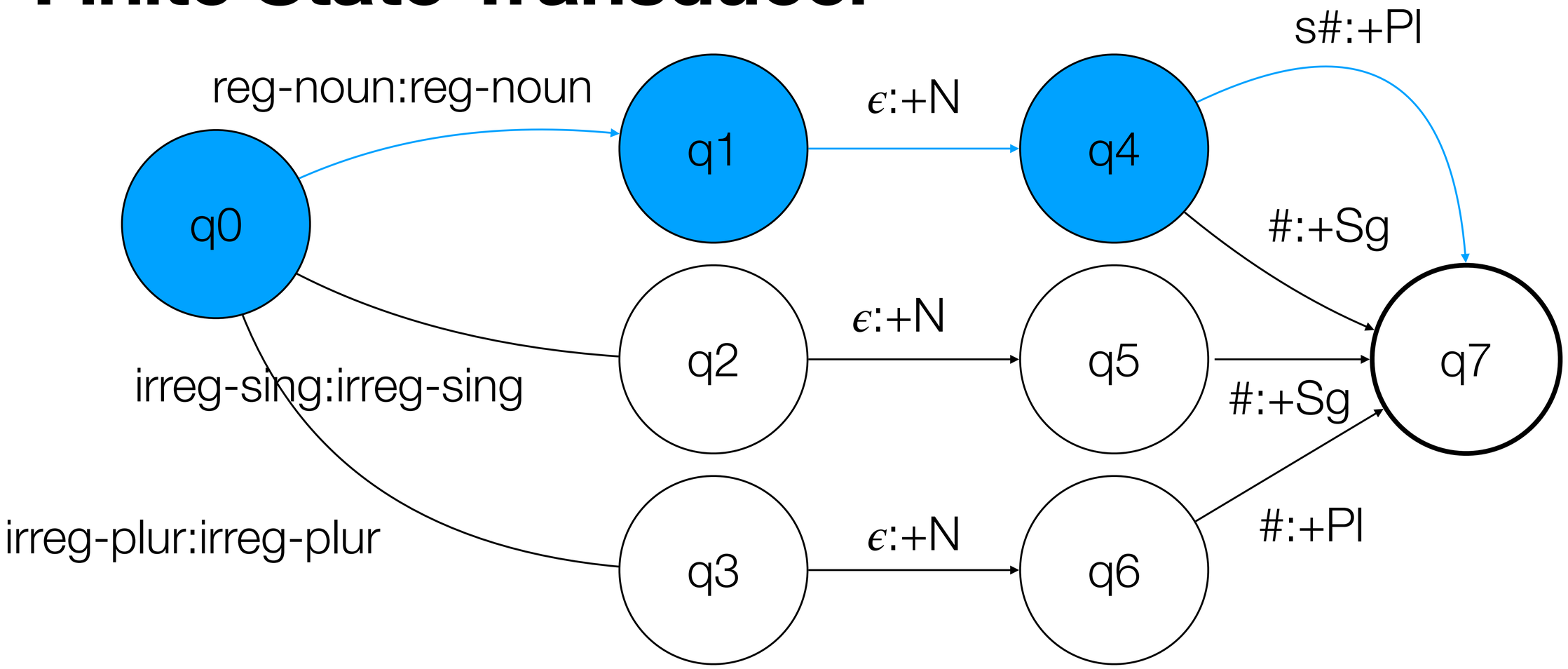


input tape: c a t s #

output tape: c a t +N

Finite State Transducer

= end of token

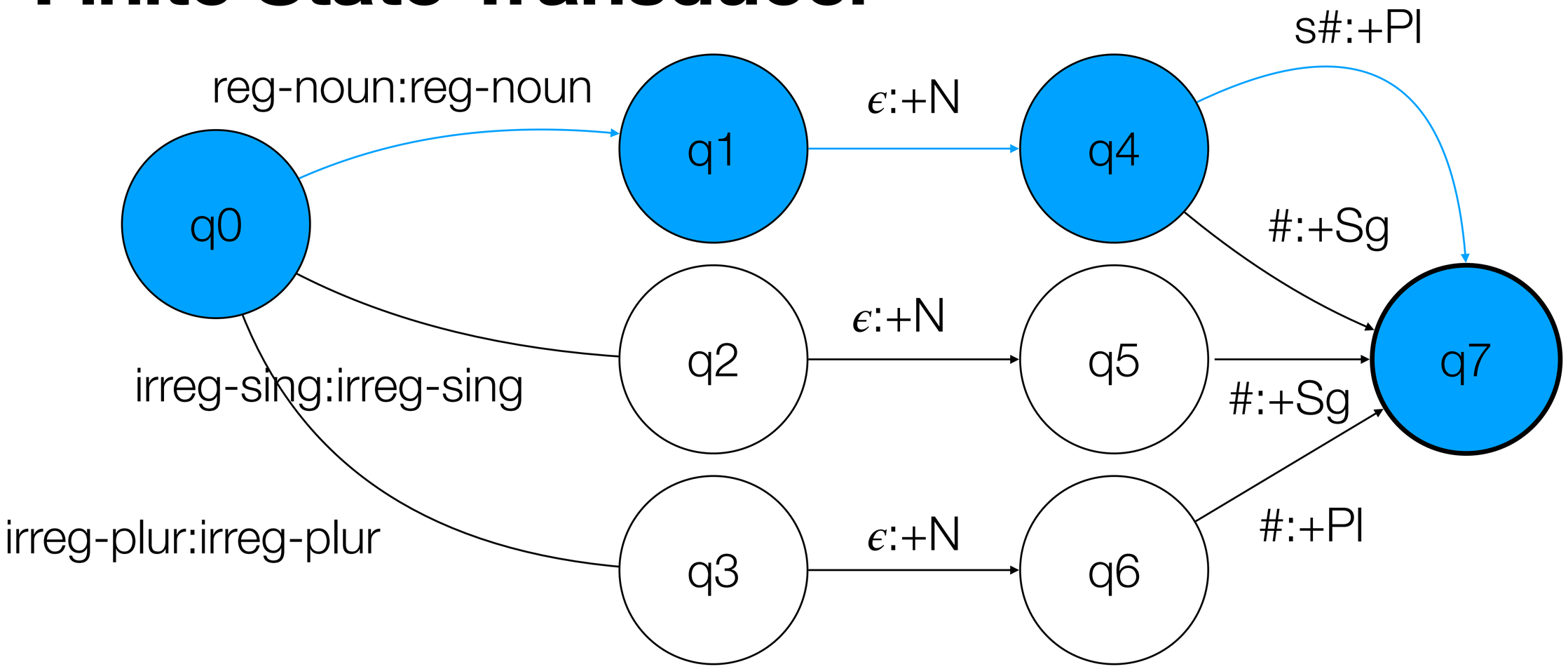


input tape: c a t s #

output tape: c a t +N +Pl

Finite State Transducer

= end of token



input tape: c a t s #

output tape: c a t +N +Pl

Topics

- (English) Morphology 101
- Finite State Machines
- **Subword Tokenization**
- [Saving for Later] Supporting all languages!

Subword Tokenization

- Problem:
  - Tokenizers result in lots of unknown words (<OOV>s)
  - Morphological analysis can help break, but:
    - Running morphological analyzers is expensive and language specific
      - Rare words might be “guessable” from cues other than standard linguistic analysis
        - “The food was disvoltable”

Subword Tokenization

- Solution:
  - “Subwords”
    - aka: “WordPieces”, “SentencePieces”, “BytePairEncoding”
  - Algorithmically identify common sub sequences of characters and merge them into atomic units
  - *Not* linguistically informed, based on data-compression algorithms
  - But works quite well in practice!

Subword Tokenization

Byte Pair Encoding

Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and **Barry Haddow** and **Alexandra Birch**
School of Informatics, University of Edinburgh
{rico.sennrich,a.birch}@ed.ac.uk, bhaddow@inf.ed.ac.uk

Subword Tokenization

Byte Pair Encoding

Algorithm 1 Learn BPE operations

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```


Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and **Barry Haddow** and **Alexandra Birch**
School of Informatics, University of Edinburgh
{rico.sennrich,a.birch}@ed.ac.uk, bhaddow@inf.ed.ac.uk

Break our words into characters.

Define a number of iterations to run ("hyperparameter")

Subword Tokenization

Byte Pair Encoding

Algorithm 1 Learn BPE operations

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and **Barry Haddow** and **Alexandra Birch**
School of Informatics, University of Edinburgh
{rico.sennrich,a.birch}@ed.ac.uk, bhaddow@inf.ed.ac.uk

Count the frequency of all the pairs of symbols

Subword Tokenization

Byte Pair Encoding

Algorithm 1 Learn BPE operations

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and **Barry Haddow** and **Alexandra Birch**
School of Informatics, University of Edinburgh
{rico.sennrich,a.birch}@ed.ac.uk, bhaddow@inf.ed.ac.uk

Merge the most frequent pair of characters, treating them as a new token

Subword Tokenization

Byte Pair Encoding

Algorithm 1 Learn BPE operations

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and **Barry Haddow** and **Alexandra Birch**
School of Informatics, University of Edinburgh
{rico.sennrich,a.birch}@ed.ac.uk, bhaddow@inf.ed.ac.uk

Repeat

Subword Tokenization

Byte Pair Encoding

1460 is fun

she is funny

they’re 14

$60 refund

Subword Tokenization

Byte Pair Encoding

1 4 6 0 # i s # f u n <eos>

s h e # i s # f u n n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e f u n d <eos>

Break our words into characters.

Subword Tokenization

Byte Pair Encoding

1 4 6 0 # i s # f u n <eos>

s h e # i s # f u n n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e f u n d <eos>

Build a vocabulary

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t}

Subword Tokenization

Byte Pair Encoding

Build a character count dictionary

1 4 6 0 # i s # f u n <eos>

s h e # i s # f u n n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e f u n d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t}

un	3
fu	3
# f	2
is	2
# i	2
60	2
re	2
0 #	2
e #	2
he	2
s #	2
14	2
{all the rest}	1

Subword Tokenization

Byte Pair Encoding

merge most frequent pair

1 4 6 0 # i s # f u n <eos>

s h e # i s # f u n n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e f u n d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t}

un	3
fu	3
# f	2
is	2
# i	2
60	2
re	2
0 #	2
e #	2
he	2
s #	2
14	2
{all the rest}	1

Subword Tokenization

Byte Pair Encoding

merge most frequent pair

1 4 6 0 # i s # f un <eos>

s h e # i s # f un n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e f un d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un}

merges = {(u,n)}

un	3
fu	3
# f	2
is	2
# i	2
60	2
re	2
0 #	2
e #	2
he	2
s #	2
14	2
{all the rest}	1

Subword Tokenization

Byte Pair Encoding

recompute pair frequencies

1 4 6 0 # i s # f un <eos>

s h e # i s # f un n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e f un d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un}

merges = {(u,n)}

f un	3
h e	2
r e	2
e #	2
i s	2
# f	2
# i	2
1 4	2
6 0	2
s #	2
0 #	2
{all the rest}	1

Subword Tokenization

Byte Pair Encoding

merge most frequent pair

1 4 6 0 # i s # f un <eos>

s h e # i s # f un n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e f un d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un}

merges = {(u,n)}

f un	3
h e	2
r e	2
e #	2
i s	2
# f	2
# i	2
1 4	2
6 0	2
s #	2
0 #	2
{all the rest}	1

Subword Tokenization

Byte Pair Encoding

merge most frequent pair

1 4 6 0 # i s # fun <eos>

s h e # i s # fun n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e fun d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un fun}

merges = {(u,n), (f, un)}

f un	3
h e	2
r e	2
e #	2
i s	2
# f	2
# i	2
1 4	2
6 0	2
s #	2
0 #	2
{all the rest}	1

Subword Tokenization

Byte Pair Encoding

recompute pair frequencies

1 4 6 0 # i s # fun <eos>

s h e # i s # fun n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e fun d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un fun}

merges = {(u,n), (f, un)}

r e	2
e #	2
1 4	2
i s	2
h e	2
0 #	2
# i	2
6 0	2
s #	2
# fun	2
{all the rest}	1

Subword Tokenization

Byte Pair Encoding

merge most frequent pair

1 4 6 0 # i s # fun <eos>

s h e # i s # fun n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e fun d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un fun}

merges = {(u,n), (f, un)}

r e	2
e #	2
1 4	2
i s	2
h e	2
0 #	2
# i	2
6 0	2
s #	2
# fun	2
{all the rest}	1

Subword Tokenization

Byte Pair Encoding

merge most frequent pair

1 4 6 0 # i s # fun <eos>

s h e # i s # fun n y <eos>

t h e y ’ re # 1 4 <eos>

$ 6 0 # re fun d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un fun re}

merges = {(u,n), (f, un), (r, e)}

r e	2
e #	2
1 4	2
i s	2
h e	2
0 #	2
# i	2
6 0	2
s #	2
# fun	2
{all the rest}	1

Subword Tokenization

Byte Pair Encoding

six more iterations

14 60# is# fun <eos>

s he # is# fun n y <eos>

t he y ’ re # 14 <eos>

$ 60# re fun d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#)}

Subword Tokenization

Byte Pair Encoding

seven more iterations

14 60# is#fun <eos>

s he # is#fun n y <eos>

t he y ’ re # 14 <eos>

$ 60# re fun d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

140 red fungi

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

1 4 0 # r e d # f u n g i

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

1 4 0 # r e d # f un g i

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

1 4 0 # r e d # fun g i

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

1 4 0 # re d # fun g i

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

14 0 # re d # fun g i

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

14 0 # re d # fun g i

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

14 0 # re d # fun g i

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

14 0 # re d # fun g i

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

14 0# re d # fun g i

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

14 0# re d # fun g i

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

14 0# re d # fun g i

vocab = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t un fun re 14 is is# he 0# 60#
is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

Byte Pair Encoding

14 0# re d # fun <oov> i

nit: in this toy example, "g" is actually <oov>.

In reality, character vocabs are very large so nothing is really <oov>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un fun re 14 is is# he 0# 60# is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #) (h, e) (0, #) (6, 0#) (is#, fun)}

Subword Tokenization

vs. Morphological Analysis

she is funny <eos>

s he # is#fun n y <eos>

she be+1p+sing fun+adj <eos>

they’re 14 <eos>

t he y ’ re # 14 <eos>

they be+1p+sing 14 <eos>