Regular Expressions:

Our misunderstood friends

Steven Bedrick

CS/EE 5/655, 10/1/14

Simpsons Episode 7G03, “Homer’s Odyssey”

Plan for the day:

1. What is a regular expression?
2. Finite-state background
3. Tying them together
4. More advanced patterns
5. Useful tools
6. Regexes in Python

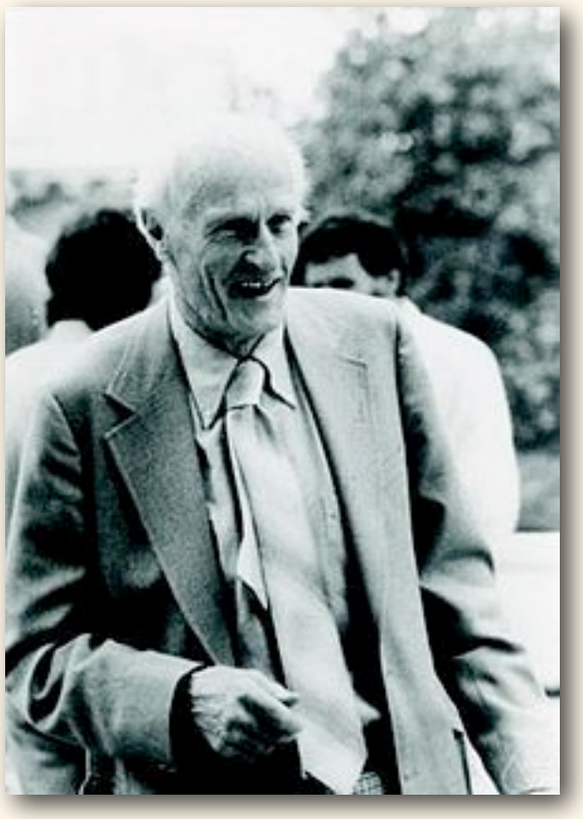
What is a regular expression?

In common usage: a programming construct for pattern matching.

http://xkcd.com/208/

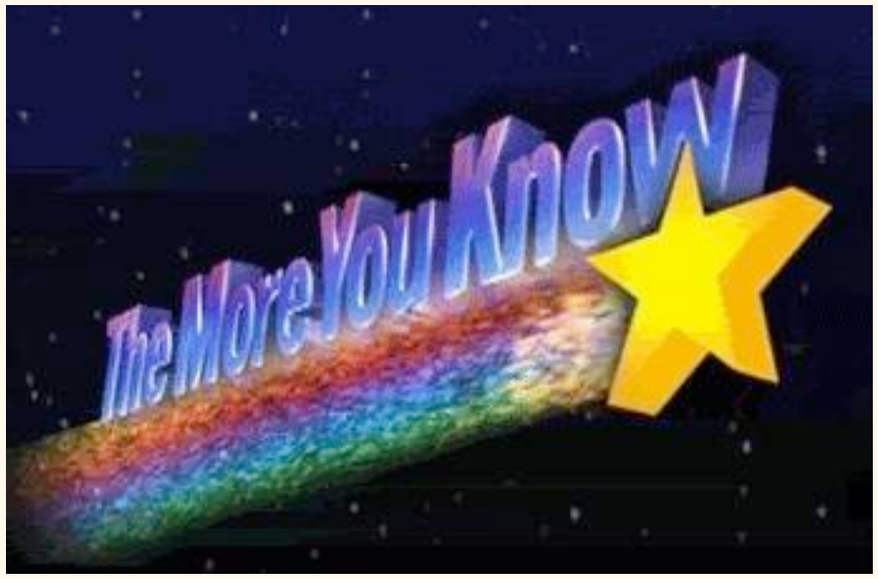
What is a regular expression?

More formally: an algebraic notation for characterizing a set of strings.



Stephen Kleene
1909–1994

Fun fact: “Kleene” is apparently pronounced /ˈkleɪniː/



What is a regular expression?

Even more formally: an algebraic notation for describing the grammar of a *regular language*.

What does that mean?

Quick bit of formal language theory:

A “formal language” can be thought of as:

Sets of strings of symbols...

... rules about how they may be combined.

For example:

Language *L* includes the alphabet:

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, =\}$$

Non-empty strings that do not contain “+” or “=” or start with “0” are in *L*;

The string “0” is in *L*;

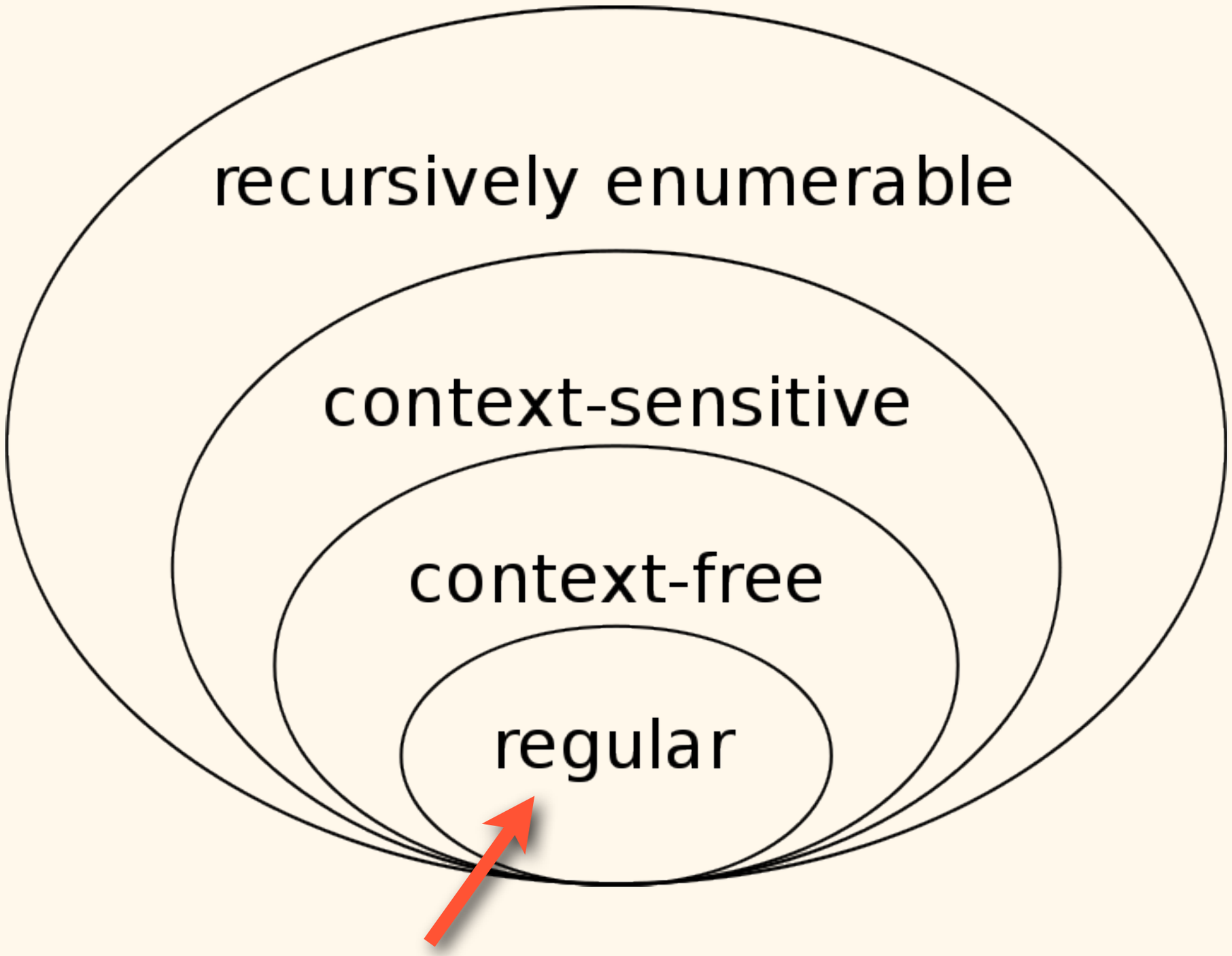
A string containing “=” is in *L* iff there is exactly one “=” and it separates two valid strings of *L*;

A string with “+” but not “=” is in *L* iff every “+” in the string separates two valid strings of *L*.

“23+4=555” is in *L*; “=234=+” is not.

Example from the “Formal Languages” Wikipedia entry, which is surprisingly good.

There are many different kinds of formal languages!



https://en.wikipedia.org/wiki/Chomsky_hierarchy

Regular expressions describe regular languages...

... which are the category of languages that can be *recognized* using a finite state automaton.

"Recognized": decide whether a particular string is a member of a given language.

Finite State Automata

Consider an imaginary machine with a finite number of states...

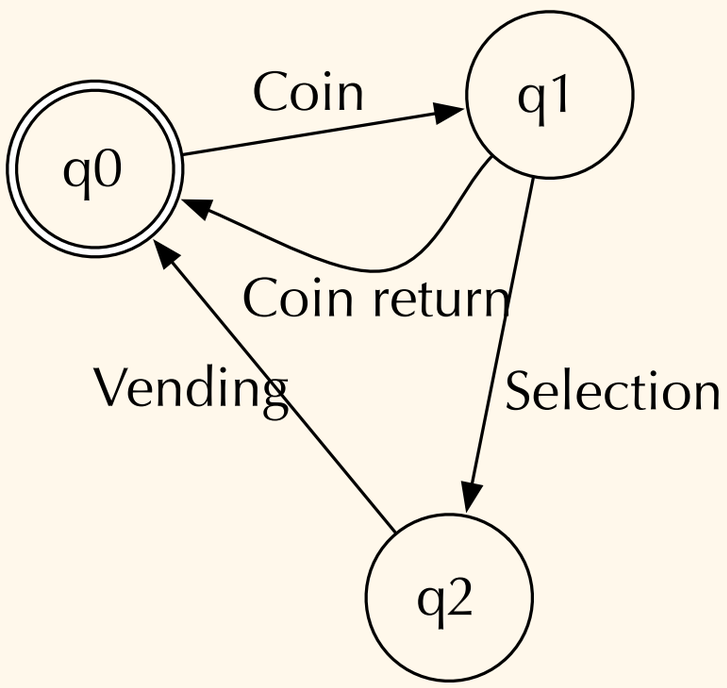
... as well as a set of *transitions* between the states...

... and designated *start* and *stop* states.

Example: a vending machine

Example: a vending machine

"To start, you put a coin in; you can hit 'coin return' to start over; after putting in a coin, you can enter a selection; after entering a selection, the machine gives you the item."



State	Coin	Coin R.	Select.	Vending
0	1	0	∅	∅
1	∅	0	2	∅
2	∅	∅	∅	0

Formal definition:

An FSA is defined by:

$Q = q_0, q_1, q_2 ... q_{n-1}$	A finite state of *n* states
Σ	A finite input alphabet of symbols
q_0	A start state
F	Set of final states $F \subseteq Q$
$\delta(q, i)$	Transition matrix between states

Conceptual operation:

There exists an “input tape” made up of a linear sequence of symbols...

Starting at q0, we read the first input symbol...

If it matches one of our transition arcs, we move to the destination state and advance the input tape...

If we hit a final state before we run out of input, we “succeed”; otherwise, we “fail.”

FSAs have lots of uses:

http://www.twistypuzzles.com/forum/viewtopic.php?f=15&t=25057

There exists a person, a goat, a cabbage, and a wolf.

They all have to cross a river...

... but their boat is only big enough for two of them at a time!

Can’t leave the goat and the wolf together...

... ditto the goat and the cabbage.

Example stolen from Picheny and Chen’s Columbia University E6884 slides.

Modeling this with an FSA:

Each state represents a combination of things and their location:

All on left side: MWGC-

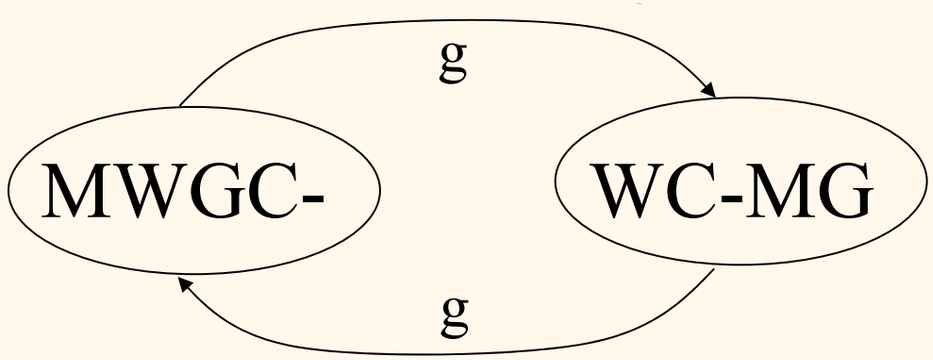
Person and goat on right side: WC-MG

All on right side: -MWGC

Example stolen from Picheny and Chen's Columbia University E6884 slides.

Modeling this with an FSA:

Transitions represent something happening:

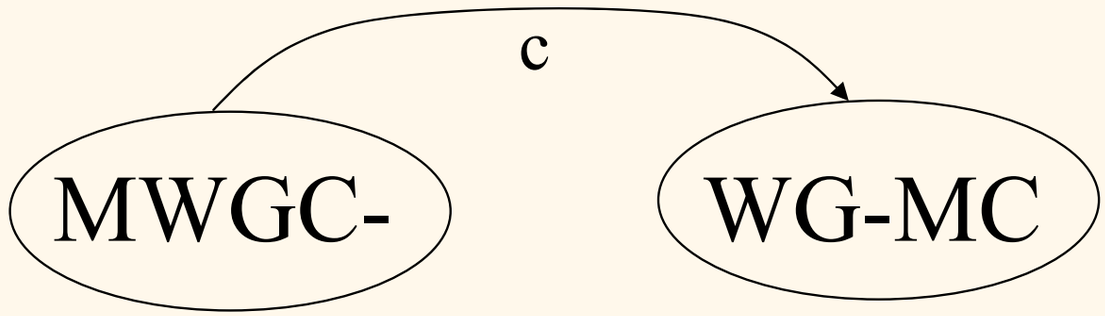


g	Person and goat cross
w	Person and wolf cross
c	Person and cabbage cross
m	Person crosses alone

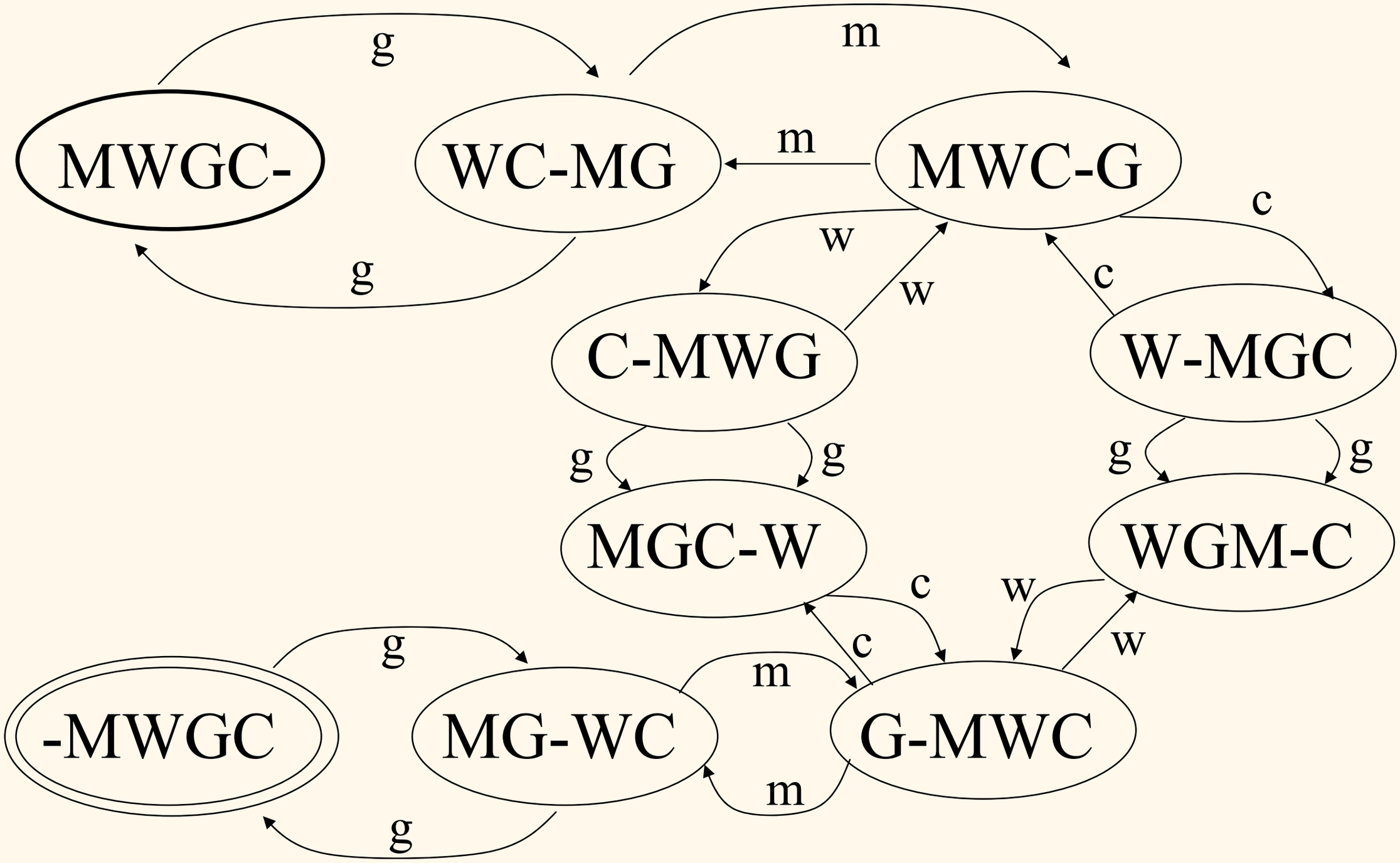
Example stolen from Picheny and Chen's Columbia University E6884 slides.

Modeling this with an FSA:

Some states are bad! Let's not bother modeling those:

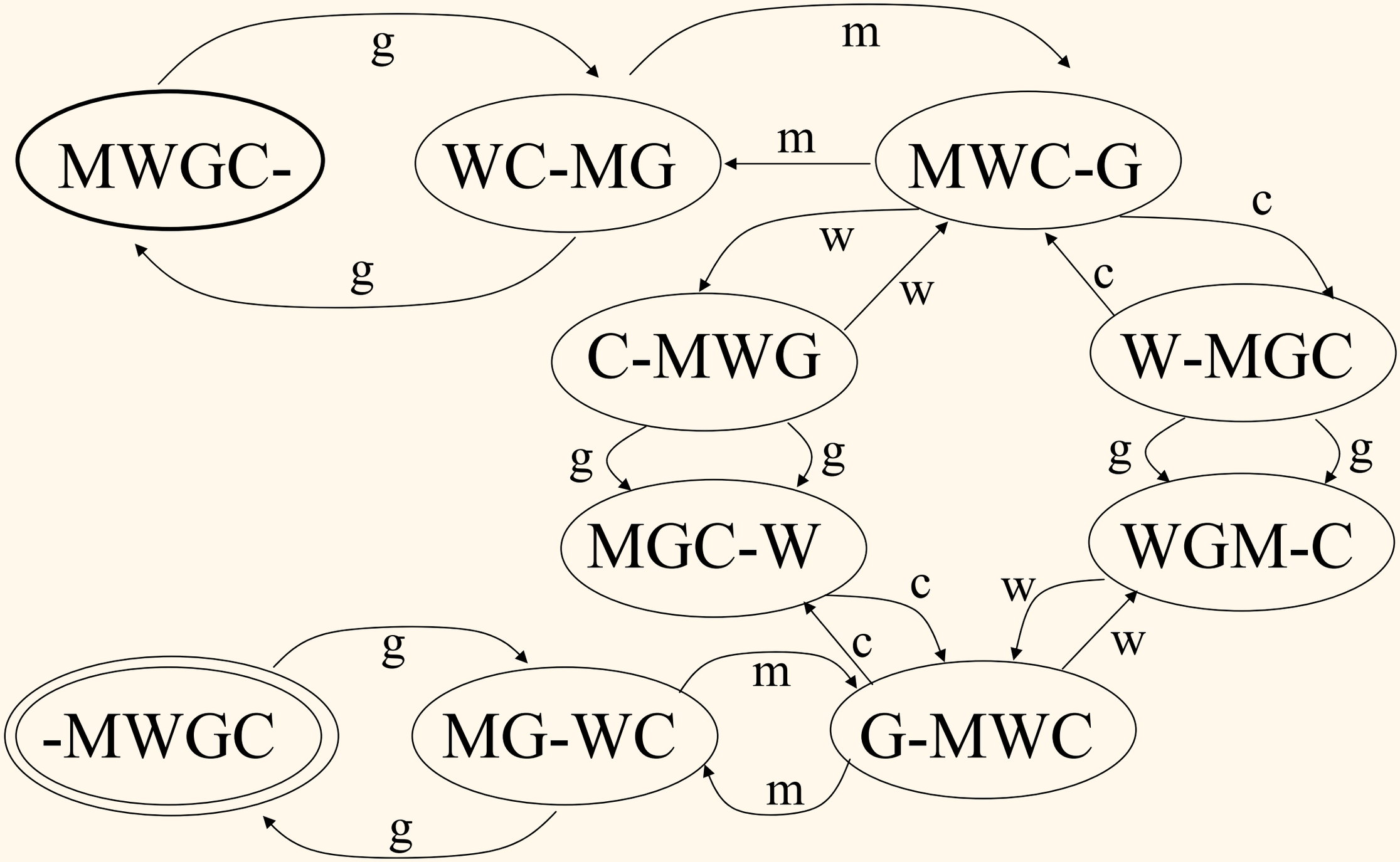


Example stolen from Picheny and Chen's Columbia University E6884 slides.



Now we have an FSA that can tell us whether a given sequence of "moves" will work!

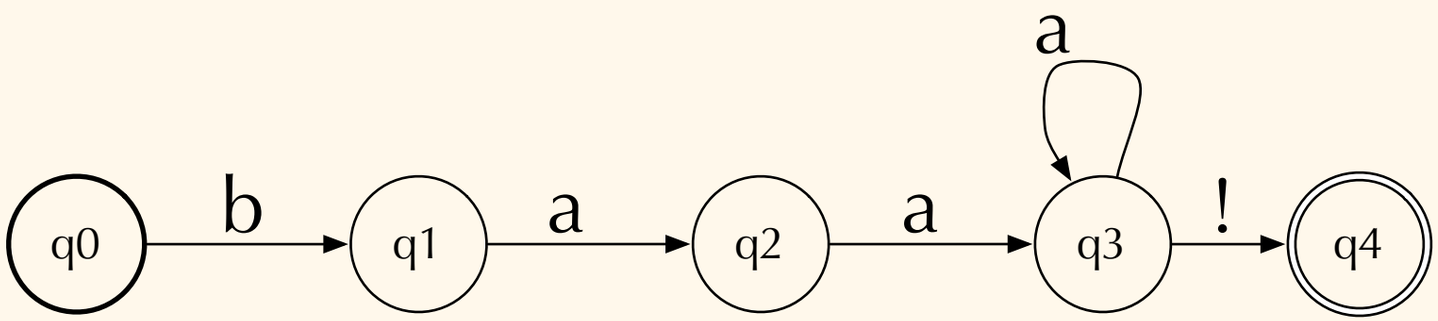
Example stolen from Picheny and Chen's Columbia University E6884 slides.



g m w g c m g

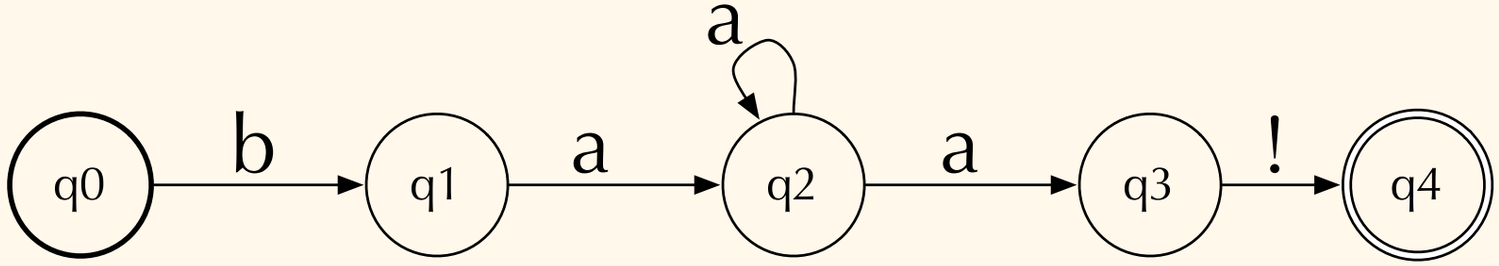
g m c g w c c

Example stolen from Picheny and Chen's Columbia University E6884 slides.



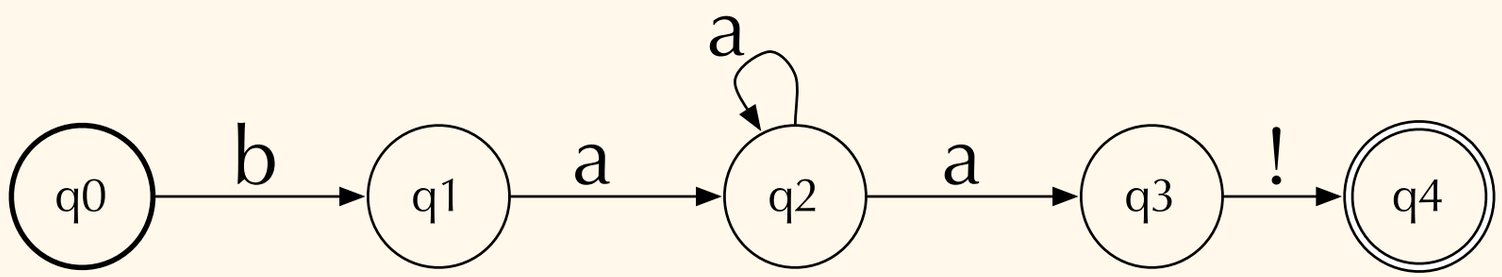
Deterministic

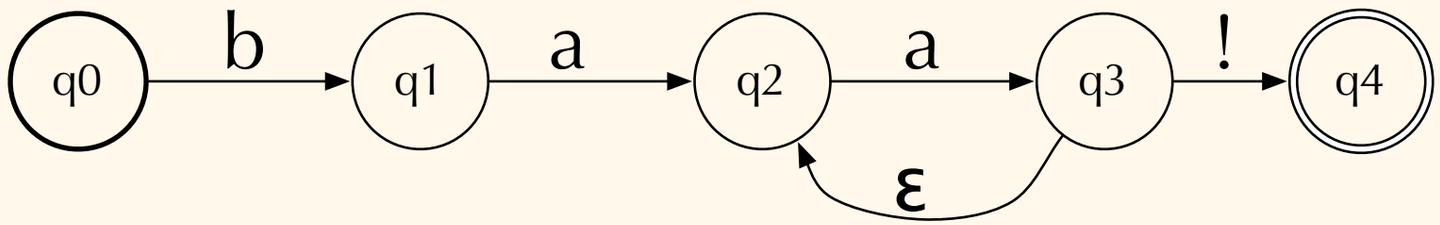
State	b	a	!
0	1	∅	∅
1	∅	2	∅
2	∅	3	∅
3	∅	3	4
4	∅	∅	∅



Non-deterministic

State	b	a	!
0	1	∅	∅
1	∅	2	∅
2	∅	2,3	∅
3	∅	∅	4
4	∅	∅	∅





Epsilon arcs change state without advancing the input tape.

State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	3	∅	∅
3	∅	3	4	2
4	∅	∅	∅	∅

All NFSAs have deterministic equivalents...

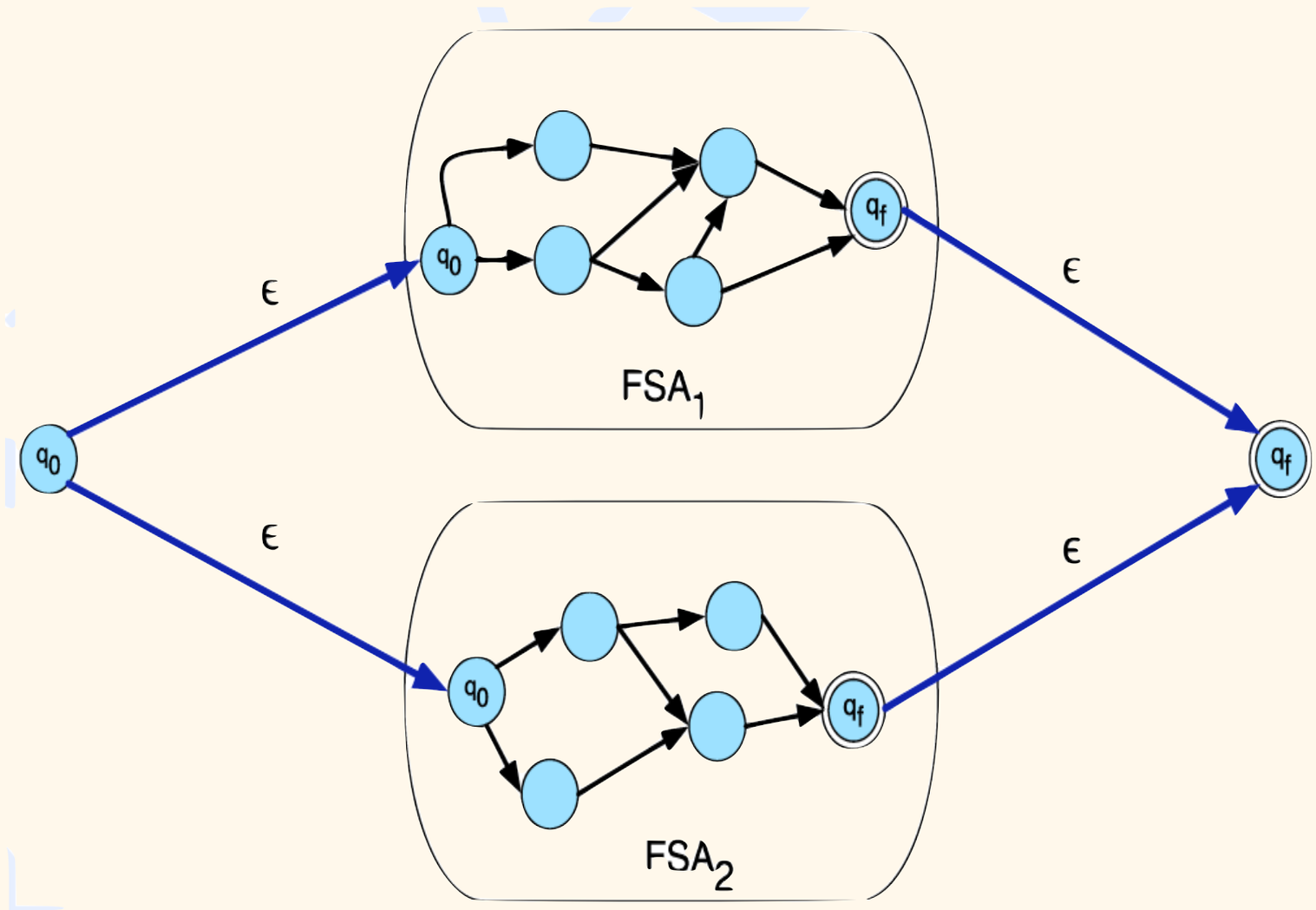
... but those equivalents may be much larger and/or more complex!

On the other hand, matching & searching using an NFSA can be slower and more involved.

(see J&M for gory details)

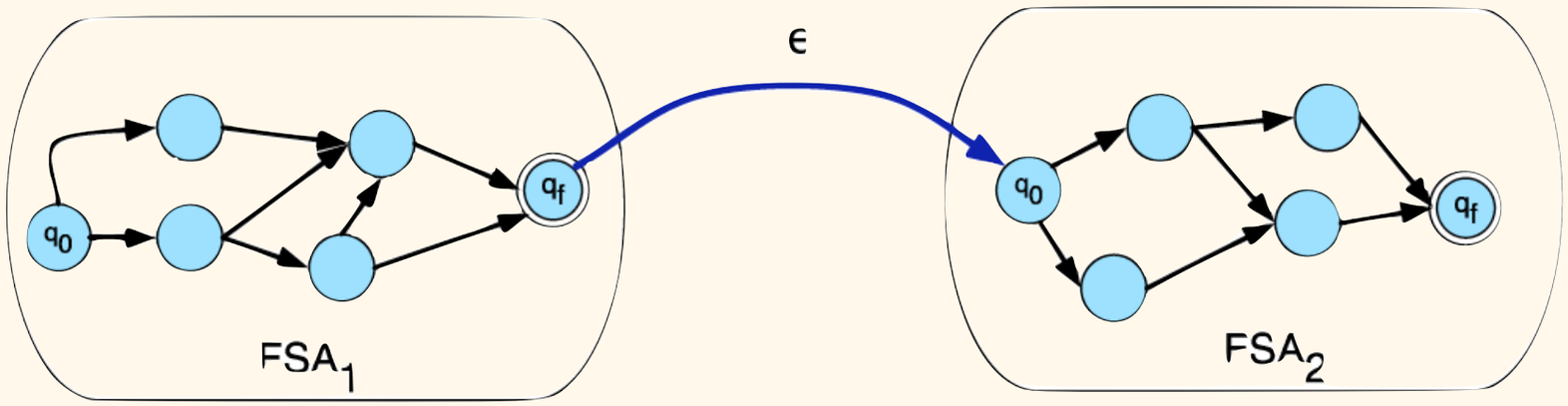
FSAs can help us understand standard formal language operations:

Union/Disjunction:

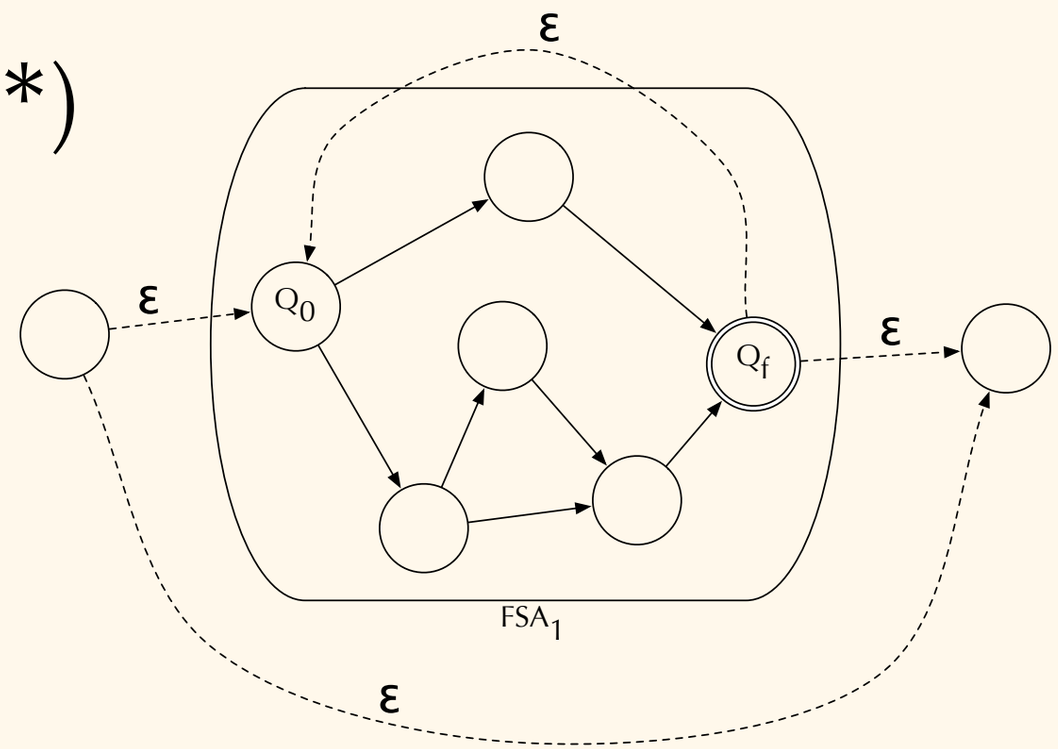


FSAs can help us understand standard formal language operations:

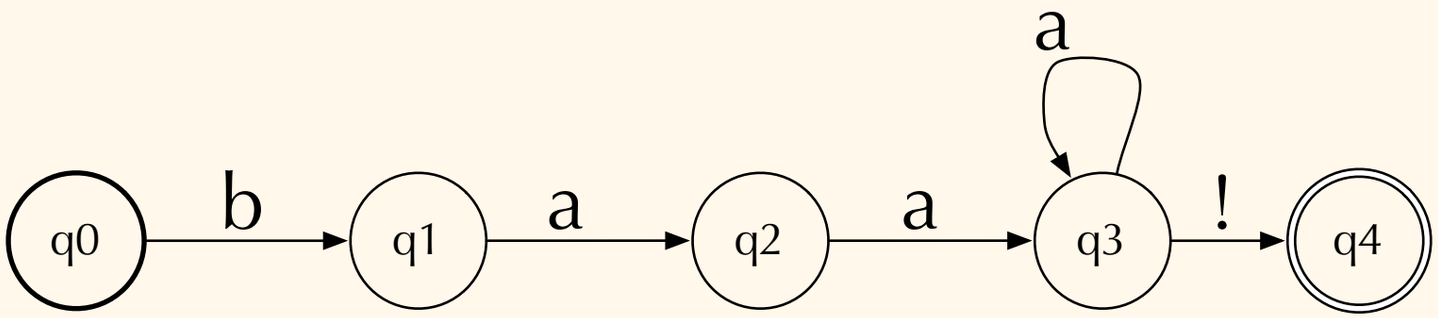
Concatenation



Closure (Kleene*)



Regular languages, by definition, can be recognized by an FSA.



baaa! baaaaaaaaa! baaaa ba!

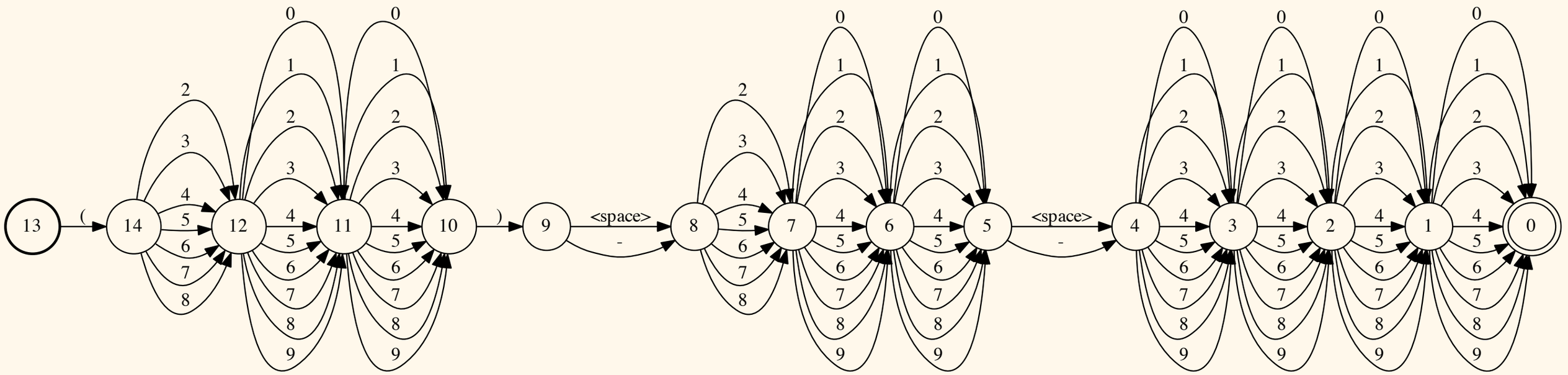
Why use this method to represent a language?

We can represent a (potentially) infinite set of strings in a closed form!

Also, thinking back to our earlier example...

Describing a language in prose is a lot of work...

... and it is hard to tell if we've done it clearly!



Thinking about it in terms of a state machine makes everything clearer.

Of course, we don't want to *always* be thinking in terms of FSAs...

... which is where regular *expressions* come in!

Conceptually: we define a language whose valid strings match the patterns that we want to find.

Such a pattern can be matched ("recognized") in linear time using an FSA!

Regular Expression basics:

`/monkeys/` Single occurrence of “monkeys”

`/[Mm]onkeys/` Same, but also match “Monkeys”
disjunctions

`/monkeys?/` “monkey”, “monkeys”
“zero or one”

`/m+onkeys/` “monkeys”, “mmonkeys”
“at least one”

`/m*onkeys/` “onkeys”, “monkeys”, “mmonkeys”
“one or more”

`/m{3,5}onkeys/` “mmmonkeys”, “mmmmonkeys”
“between three and five”

Additional pattern specifiers:

`/./`	Any character
`/\d/`	Any digit
`/\w/`	Any alphanumeric character
`/\W/`	Any non-alphanumeric character
`/\s/`	Any whitespace character
`/\S/`	Any non-whitespace character
`/\b/`	Word boundary

Note that all of these are just "shorthand," and could each be represented as regular expressions on their own!

Putting it all together:

`/\b\d{3}-\d{4}\b/` “867-5309”, “341-5588”

`/\d+(\.\d+)? GB|[Gg]igabytes?/`

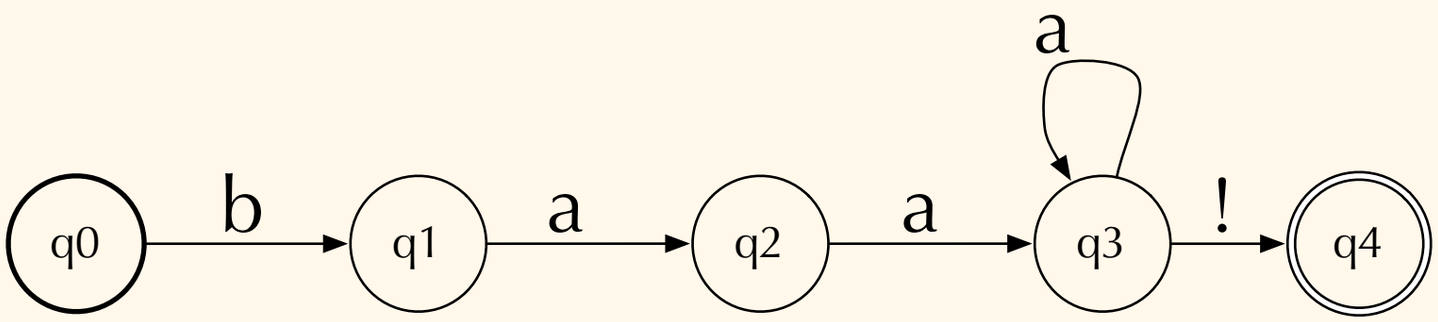
“1 GB”, “2.5 gigabytes”

```
/(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08\x0b
\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9]
(?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|
[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:
[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])/
```

“bedricks@ohsu.edu”, etc.

Regular languages, by definition, can be recognized by an FSA.

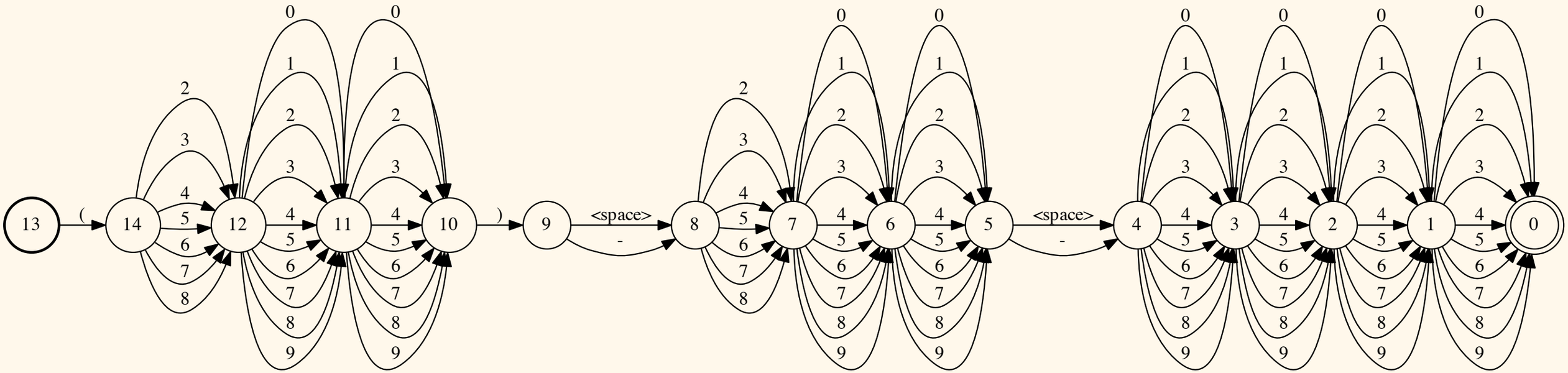
/baa+!/



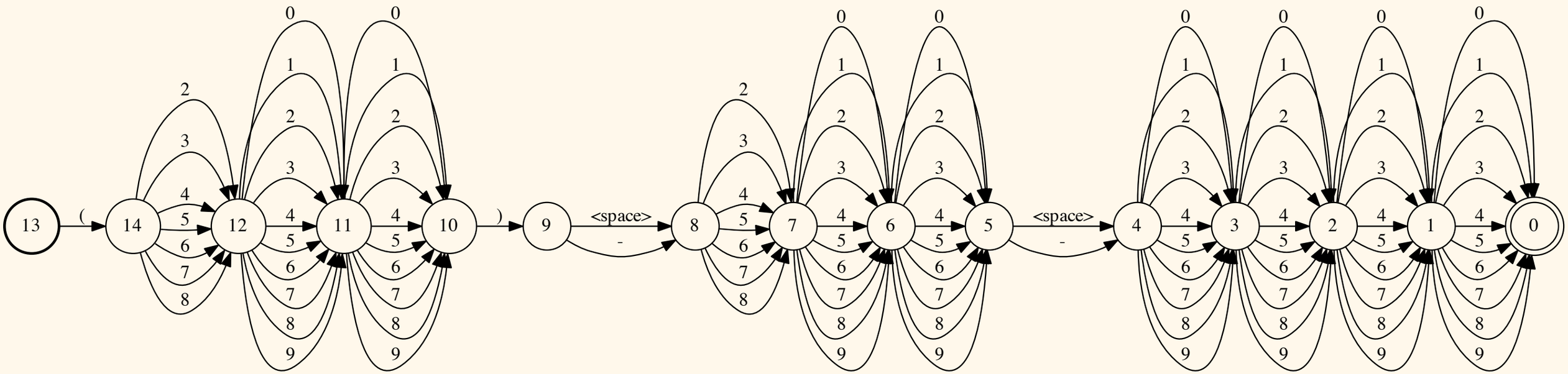
baaa! baaaaaaaa! baaaa ba!

Group activity: Let's draw an FSA!

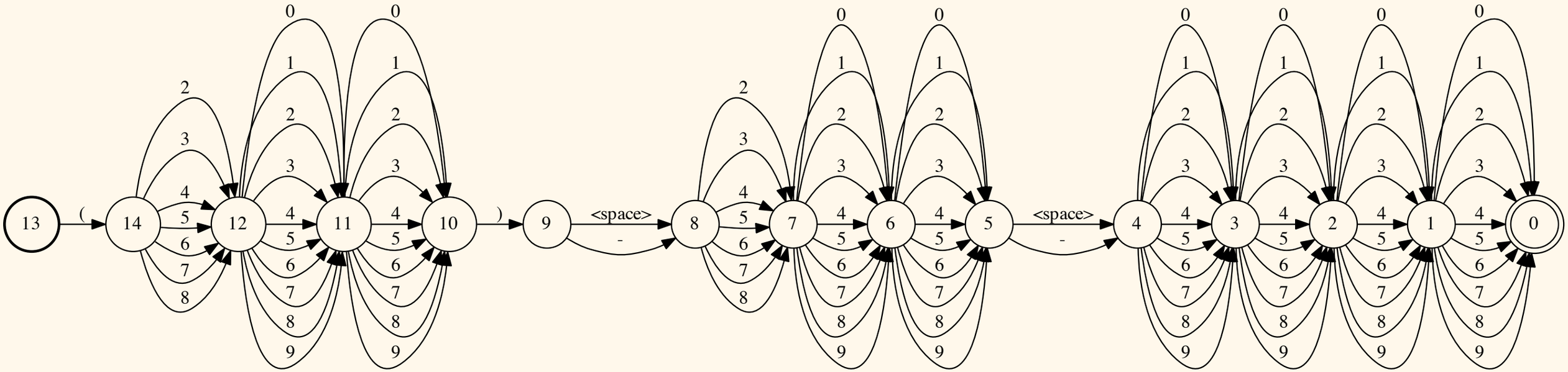
```
/a[ab]*b[cd]/
```



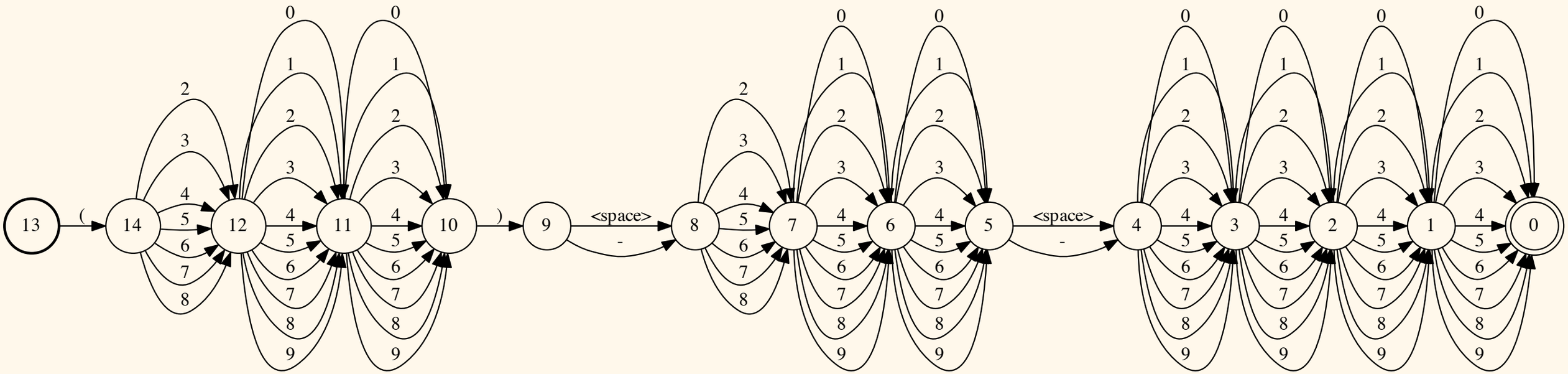

```
/\([2-8]\d\d\)[- ][2-8]\d\d[- ]\d{4}/
```



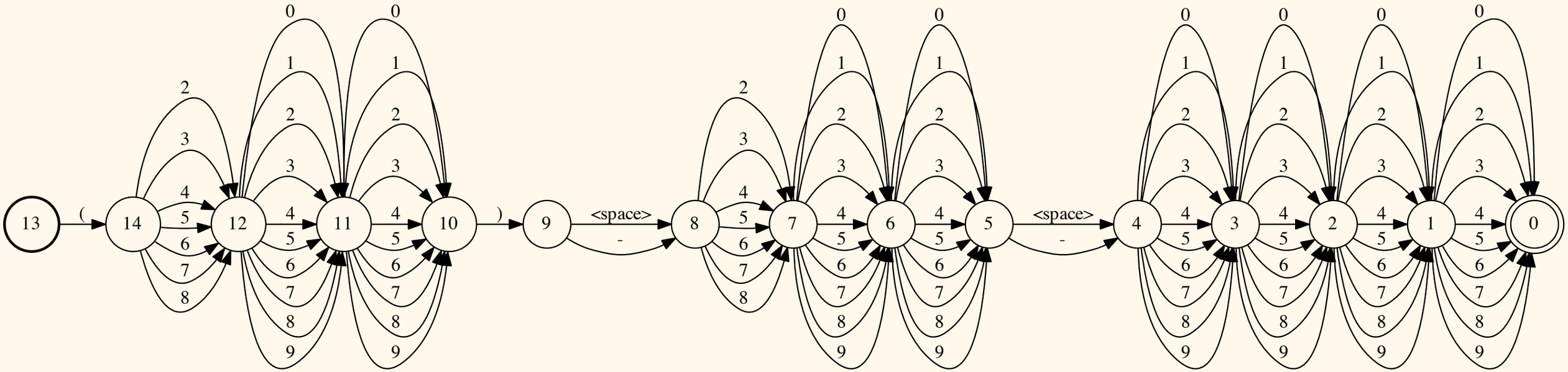

```
/\([2-8]\d\d\)[- ][2-8]\d\d[- ]\d{4}/
```



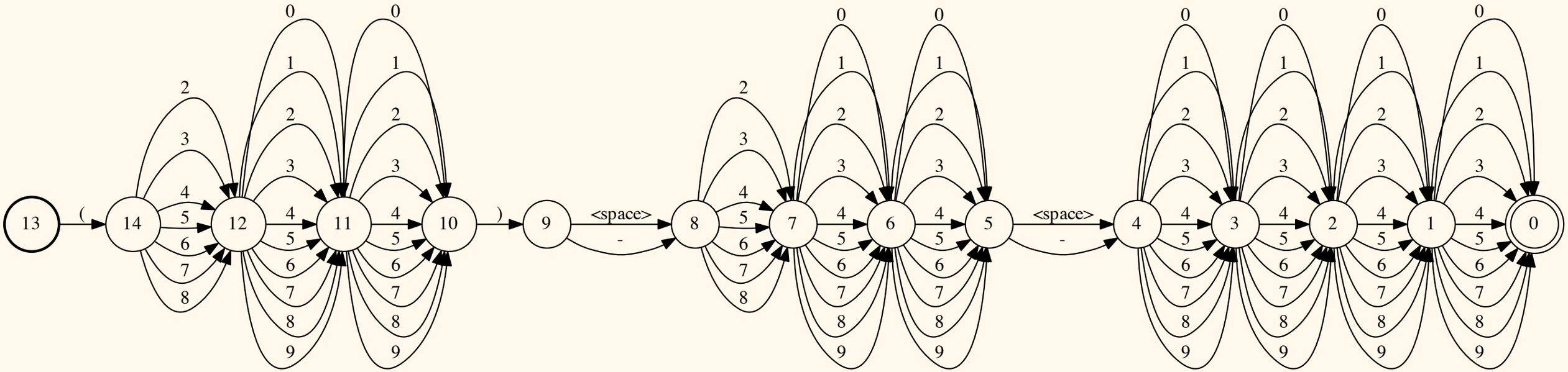

```
/\([2-8]\d\d\)[- ][2-8]\d\d[- ]\d{4}/
```



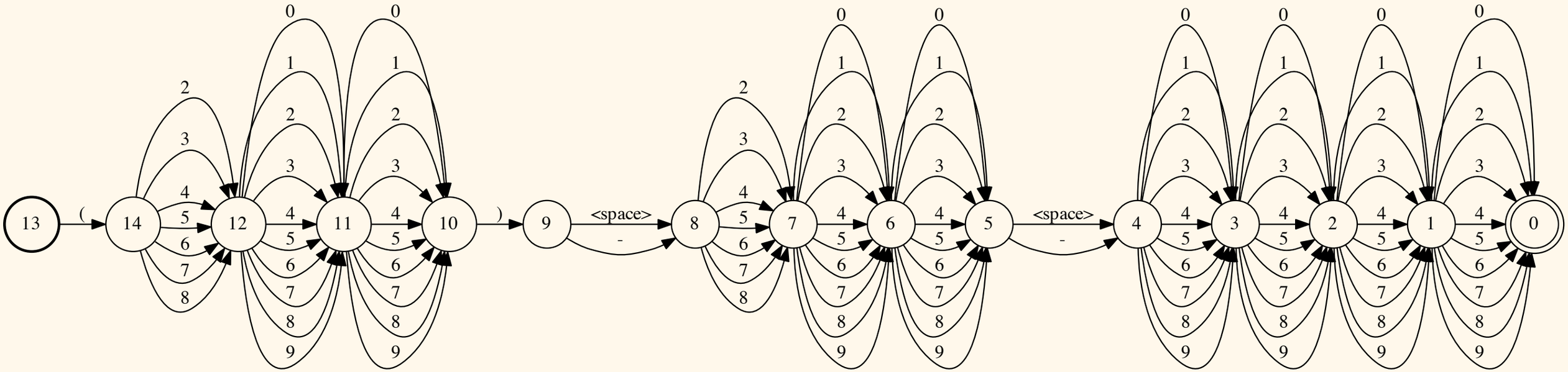

```
/\([2-8]\d\d\)[- ][2-8]\d\d[- ]\d{4}/
```



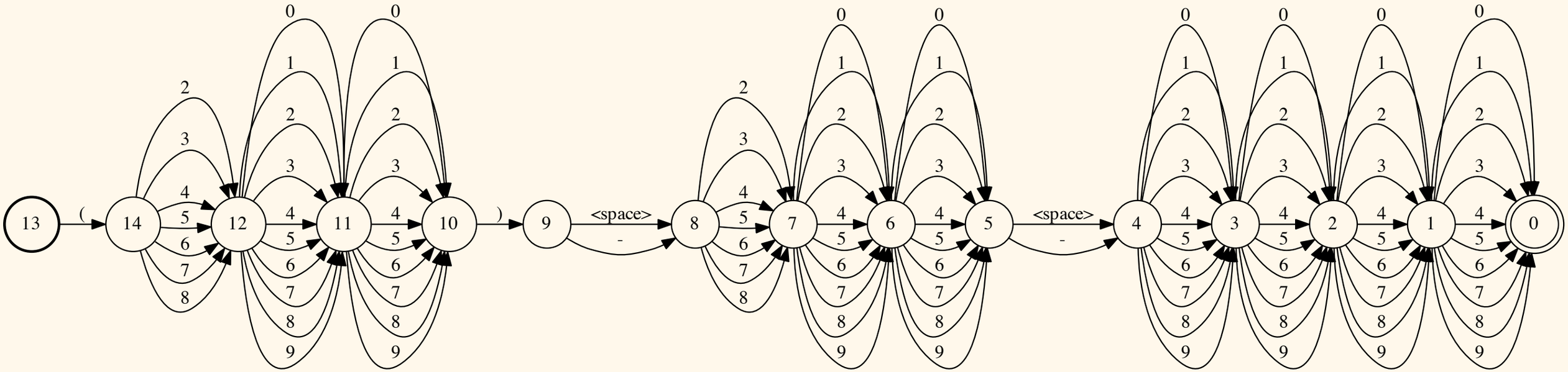

```
/\([2-8]\d\d\)[- ][2-8]\d\d[- ]\d{4}/
```



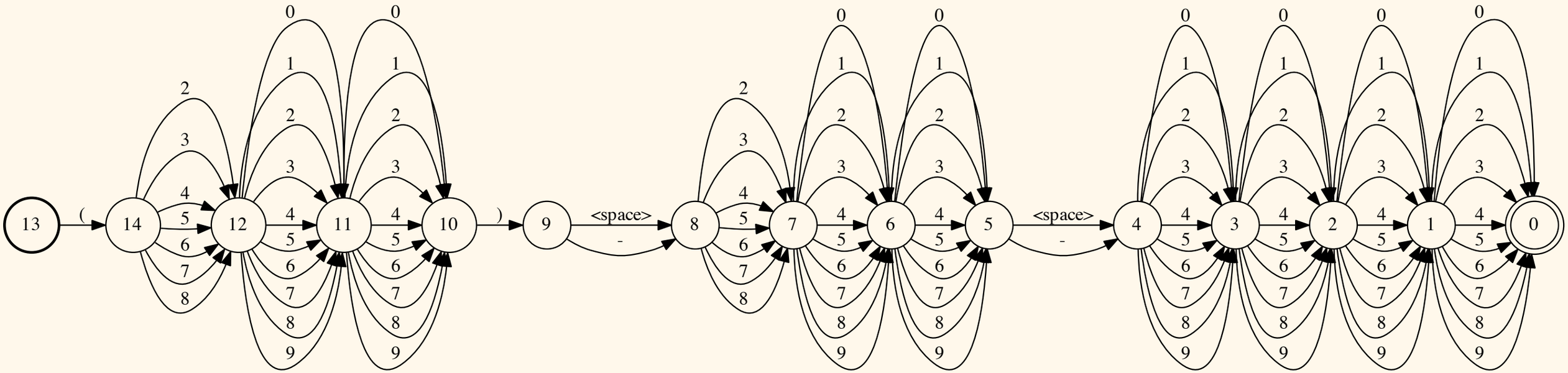

```
/\([2-8]\d\d\)[- ][2-8]\d\d[- ]\d{4}/
```




```
/\([2-8]\d\d\)[- ][2-8]\d\d[- ]\d{4}/
```




```
/\([2-8]\d\d\)[- ][2-8]\d\d[- ]\d{4}/
```




```
/\([2-8]\d\d\)[- ][2-8]\d\d[- ]\d{4}/
```

More tools:

```
/\d+(\.\d+)? GB|[Gg]igabytes?/
```

"1 GB", "2.5 gigabytes"

Grouping

```
/[^aeiou][1-9]/
```

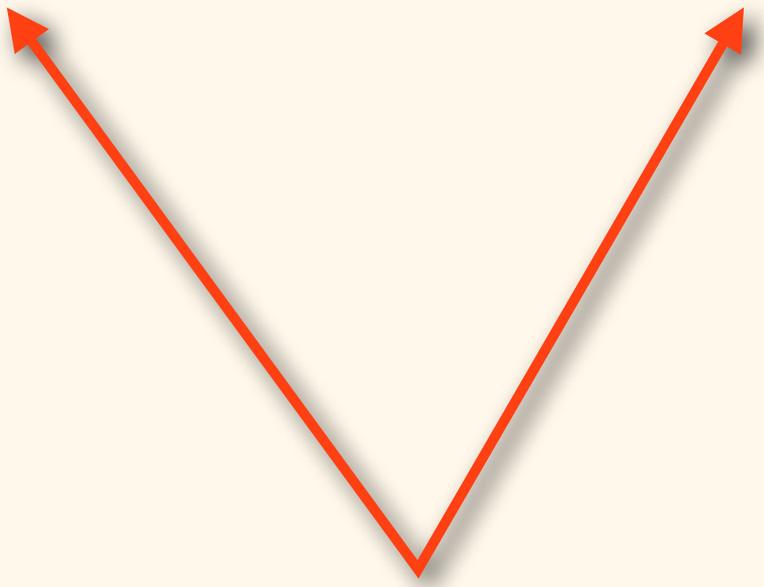
"v8", *not* "a8"

Negation

Ranges

More tools:

```
/^Chapter \d+$/
```



Anchoring

"Chapter 23";

not "See Chapter 23 for more details"

Note: you can anchor at both the start and end (as above), or at either one on their own.

Great for problems like "Find lines that start with..."

Some important things to know:

What most programming languages call "regular expressions" are often *not* "regular" in the linguistic sense.

Many useful features offered by modern regexp libraries are not possible in a regular language.

More tools:

```
/^Chapter (\d+)$/
```

"Chapter 23";

not "See Chapter 23 for more details"

"Capture Groups"

Many regexp engines allow you to refer to the contents of a capture group with a *backreference*.

More tools:

`/\b(\w+)\s+\1\b/`

the the

word word

This can be useful for matching repeated words or letters...

`/(\w)\1/` llama

lagging

`/(.)(.)\2\1/` abba

Capture groups and backreferences are useful for extracting pieces of our matches (to be discussed shortly).

Even more tools:

```
/grey(?=hound)/
```

greyhound, *not* greybeard

Lookahead assertion

Note: In this example, the “match” is “grey”- *not* greyhound!

```
/(?<=grey)hound/
```

greyhound, *not* bluehound

Lookbehind assertion

A note on lookarounds:

Lookahead and lookbehind assertions are very powerful...

... and, as such, should be used with caution.

There can be performance implications, and it also can make your regex harder to debug!

Some important things to know:

There is no single standard for regular expressions.

Different languages, platforms, etc. support different sets of features.

≈80-90% of the syntax is the same across platforms/languages, but not all!

Some important things to know:

Be mindful of Unicode issues!

Is “A” the same as “A”?

Remember normalization!

Does “\s” include *all* Unicode space characters?

Many regex engines support `/\p{property}/` expressions to match Unicode properties.

/\p{Greek}+/

Ειδήσεις

/\p{P}+/

,.;

/\p{White_space}+/

Some important things to know:

Regular expressions are great, but there are many situations where they are inappropriate:

Parsing HTML/XML

Validating user input in security-sensitive places

Validating email addresses/URLs, etc.

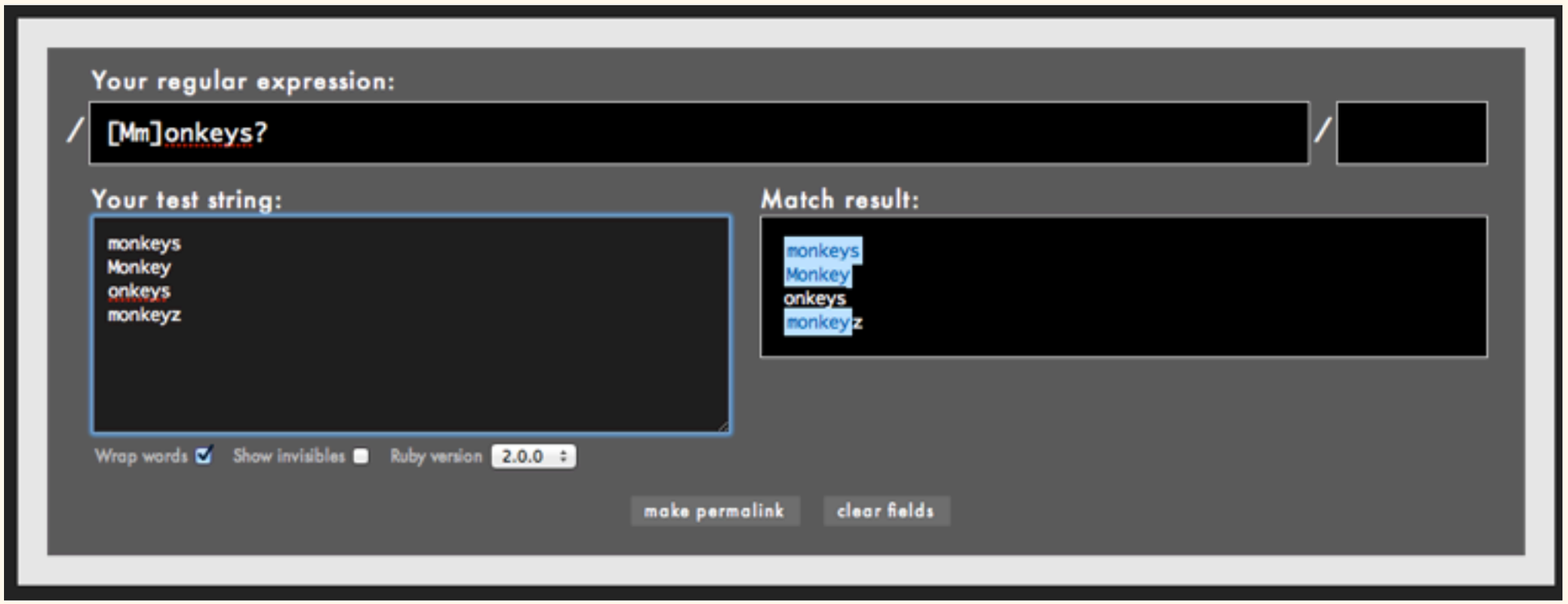
A regex can tell you if an address is well-formed, but not whether or not it will work!

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

Jamie Zawinski, 1997

Useful Regular Expression Tool Demo

http://rubular.com



In Python:

The standard Python library includes the “re” module:

```
import re
my_string = "Hello!"
m = re.search('^H[aeiou]', my_string)
m.start() # 0
m.end() # 2
re.search('f', my_string) == None # True
```

```
my_string = "Chapter 3"
m = re.search("Chapter (\d+)", my_string)
m.groups() # ('3',)


s = "There were many monkeys in the monkey section of the
monkey house at the zoo."
re.findall("monkeys?") # ["monkeys","monkey","monkey"]
```

A drop-in replacement (with more functionality) is linked to on the course website.

Homework assignment:

1. Converting FSA to Regex
2. Converting Regex to FSA
3. Counting sonnets
4. Counting surnames and titles
5. Regex golf