CISS 445 Programming Languages

31. Programming Language Syntax: Lexing

Dr. Liow

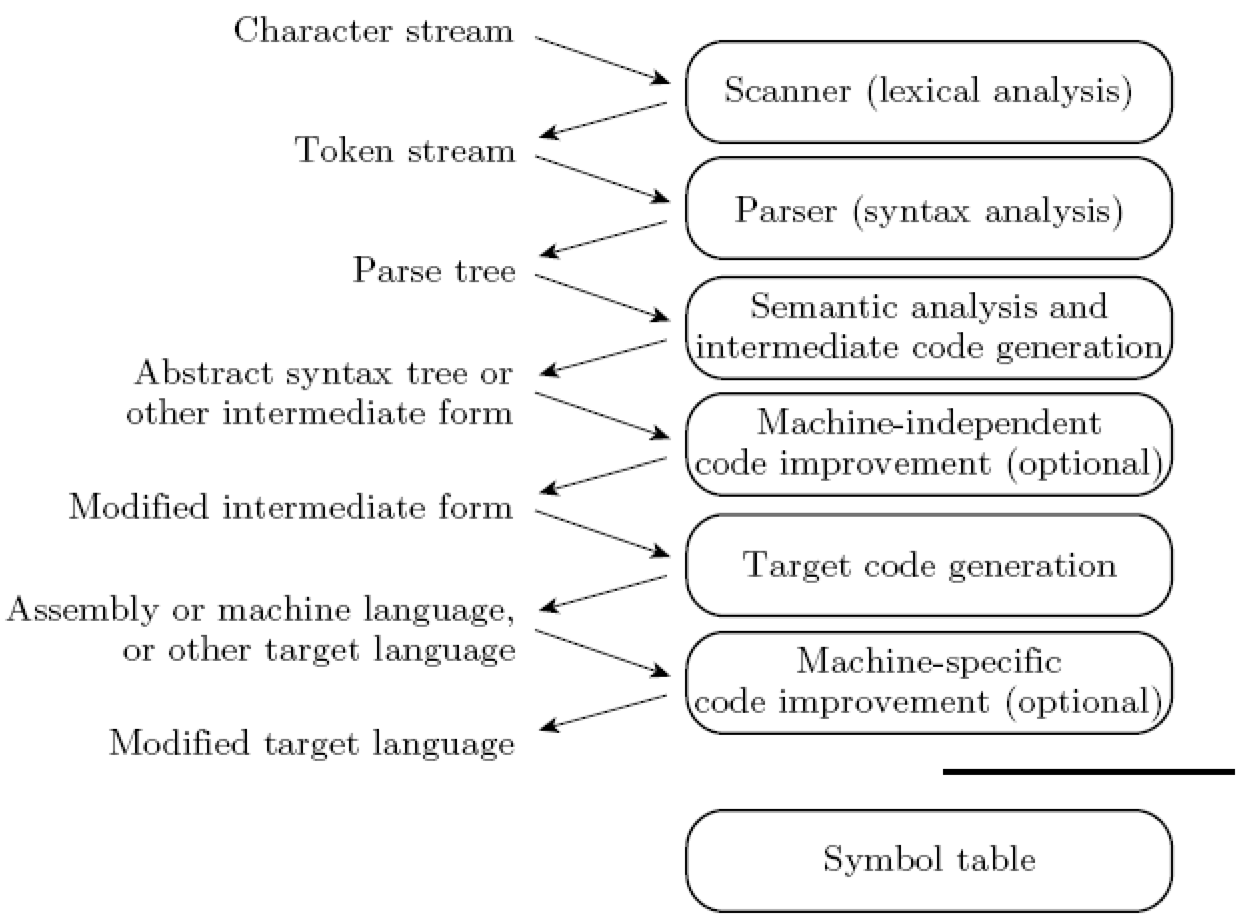
Roadmap

- Talked briefly about overall structure of compilers.
- Functional programming, OCAML
- Types
- Next: We will now look at the syntax of programming languages.

Roadmap

- Intro to FP
  - Ocaml, recursion, higher order function
- Foundations
  - Types
  - Lambda calculus
- Language syntax and parsing
  - DFA, grammars

Syntax


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Assembly or machine language, or other target language
Machine-specific code improvement (optional)
Modified target language
Symbol table

Syntax

- Syntax = description of strings which are valid and the organization of the “words” so that you get a valid program for a particular programming language
- Semantics = actual meaning of the program (what is it supposed to compute)

Elements of Syntax

- Character set (ASCII, Unicode)
- Keywords
- Special constants (cannot be assigned)
- Identifiers (can be assigned)
- Operator symbols
- Whitespaces

Elements of Syntax

- Expressions
- Type expressions
- Declarations
- Statements
- Subprograms

Elements of Syntax

- Modules
- Interfaces
- Classes

Formal Languages

- To describe syntax we use
  - Regular languages, regular expressions (regex), finite state automata (deterministic and non-deterministic)
  - Context-free languages, context-free grammars (CFG), BNF
- Theory of languages is not just a theoretical foundation for CS. It's has many practical uses. For instance regex is heavily used in web applications, AI, pattern recognition, etc and state diagram is used in software design.
- There are other classes of languages more powerful than the above. Details in class on automata theory.

Formal Languages

- First fix a set of symbols.
- From the symbols you can create strings.
- A language over the set of symbols is just a set of strings.
- Example:
  - Symbols: a, b. E.g. of lang = {}
  - Symbols: a, b. E.g. of lang = {a, ab, aab, aaab}
  - Symbols: x, y. E.g. of lang = {x, xy, xxy, xxxy, …}
  - Symbols: 0, 1. E.g. of lang = {00, 010, 0110, …}

Formal Languages

- Frequently we can’t write down everything in a language because it's infinite.
  - Example: Symbol 0, 1.
    - {binary numbers} is infinite
    - {string of 0s 1s ending with two 1s} is infinite
- Therefore we want a compact description of all the strings in a language.
- Note that the empty string is written ε. (Some books use λ.)

Regular Languages

- A **regular language** over a set of symbols S is a language over S satisfying some properties. (Not every set of strings over S is a regular language).
- Regular languages can be described compactly using regular expressions (regex), deterministic finite state machines, or non-deterministic finite state machines.
- (Regex is frequently used by programmers.)

Regular Expressions

- First talk about regex. You can think of a regex as a string itself. Regular expressions are important for breaking a string into words.
- Fix a set of symbols S.
  - Ø and ε are regex.
  - Every symbol in S is a regex.
  - If r is a regex, then (r) is also a regex
  - If r, s are regex, then rs is also a regex.
  - If r, s are regex, then r U s is also a regex.
  - If r is a regex, then r* is also a regex.

Regular Expressions

- Example: S = {a, b, c}. Ø is a regex.
- Example: S = {a, b, c}. ε is a regex.
- Example: S = {a, b, c}. b is a regex.
- Example: S = {a, b, c}. bc is a regex.
- Example: S = {a, b, c}. a U bc is a regex.
- Example: S = {a, b, c}. (a U bc) a* is a regex.
- Example: S = {a, b, c}. a*bc U a* is a regex.

Regular Expressions

- A regex is used to describe a language, i.e., a set of strings.

Symbols	Regex	Language
{a,b,c}	Ø	{ } (i.e., nothing)
{a,b,c}	ε	{ε} (empty string)
{a,b,c}	a	{a}
{a,b,c}	ab	{ab}
{a,b,c}	a U b	{a, b}

Regular Languages

Symbols	Regex	Language
{a,b,c}	a U b	{a, b}
{a,b,c}	(a)	{a}
{a,b,c}	a*	{ε,a,aa,aaa,aaaa,…}
{a,b,c}	a U bc	{a, bc}
{a,b,c}	(aUbc)a*	{a, bc} {ε,a,aa,…} = {a, bc, aa, bca, aaa, bcaa, …}

Regular Languages

Symbols	Regex	Language
{a,b,c}	a*bcUa*	{ε,a,aa,…}bc U {ε,a,aa,…} = {bc, abc, aabc, …} U {ε,a,aa,…} = {bc, abc, aabc, …, ε,a,aa,…}
{0,1}	(01U10)*	?

Regular Languages

- Example.
  - S = {0,1,2,3,4,5,6,7,8,9}
  - digit = 0 U 1 U 2 U 3 U 4 U 5 U 6 U 7 U 8 U 9
  - digit is the regex whose language is S.
- Example.
  - S = {0,1,2,3,4,5,6,7,8,9}
  - nonzerodigit = 1 U 2 U 3 U 4 U 5 U 6 U 7 U 8 U 9
  - nonzerodigit is the regex whose language is {1, 2, 3, 4, 5, 6, 7, 8, 9}

Regular Languages

- Example.
  - S = {0,1,2,3,4,5,6,7,8,9}
  - posint = nonzerodigit digit*
  - The language described by posint is

= {1, 2, ..., 9}{0, 1, 2, …, 9}*

= set of positive integers (first digit is nonzero)

Regular Languages

- Example.
  - S = { _, a, b, c, …, z, A, B, …, Z, 0, 1, …, 9}
  - identifier = (_ U a U … U z U A U … U Z)
    (_ U a U … U z U A U … U Z U 0 U 1 U … U 9)*
  - The language generated by identifier is the set of valid C/C++ identifier names.

Exercise

- Exercise. What is the regex describing floating point literal in C/C++, Java, etc? Write down some examples to guide you.
- Exercise. What is the regex for phone numbers? (Examples: 573-123-1234, (573)123-1234, 5741231234, 573-1231234.)
- Exercise. What is the regex for phone numbers with 1-4 digit extensions (1-4 digits)? (Examples: 573-123-1234ext42)

Exercise

- Exercise. What is the regex for email address? (Example: yliow@ccis.edu)
- Exercise. What is the regex for integer expression? Binary operations are +,-,*,/,%,^.

Deterministic Finite State Automata

- So a regex describes sets of strings.
- Great … I still need to write boolean functions that tells me if a string is allowed by the regex or not.
- Example: Suppose I have the regex for phone numbers. I still have to write a function valid_ph that tells me if a string is valid phone number or not.
- What now? ...

Deterministic Finite State Automata

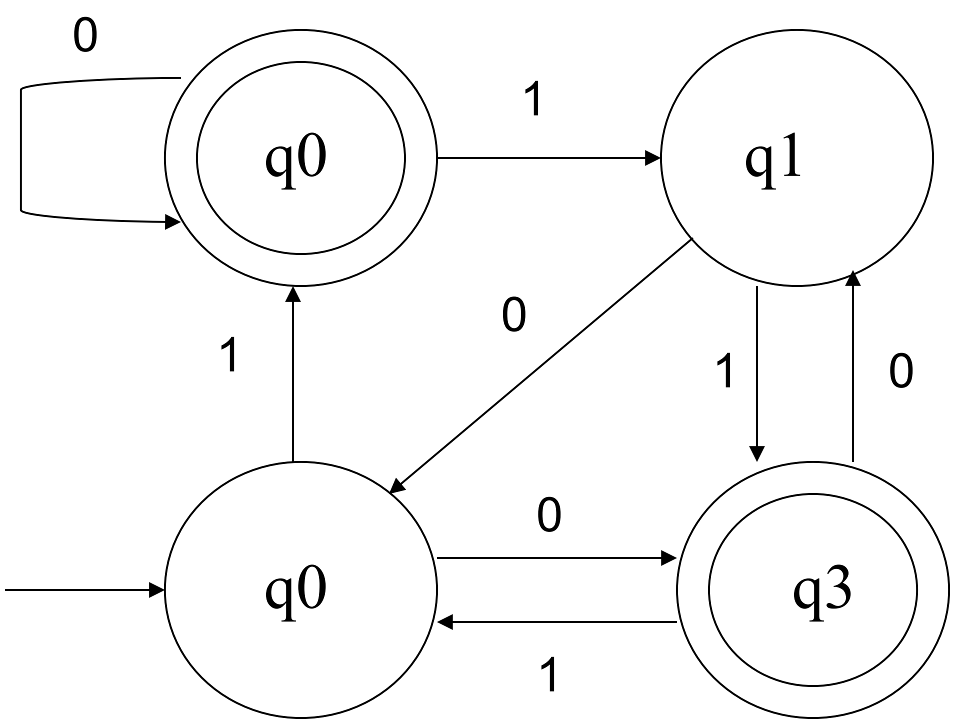
- A **deterministic finite state automata** or **machine** (**DFA** or **DFSM**) over a set S of symbols is a directed graph.
  - Nodes are **states** and edges are **transitions**.
  - There is one state called the **start state**.
  - Some states (including none) are designated as **final states** or **accept states**.
  - For each state q and each symbol s in S, there is exactly one directed edge out of q labeled s.
  - An edge can start and end with the same state.
  - The start state can also be a final state.

Deterministic Finite State Automata

- Conventions:
  - The start state is indicated by an arrow (or a double arrow) pointing to it from nowhere
  - The final/accept states are draws with double boundaries
- A string is **accepted** by the DFA if on traversing the graph starting at the start state and moving from one state to another based on the symbol on the string (read left-to-right), the last symbol of the string land you in a accept/final state.
- The **language accepted** or **generated** by the DFA is the set of strings accept by the machine.
- If M is a DFA, the language of M is written L(M).

Deterministic Finite State Automata

- S = {0,1}



Where is the start state?
Where are the final states?
Where do you end for the string 01?

$q0 \rightarrow q0 \rightarrow q1$

What about 001? 010? 0000?
Can you describe the Language of this DFA?

Exercises

- Design a DFA over the symbols {a, b} that accepts exactly the language of {}.
- Design a DFA over the symbols {a, b} that accepts exactly the language of {ε}.
- Design a DFA over the symbols {a, b} that accepts exactly the language of {a}.
- Design a DFA over the symbols {a, b} that accepts exactly the language of {a, b}.

Exercises

- Design a DFA over the symbols {a, b} that accepts exactly the language of the regular expression a*.
- Design a DFA over the symbols {a, b} that accepts exactly the language of the regular expression aa*.
- Design a DFA over the symbols {a, b} that accepts exactly the language of the regular expression (aa)*.

Exercises

- Design a DFA over the symbols {a, b} that accepts exactly the language of the regular expression (aba)*.
- Design a DFA over the symbols {a, b} that accepts exactly the language of the regular expression (a U bb)*.

Theorem

- It turns out that the following is true (although not obvious):
- (a) If L is the language of a regex, then you can construct a DFA M such that the language accepted by M is L.
- (b) If L is the language accepted by a DFA M, then there is a regex r such that the language of r is L.
- We say that regex and DFA are “equal in power”.

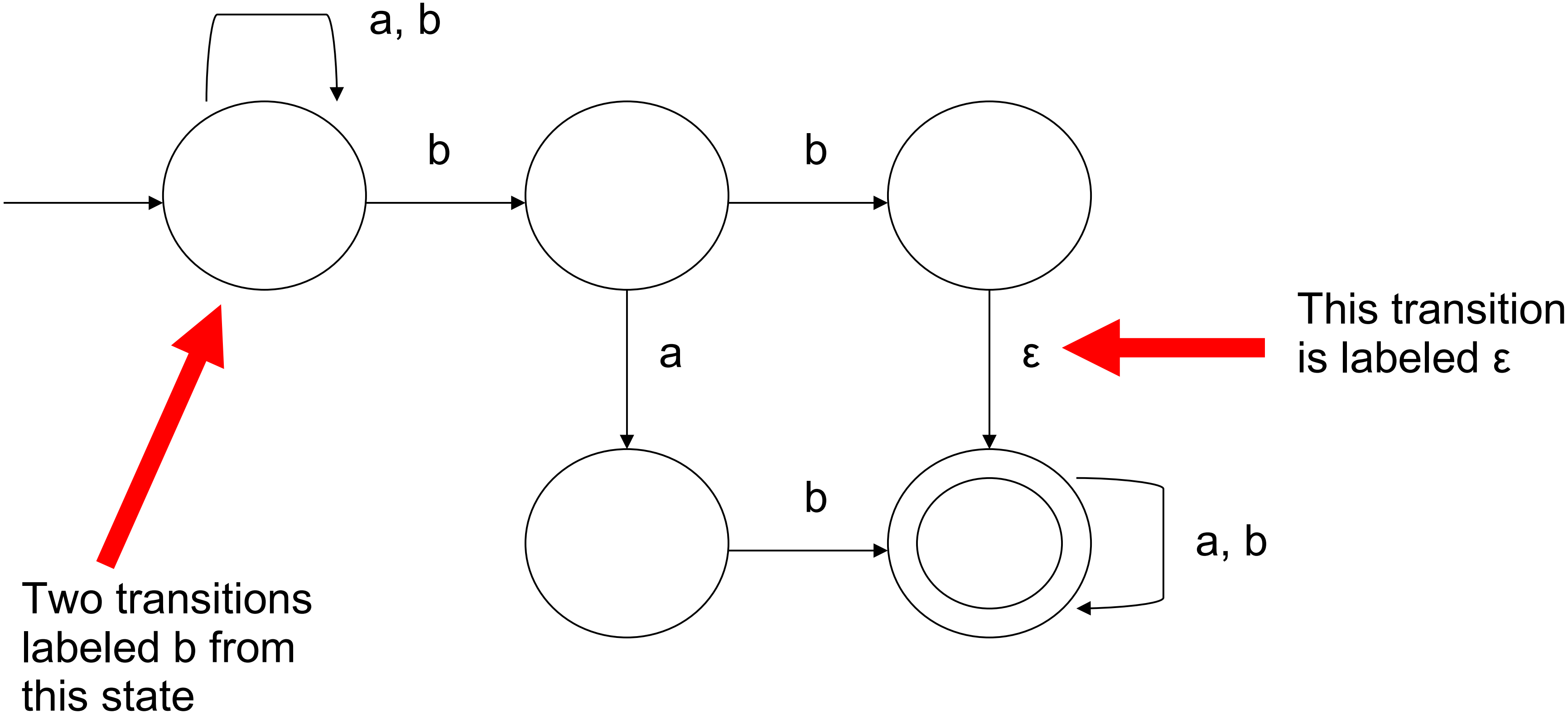
Exercises

- Construct if possible a DFA accepting each of the languages below,
  - The language of words with symbols 0,1 such that each word begins with 0 and ends with 1.
  - The language of words with symbols 0,1 such that each word contains an even number of 0s.
  - The language of words with symbols 0,1 such that each word does not contain 0101.
- In each case, if it's impossible, explain why it cannot be done!

Nondeterministic Finite State Automata

- A **nondeterministic finite state machine/automata** (**NFA**) is just like a DFA except that
  - Each state can have any number of transitions.
  - Labels on the transition includes ε.
  - A label can occur multiple times for a state.

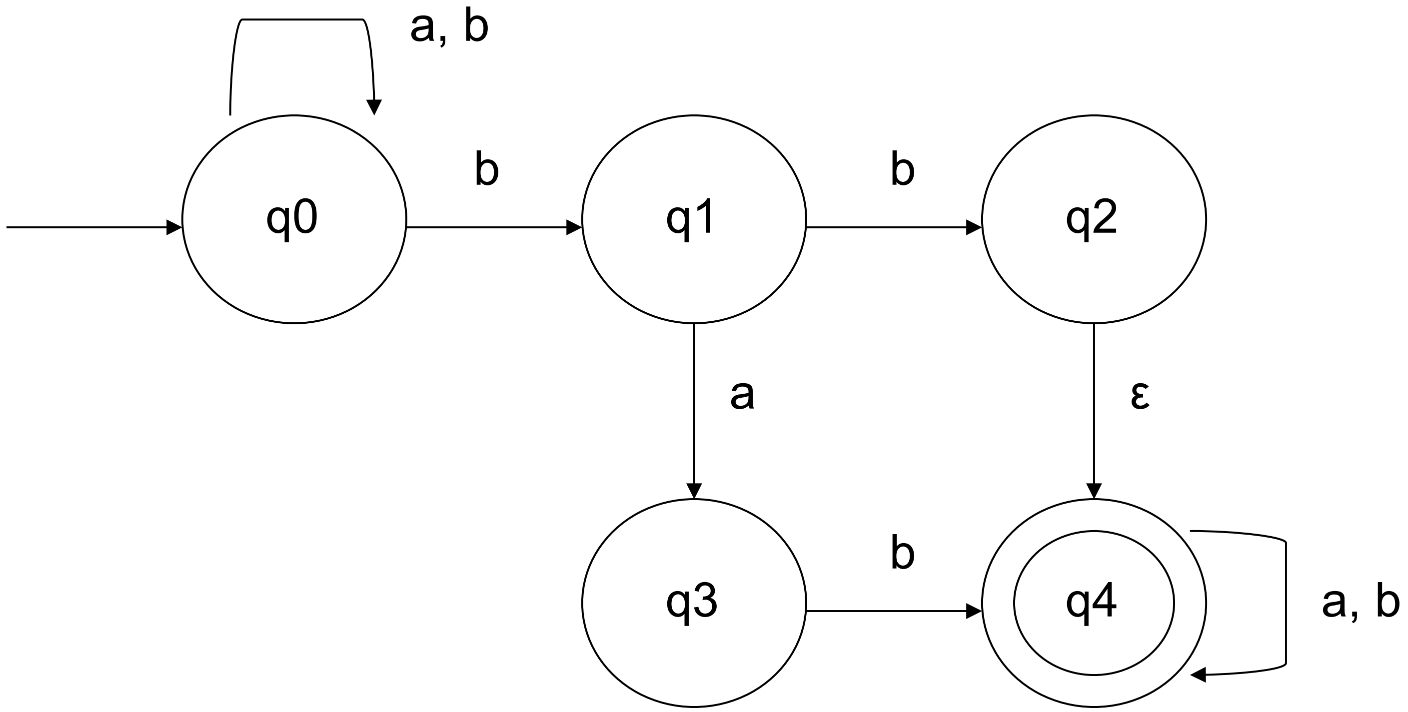
Nondeterministic Finite State Automata



Nondeterministic Finite State Automata

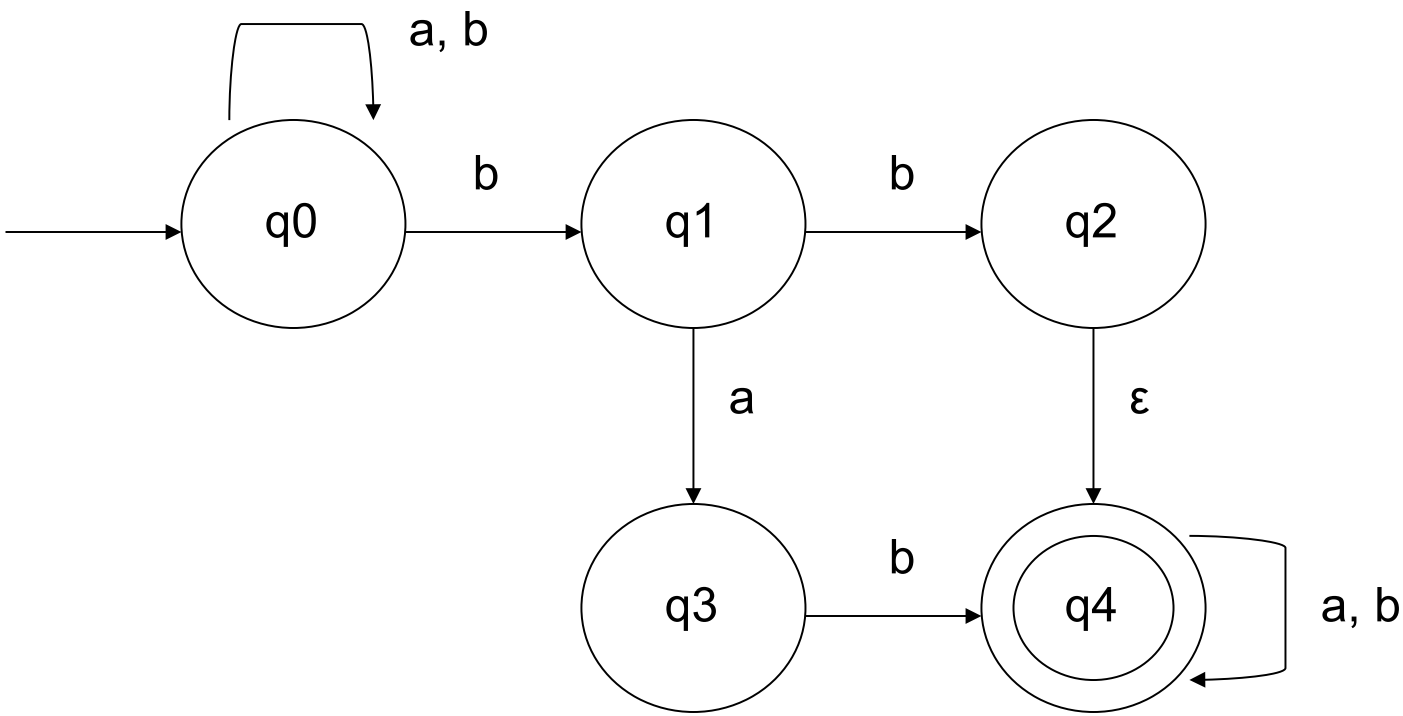
- A string is accepted by the NFA if you do the same thing as in DFA except for the following:
  - If there are two transitions labeled with the same symbol, then you need to try both.
  - If there is a transition labeled ε, you can choose either to stay or move to the next state without consuming a character
  - So there are (possibly) many traversals.
  - A string is accepted if ***any*** traversal ends in a final state.
  - If you land in a state where you cannot make a transition, you stop and try another new traversal.

Nondeterministic Finite State Automata



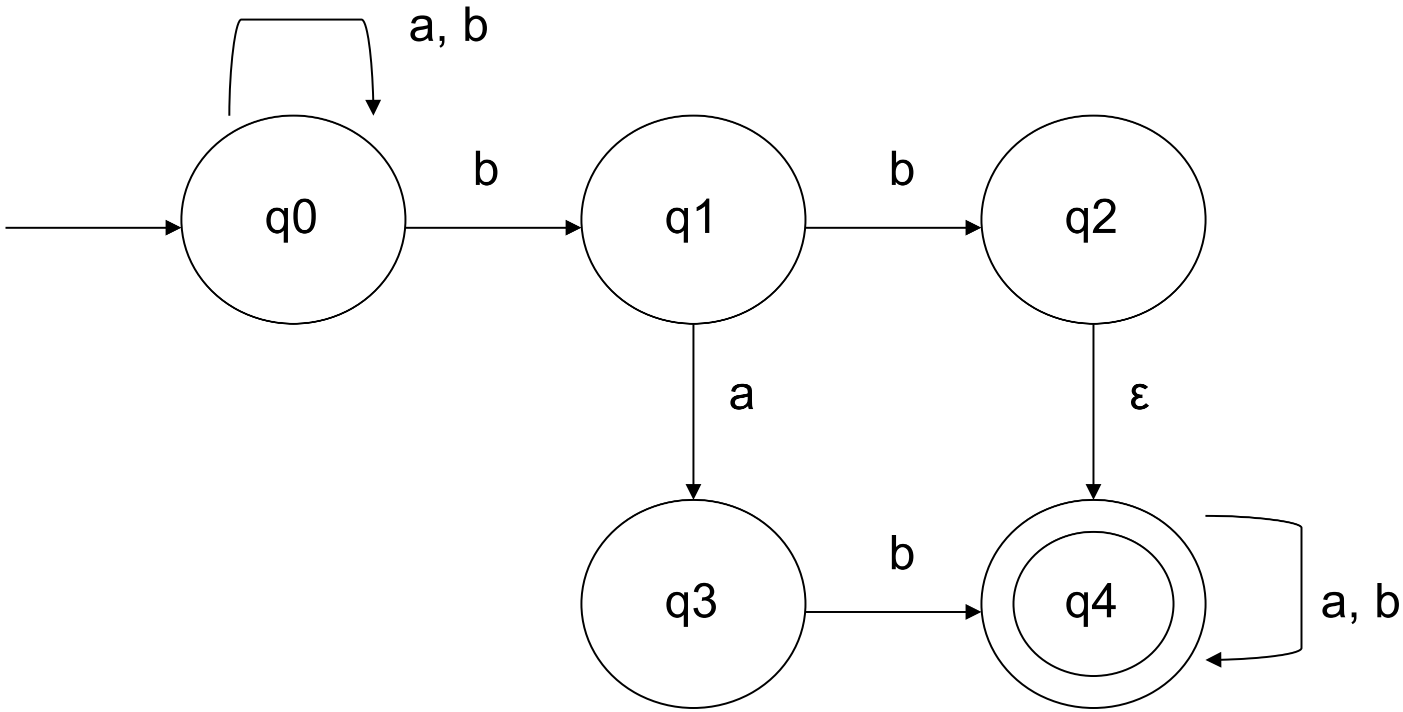
- Is ab accepted by this machine?
- One possible traversal: q0, q0

Nondeterministic Finite State Automata



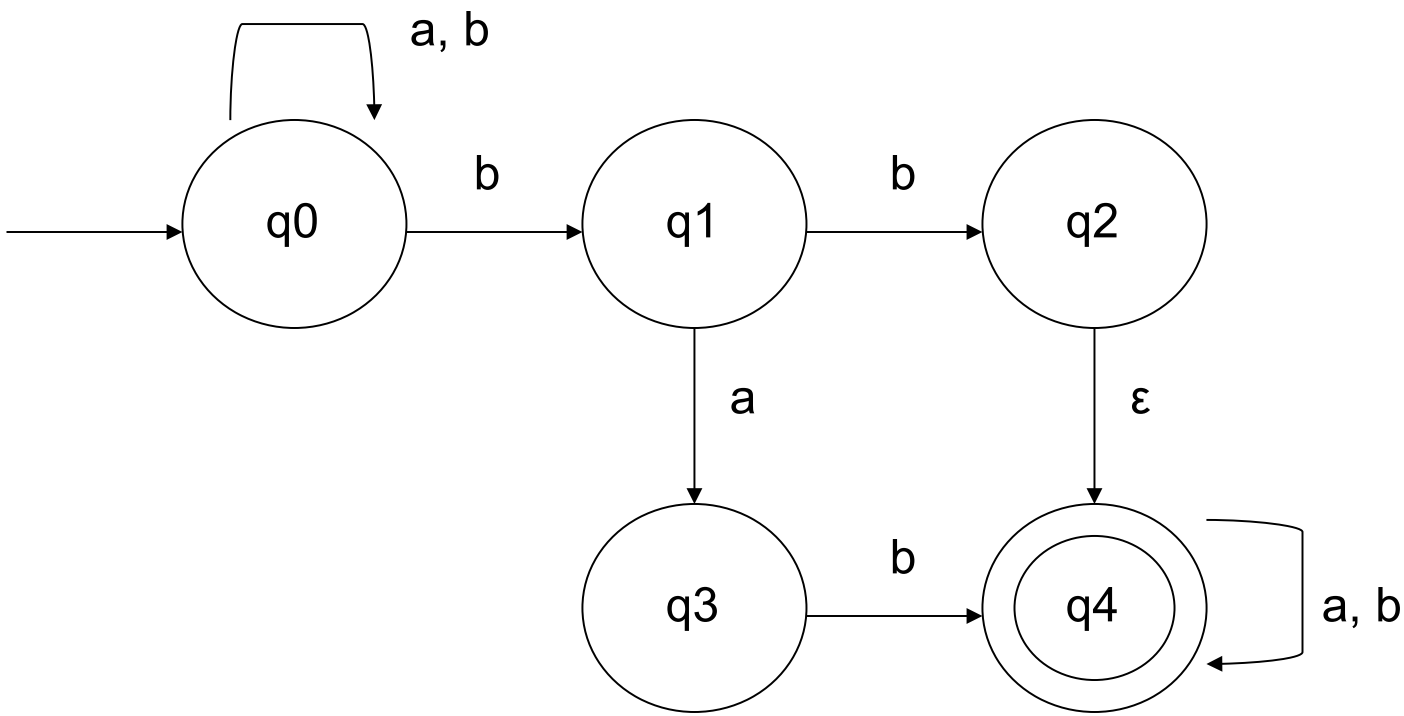
- Is ab accepted by this machine?
- Another traversal: q0, q1

Nondeterministic Finite State Automata



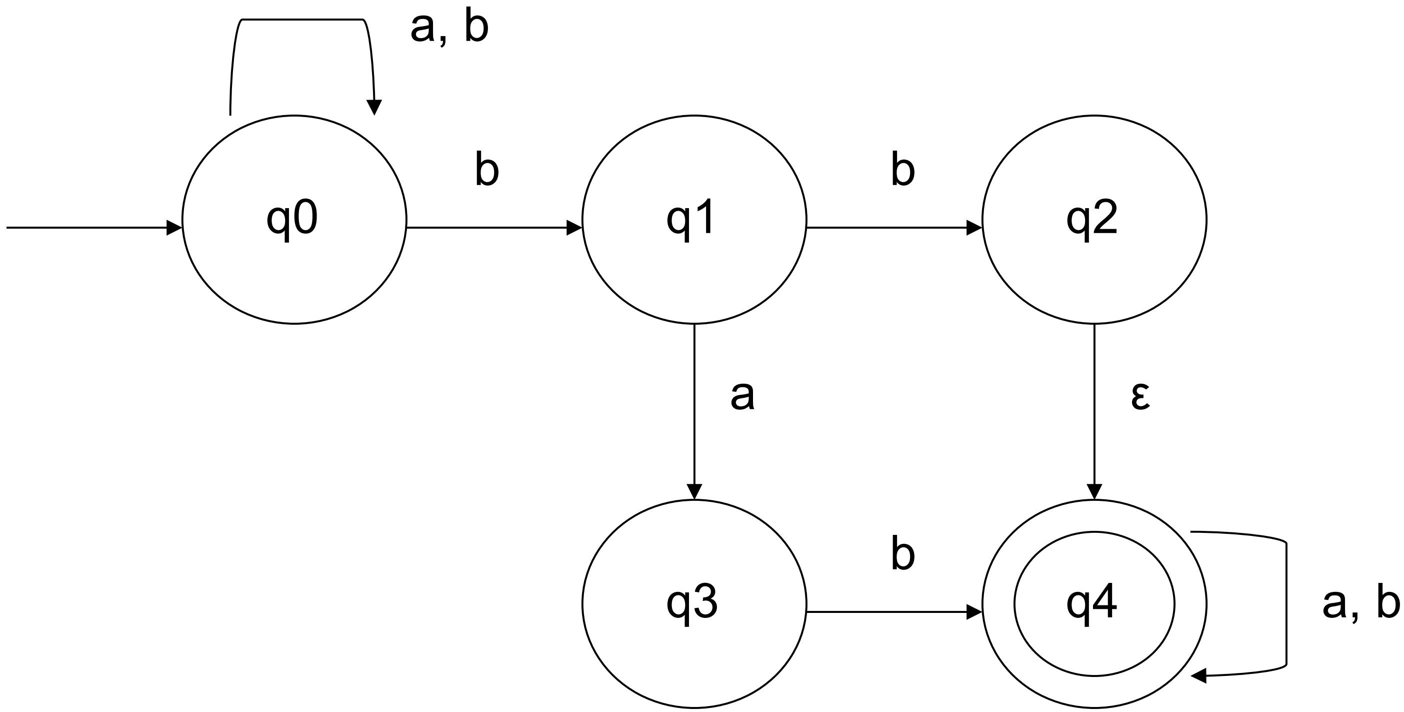
- Is ab accepted by this machine?
- No more traversals. ab is not accepted.

Nondeterministic Finite State Automata



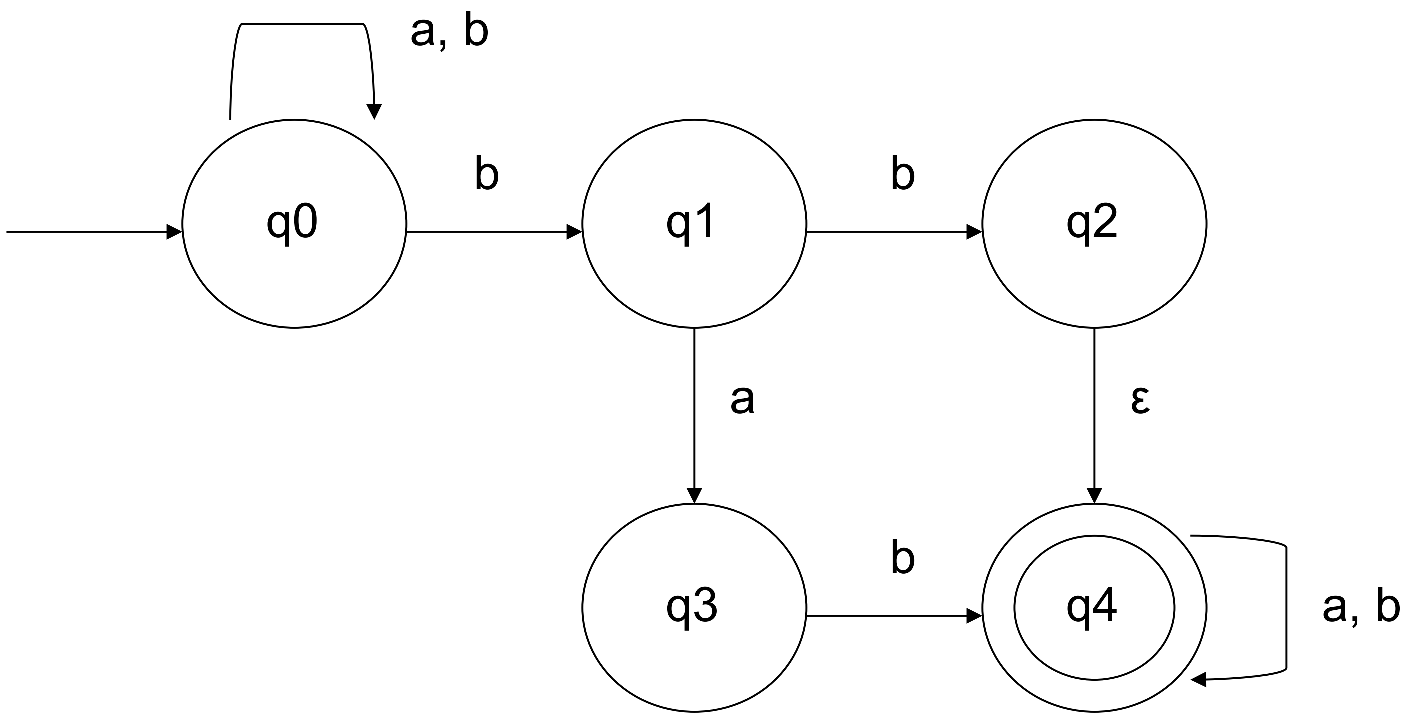
- Is abbb accepted by this machine?
- One traversal: q0, q0, q0, q0, q1

Nondeterministic Finite State Automata



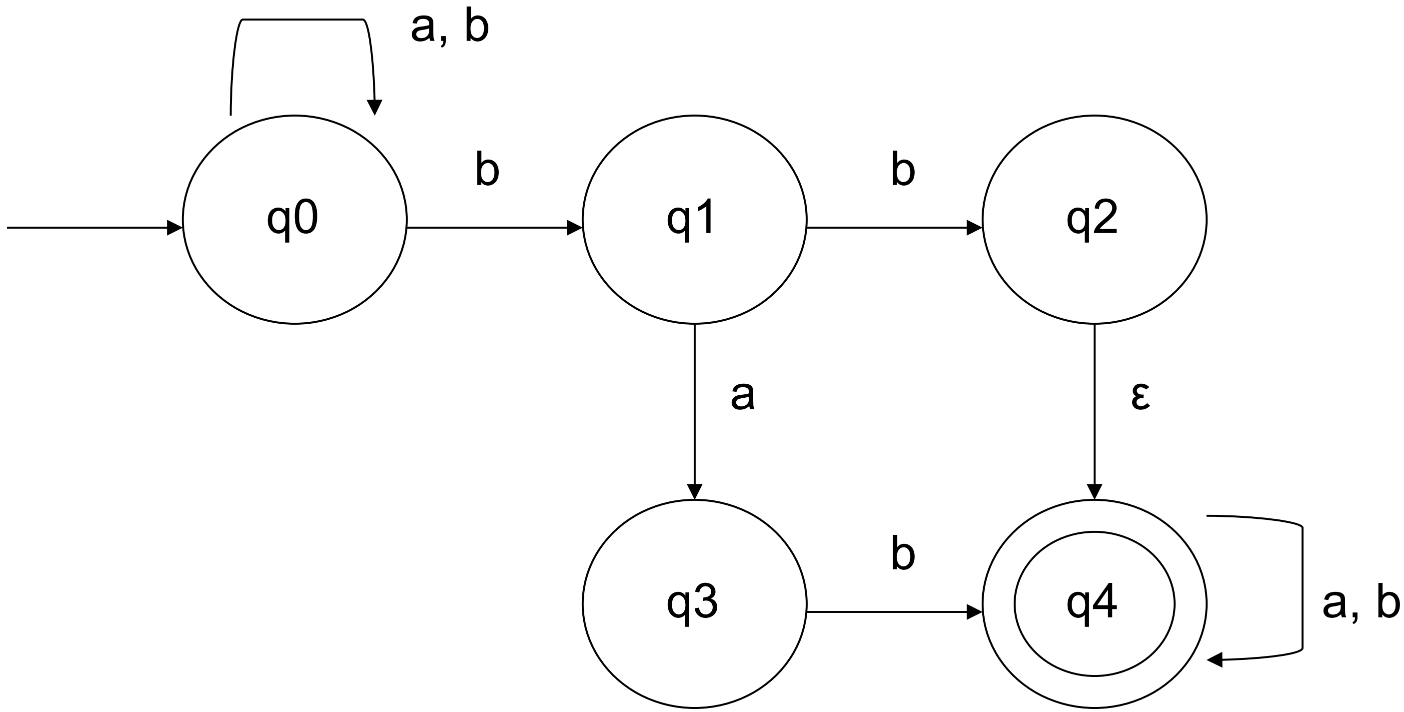
- Is abbb accepted by this machine?
- Another traversal: q0, q0, q0, q1, q2

Nondeterministic Finite State Automata



- Is abbb accepted by this machine?
- Another traversal: q0, q0, q1, q2, q4

Nondeterministic Finite State Automata



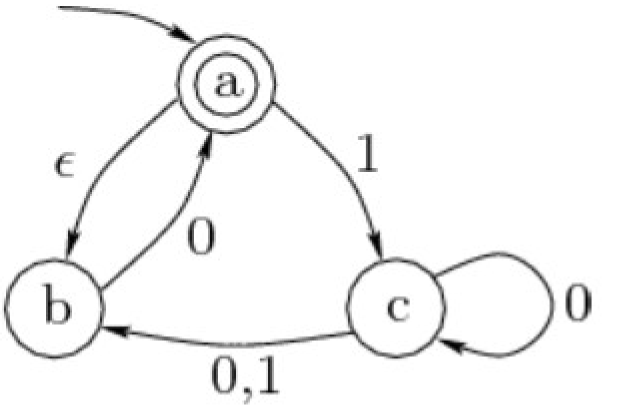
- Is baabaa accepted by this machine?

Nondeterministic Finite State Automata

- You can think of NFA as a rule-based game where if you're stuck, you backtrack to a previous state where there was a remaining choice.

Exercise

- Here's an NFA:



The states are {a,b,c}. The symbols/alphabet of the words is {0,1}. Which of the following strings are accepted/not accepted?

ε, 0, 1, 00, 01, 10, 11, 000, 001, 010, 100, 011, 101, 110, 111

Exercise 1

- Write down a regex for positive and negative integers.
- Write down a DFA for positive and negative integers. How many states do you need?
- Write down a NFA for positive and negative integers. How many states do you need?

Exercise 1 - Solution

- A positive integer
  = non-zero digit followed by any no. of digits
  = (1U…U9) followed by any no. of (0U…U9)
- So regex of pos int is

  posint = (1U…U9)(0U…U9)*.
- What about neg int? It's just – followed by a pos int. So the regex is

  negint = –(1U…U9)(0U…U9)*.

Exercise 1 - Solution

- So the regex for pos and neg int is
  posint U negint =
  (1U…U9)(0U…U9)*U (– (1U…U9)(0U…U9)*)
- It's simpler to write

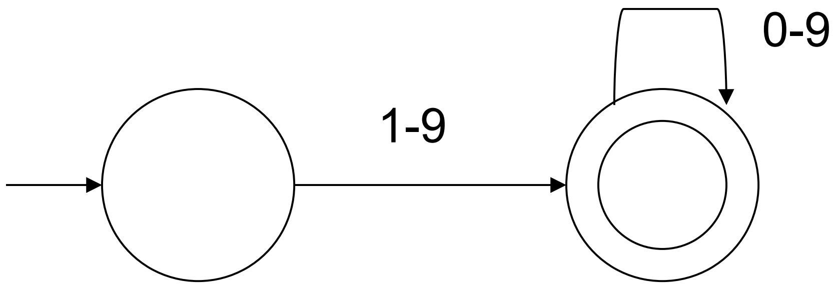
posint U negint = (– U ε) ((1U…U9)(0U…U9)*)

- Note that zero is not included. So altogether the regex for int is

  (– U ε) ((1U…U9)(0U…U9)*) U 0

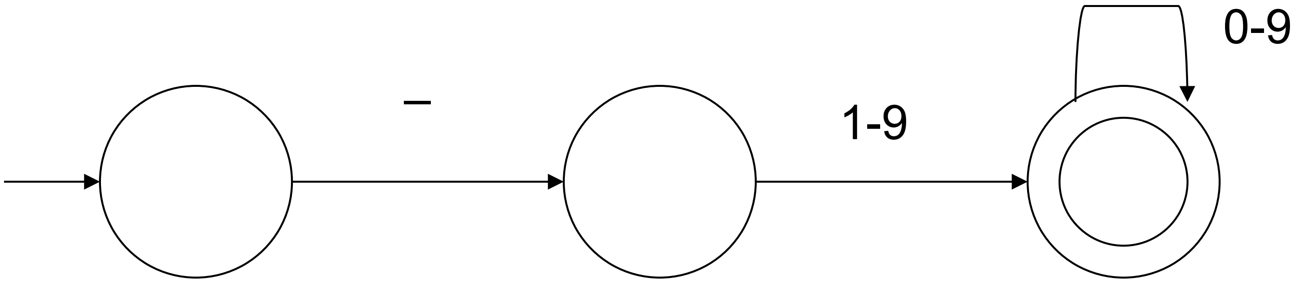
Exercise 1 - Solution

- Now for the NFA. First the pos int:



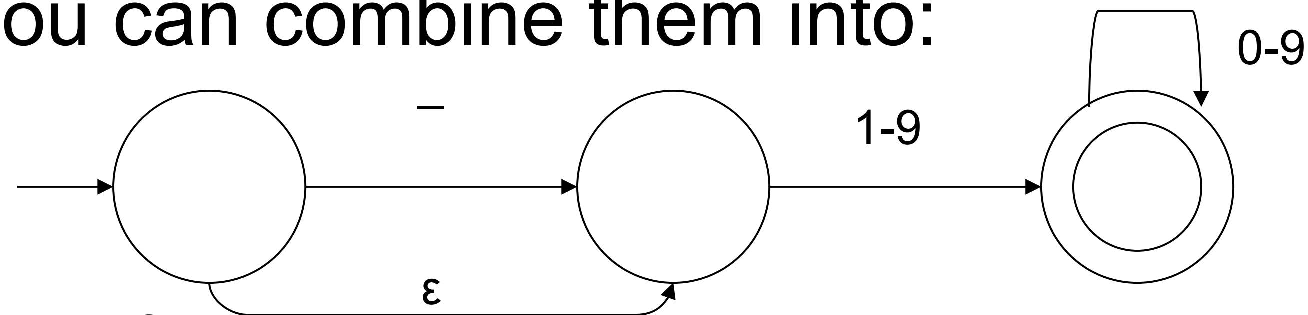
This is basically the regex
(1U…U9)(0U…U9)*

- Neg int looks like this:

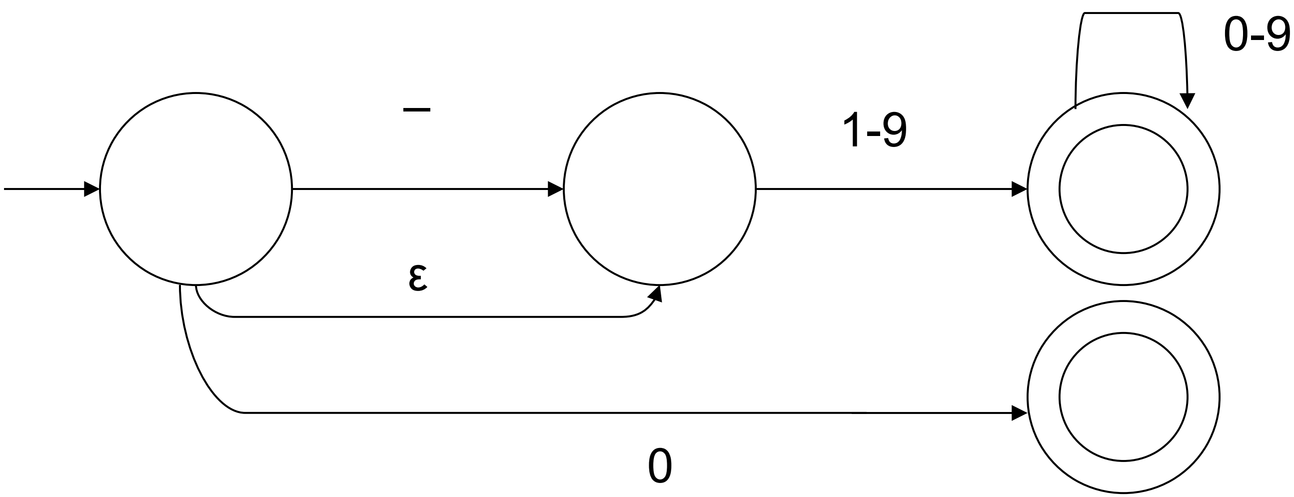


Exercise 1 - Solution

- You can combine them into:



- Now for zero:



Exercise 1

- Write down a regex for positive dollar amounts (up to cents). The follow are valid words in this language:

  $1, $1.23, $0.23, $.23, $1.23, 23c
- Write down a DFA for the above language.
- Write down a NFA for the above language.

Regex in Programming

- Regular expressions are extremely important to a programmer.
- Example of use: input validation, searching
- The following is an example using Python. Most programming use the same syntax for construction of regular expressions.

Regex in Programming

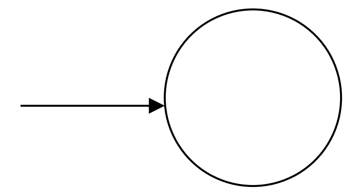
```
>>> import re
>>> pattern = re.compile("[0-9]{3}-[0-9]{3}-[0-9]{4}")
>>> if pattern.match("123-456-7890"): print "OK"
...
OK
>>> if pattern.match("123-456-789"): print "OK"
...
>>> if pattern.match("123-456_7890"): print "OK"
...
>>> pattern = re.compile("[0-9]{3}-[0-9]{3}-[0-9]{4}(ext[0-9]{3,4})
{0,1}")
>>> if pattern.match("123-456-7890"): print "OK"
...
OK
>>> if pattern.match("123-456-7890ext123"): print "OK"
...
OK
```

General Tricks

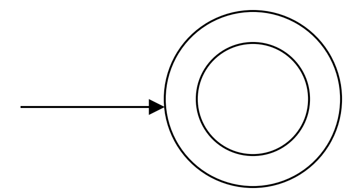
- If you can describe the set of words in your language (example: regex), you can slowly build up the NFA one step at a time and also “glue” NFA together.

General Tricks

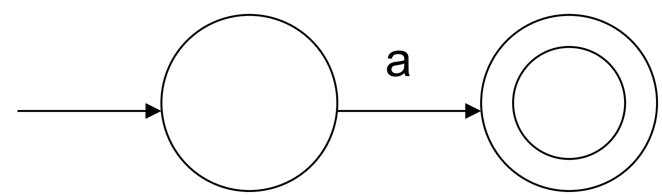
- The NFA accepting { } is



- The NFA accept {ε} is

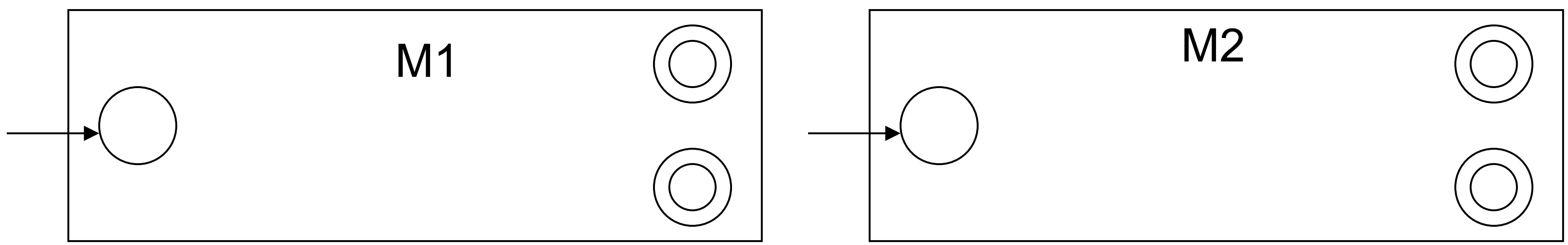


- The NFA accepting symbol a is

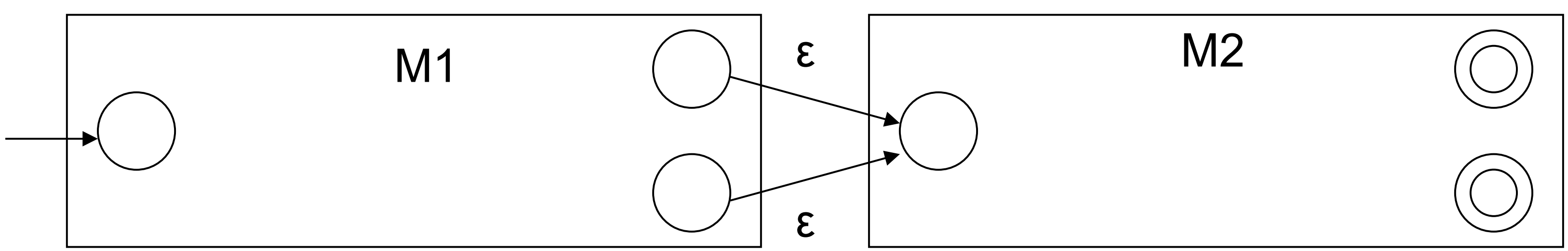


General Tricks

- Suppose you have two NFA M1, M2:



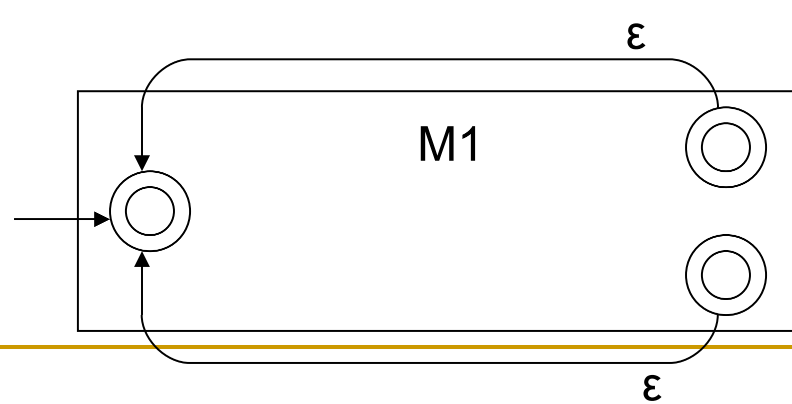
You want to accepts strings of the form s1s2 (s1 concat with s2). Then the NFA is



General Tricks

- Suppose you have the NFA M1

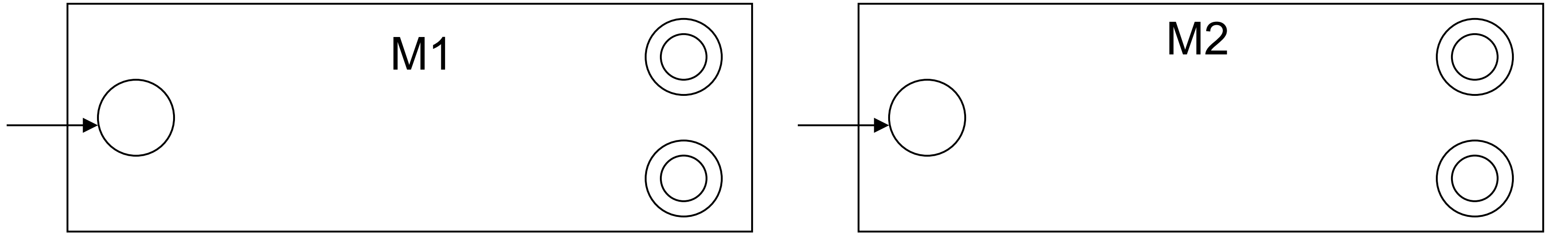
- You want to accepts strings of the form ε or s1s2…sn where s1,s2,…,sn are accepted by M1. Then the NFA is



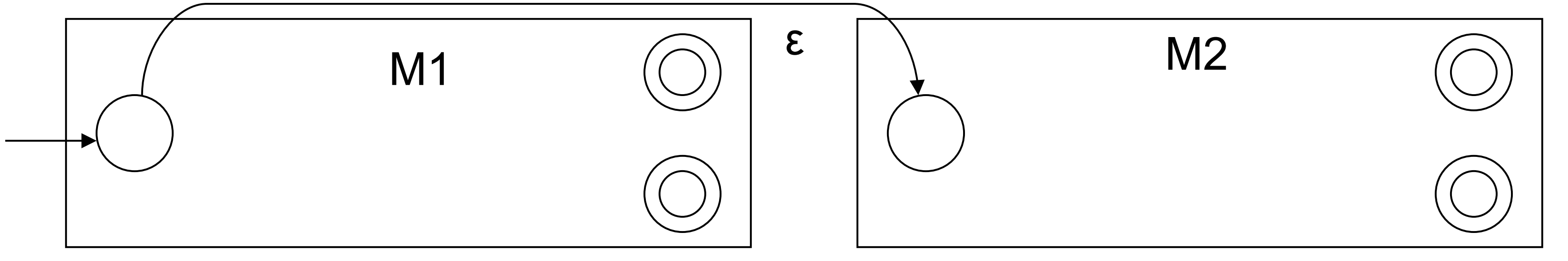
If r is the regex of The language for M1, then the new language is r*.

General Tricks

- Suppose you have two NFA M1, M2:



You want to accepts strings of the form s where s is accepted by M1 or M2. Then the NFA is



General Tricks

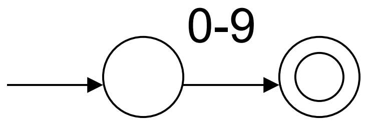
- The above tricks need not yield the “simplest” NFA.
- The complexity of a finite state automaton (DFA or NFA) is usually measured in terms of the number of states.

Example

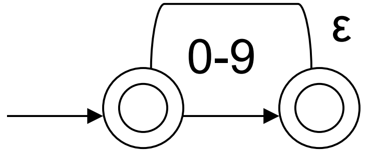
- Here is an example of building up an NFA using the above tricks.
- What is the NFA for recognizing an expression of the form <integer> <op> <integer> where <integer> is a positive integer and <op> is either + or -.
- We'll go this slowly.

Example.

- This NFA accepts a digit.

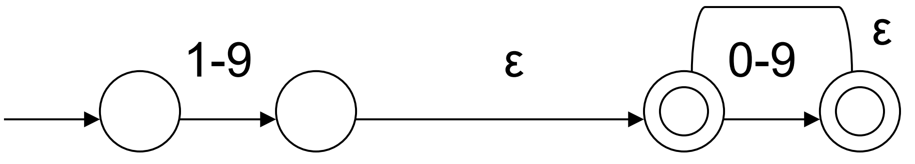


- This NFA accepts any number of digits (including none).

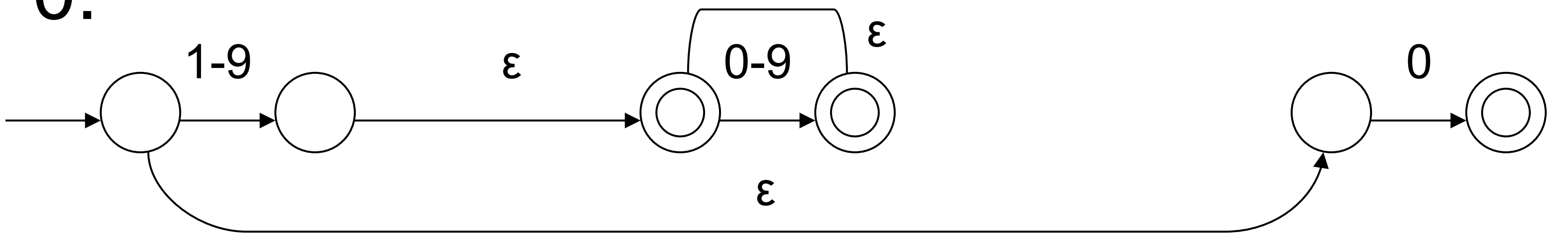


Example.

- Now we concat a nonzero digit in front:

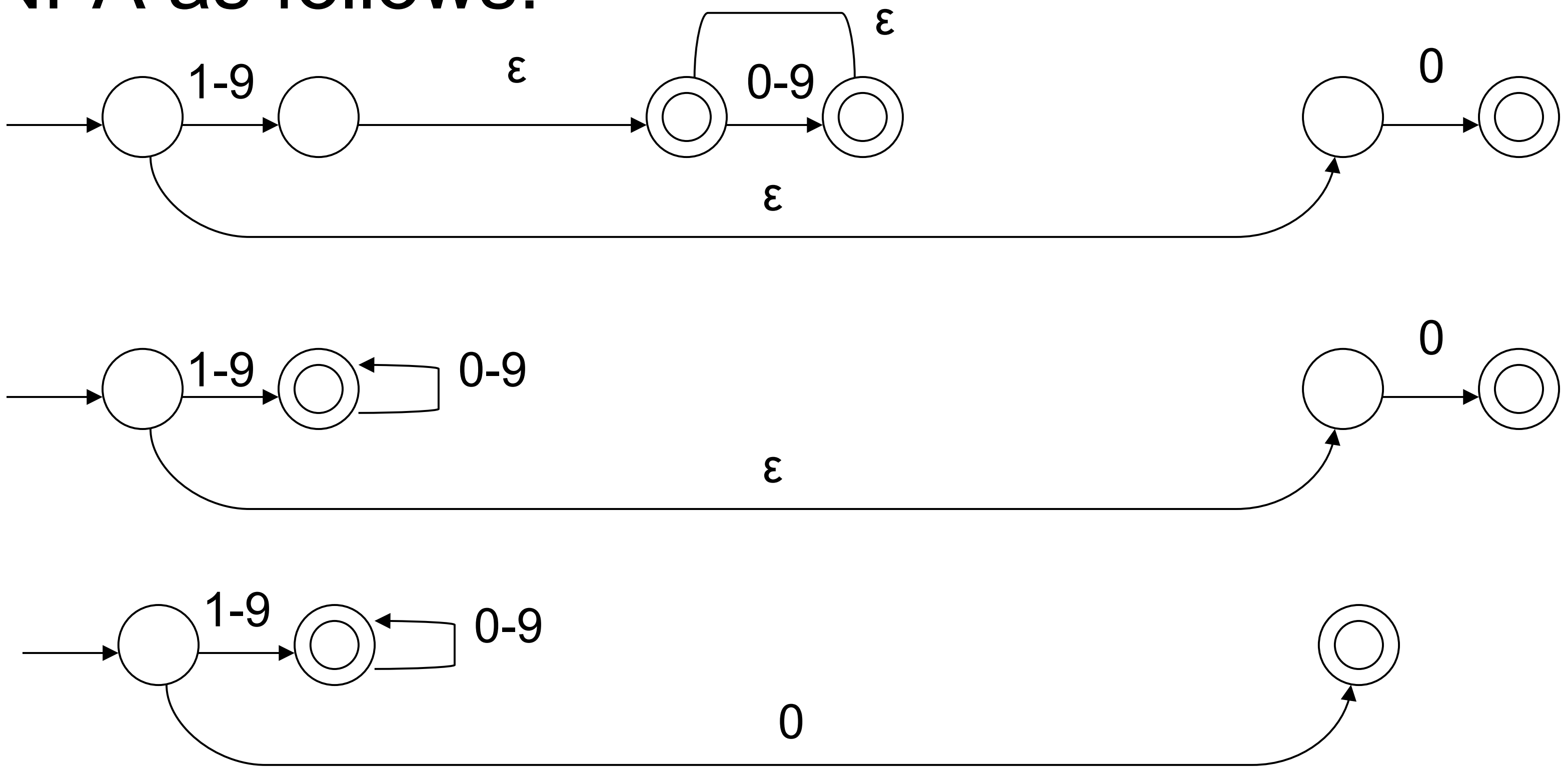


- There’s a number that can start with a zero and that is 0. We want to allow the above or 0:



Example.

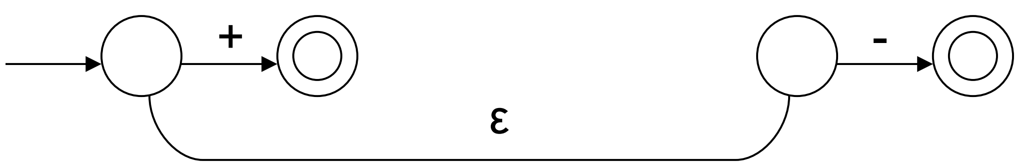
- Now if you think about it you can simply the NFA as follows:



Example.

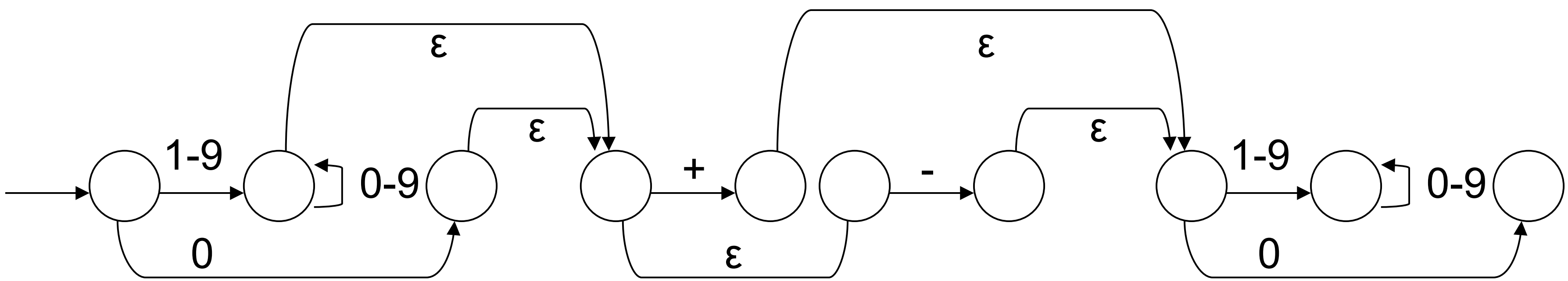
- OK. We now have the NFA for a positive integer.
- The NFAs for the operations are

- So the NFA for either the + or the - is



Example.

- Finally the NFA for the require expression is joining up the one for the positive integer with the one for the operation and then the positive integer again:



Big Picture

- Recall that a regular language is the set of words (i.e. strings) over a fixed set of symbols generated by a regular expression.
- We have also the deterministic finite state machine and the nondeterministic finite state machine.
- It turns out that these three devices have equal power: Any language described by a regex can be described by a DFA and a NFA, etc.

Big Picture

- A program starts off by being just a string.
- We need to convert the program string into an abstract syntax tree AST (see lecture notes from first week).
- There are two stages: lexing and parsing.
  - Lexing: Converting the string into lists of tokens (i.e. valid words in the language)
  - Parsing: Converting a list of tokens into an AST
- Lexing involves regular languages

Big Picture

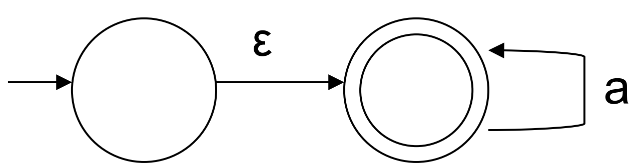
- Note that a regex is a string. Most lexing and parsing tools for compiler construction uses regex.
- Regex describes what strings are valid. DFAs are used for string recognition.

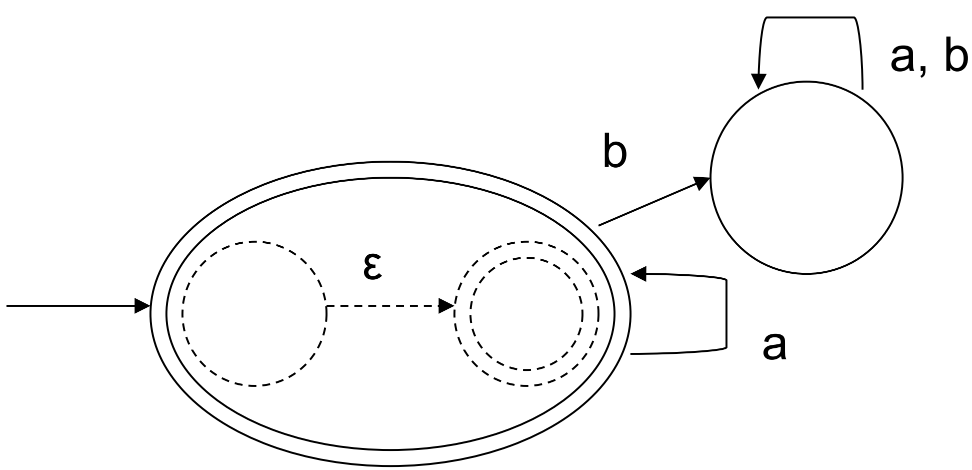
Translating NFA to DFA

- The idea is very simple. Recall the differences between DFA and NFA with symbols in S:
  - For a DFA, transitions are labeled with symbols in S. For a NFA, transitions are labeled with symbols in S or with ε.
  - For each state in a DFA, there is exactly one transition labeled with each symbols in S. There is no such restriction for NFA.

Translating NFA to DFA

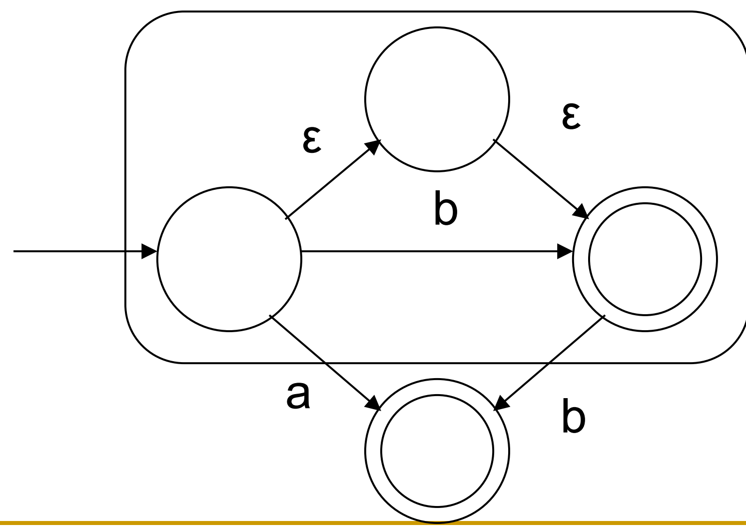
- You're given an NFA.
- You want to write down the DFA that accepts the same set of words.
- Example: S = {a,b}





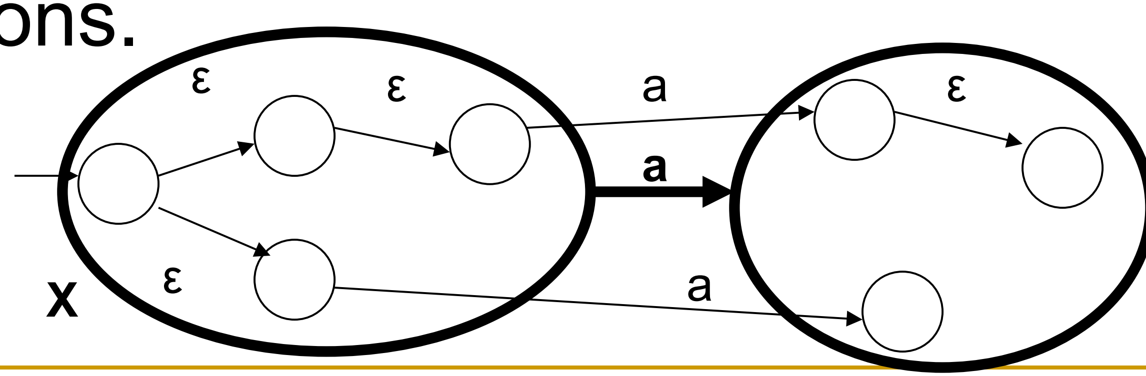
Translating NFA to DFA

- Start with the start state. Look at all the states you can reach from the start state using ε transitions. This is the ε-closure of the start state.
- Example:



Translating NFA to DFA

- The ε-closure of the start state of the NFA is the start state of the new DFA.
- From the states in the ε-closure X of start state, for each symbol s in S, look at all the states you can reach from X using s, including those that you can reach using ε transitions.

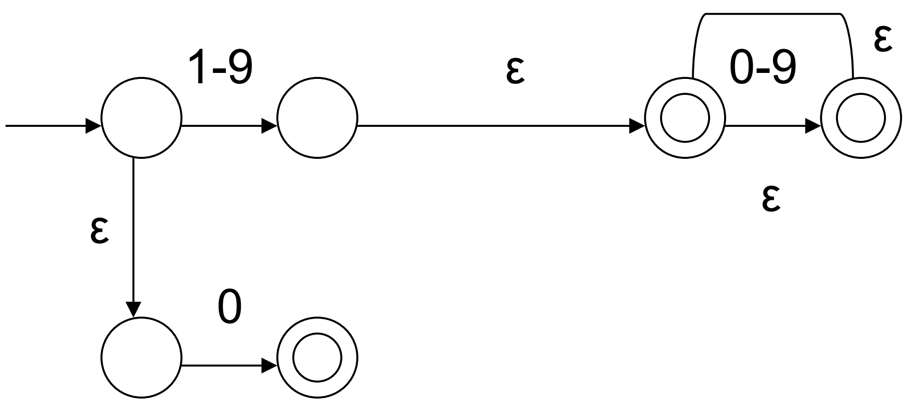


Translating NFA to DFA

- Repeat until every bunch of states have a complete set of transitions.
- When you're done (i.e., no more states left not bunched up) there might be bunches of states without the complete set of transitions – the missing transitions.
- Create a new dummy state for the bunches of states to transition to for each missing symbol.
- The accepting states of the DFA are those states containing at least one accept state from the NFA
- Note that a state from the NFA can appear in two different bunches of states.

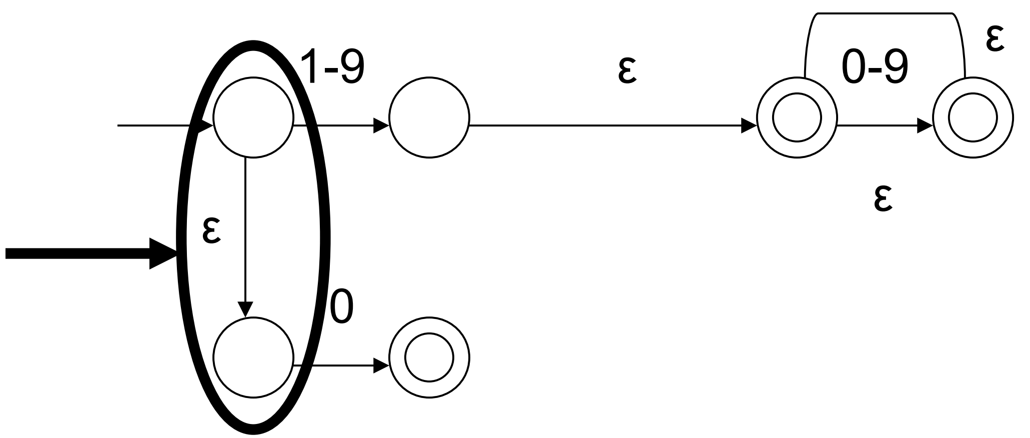
Example

- Example. What is the DFA of positive integers?
- First write down the NFA:



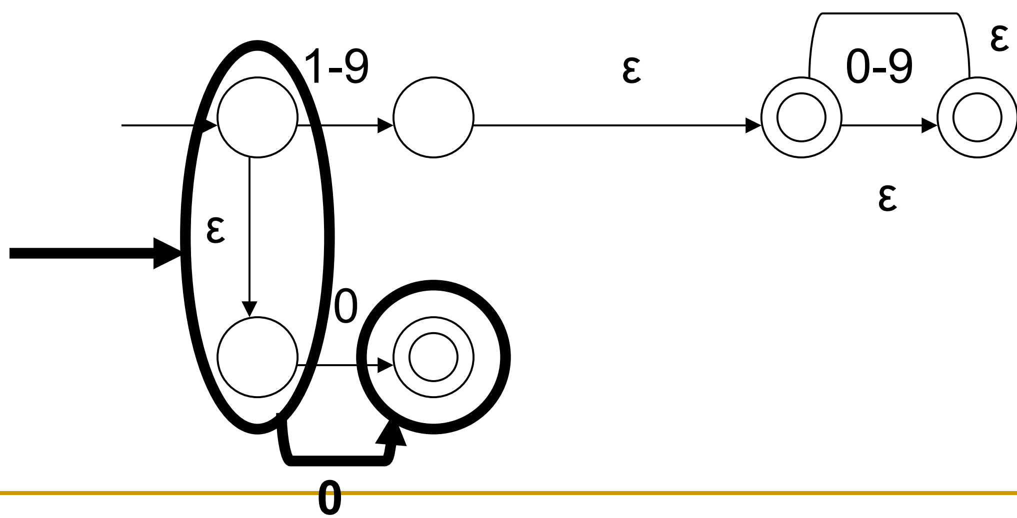
Example

- Now to construct the DFA.
- Start with the start state of the NFA. The ε-closure of the start state of the NFA is the start state of the DFA



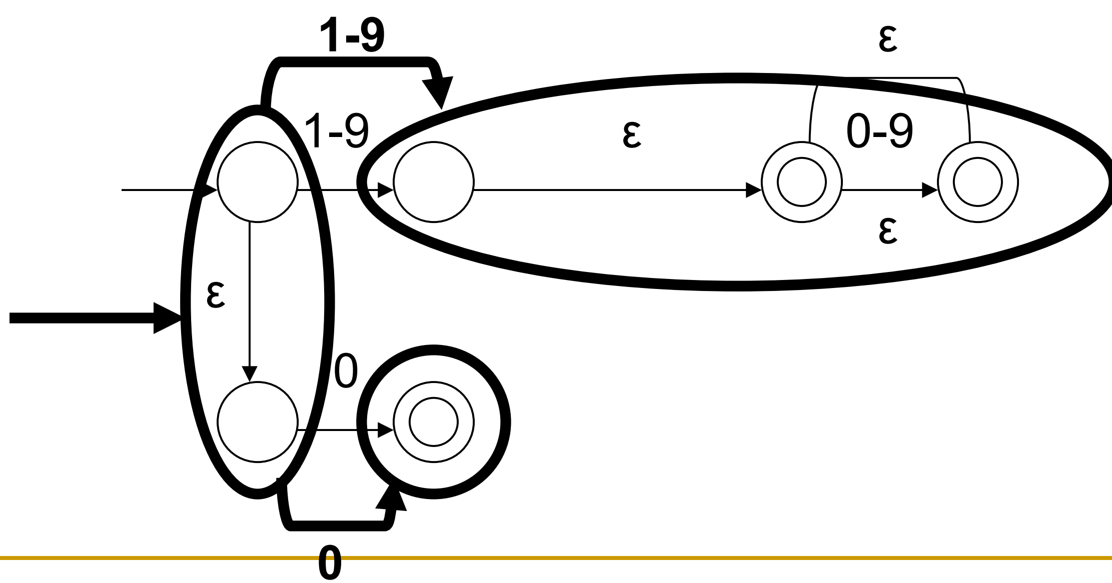
Example

- From the start state of the NFA, i.e. the ε-closure of the start state of the DFA, first look at the 0-transition, then include the ε-closure.



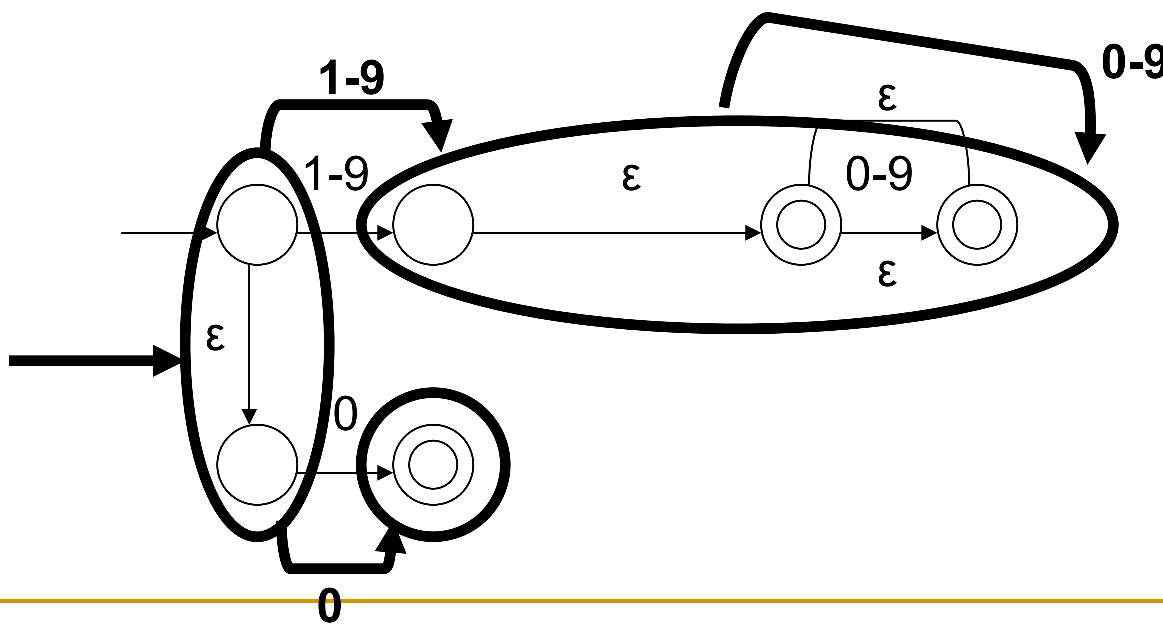
Example

- Now look at the 1-transition. Include the ε-closure. Note that this is the same for transition 2, 3, …, 9.



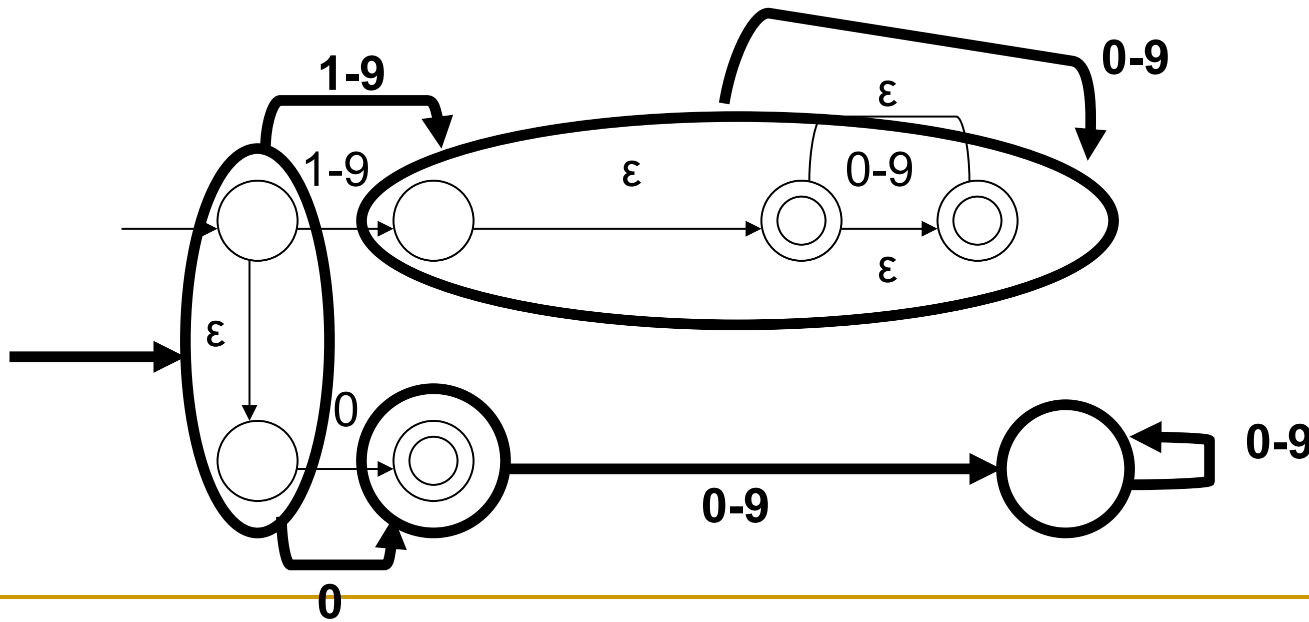
Example

- Repeat …



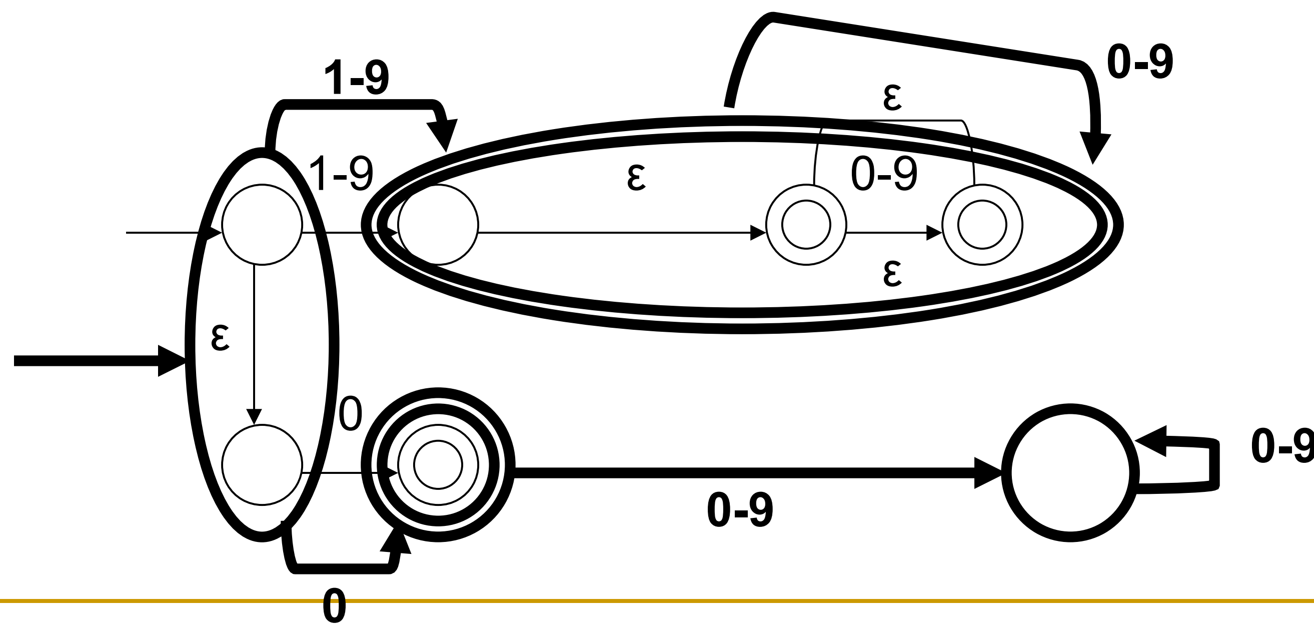
Example

- Now create a dummy state for those with incomplete set of transitions. This is the "dead end" state.



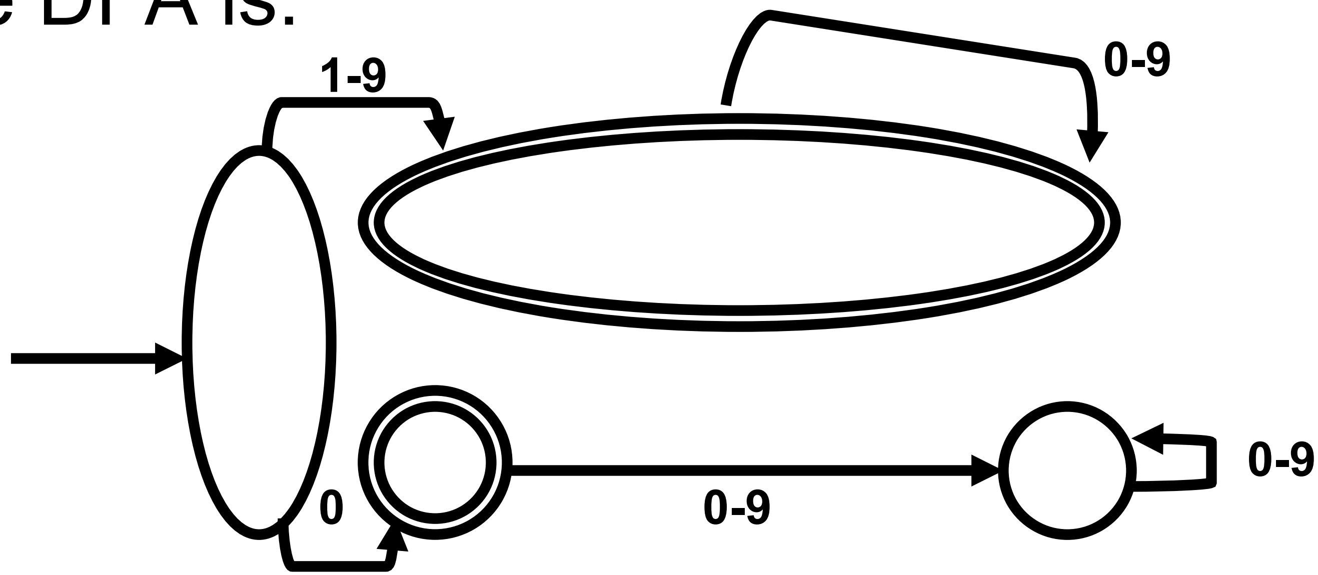
Example

- The final states of the DFA are those containing at least one final state from the NFA.

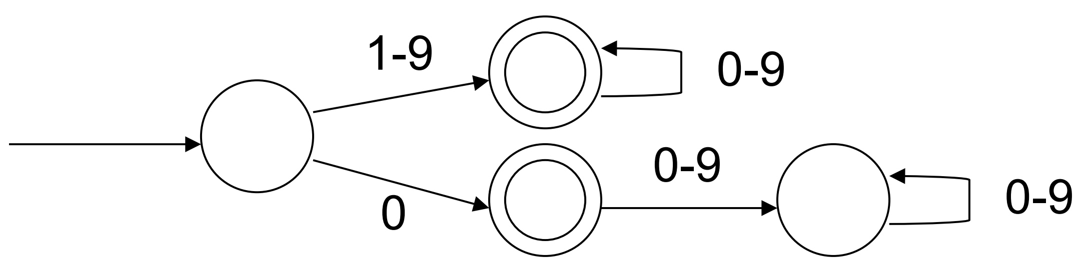


Example

- The DFA is:

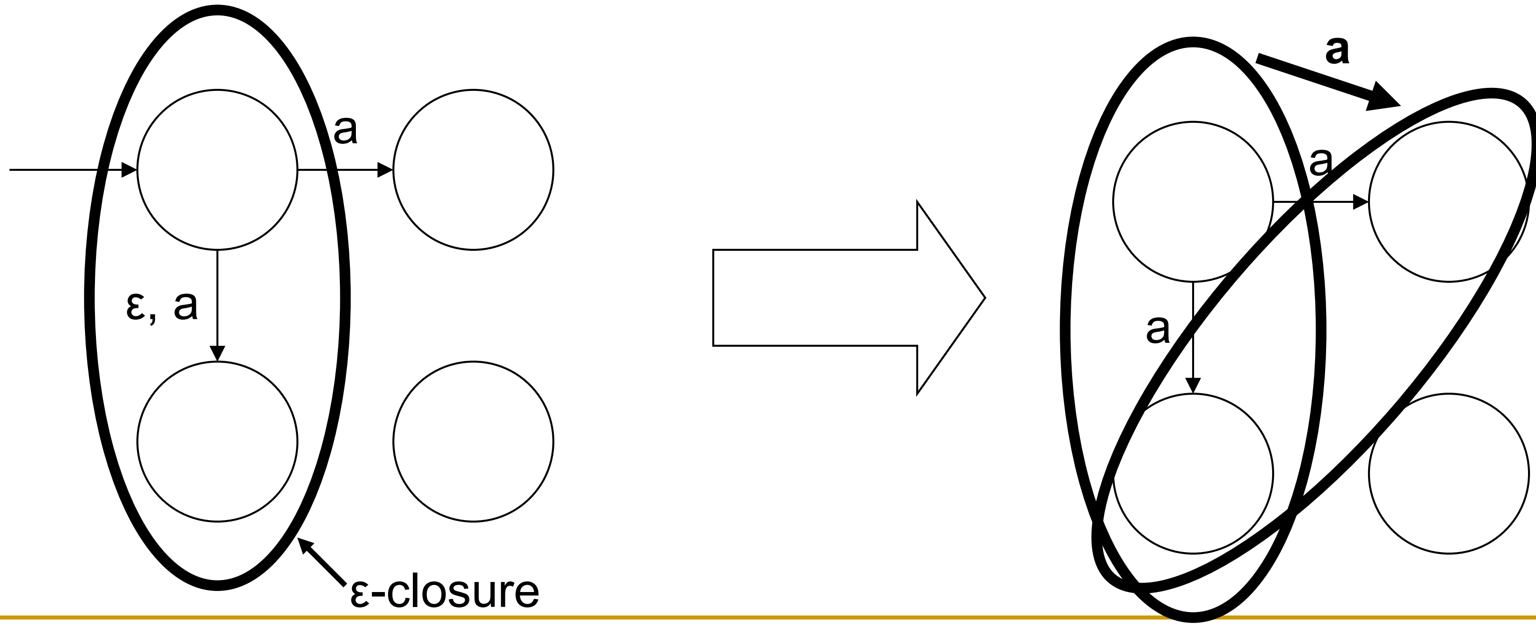


i.e.



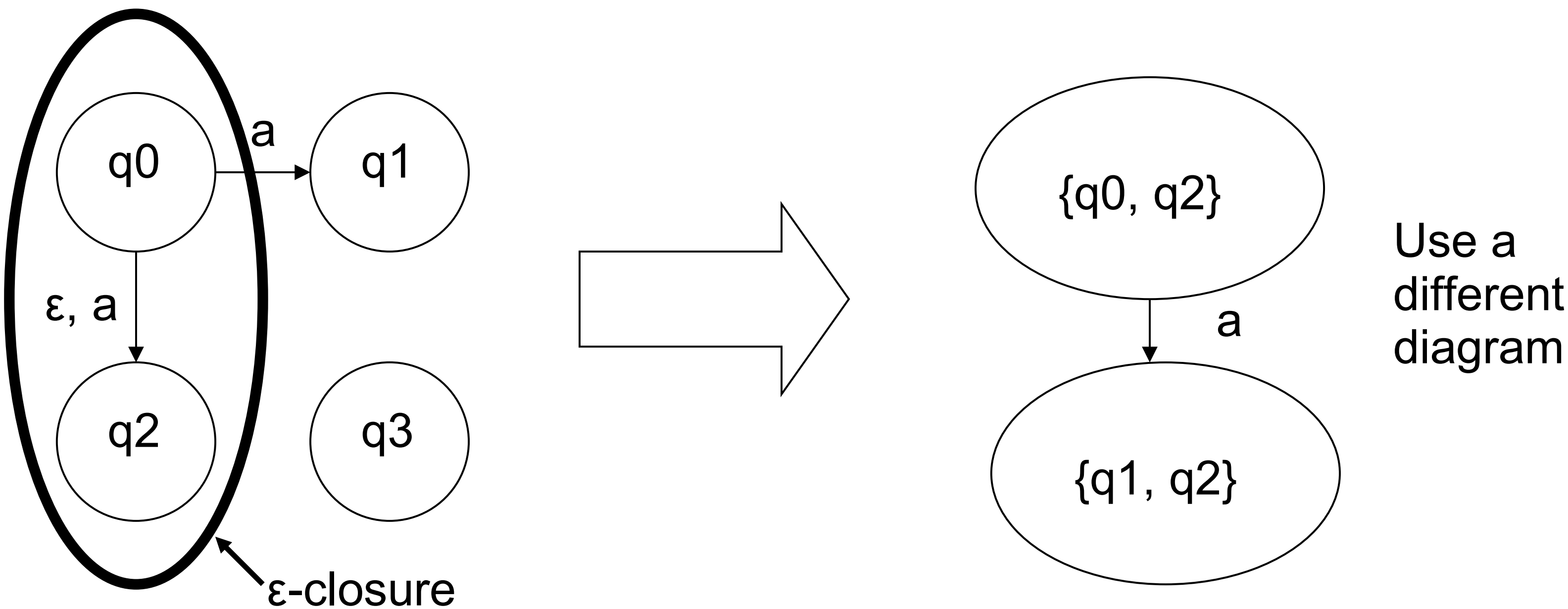
Translating NFSM to DFSM

- Note that sometimes a state appears in two bunches.
- Example. When you follow the a-transition:



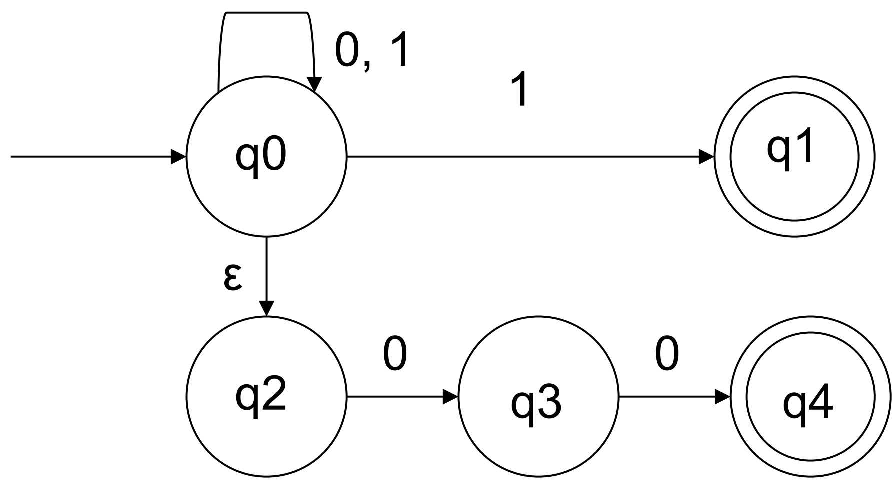
Translating NFSM to DFSM

- It's neater to name all the states in the DFA and for the NFA name the states with a collection of state names from the DFA:



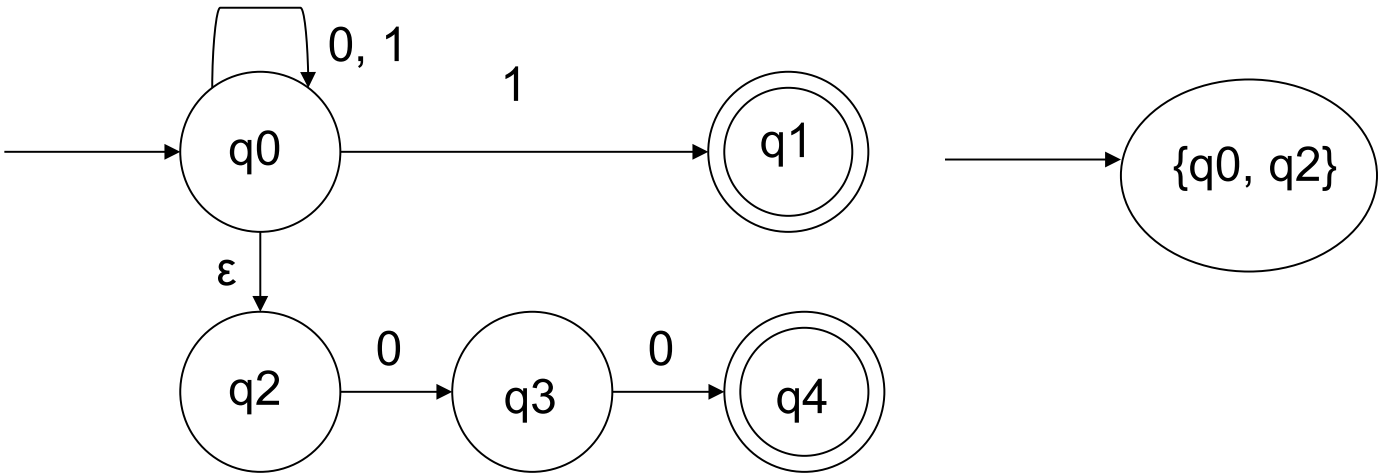
Example

- What is the DFA for the set of words with symbols in {0,1} ending with 1 or 00.
- The NFA is



Example

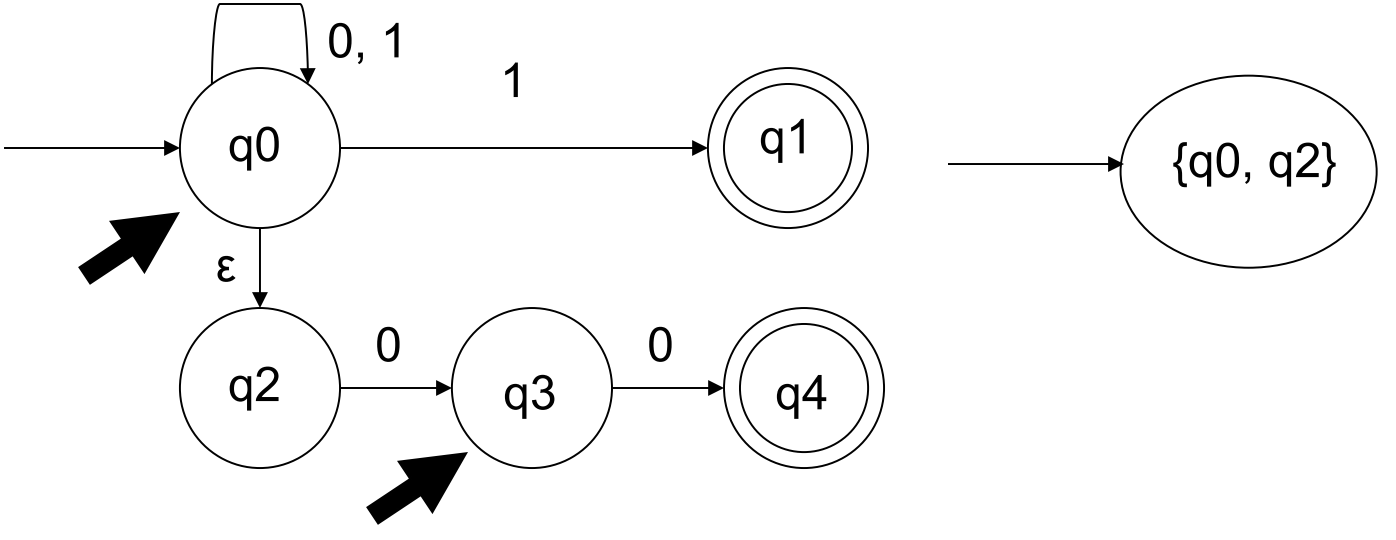
- ε-closure of the start state



- Now compute the 0-transition of {q0,q2}…

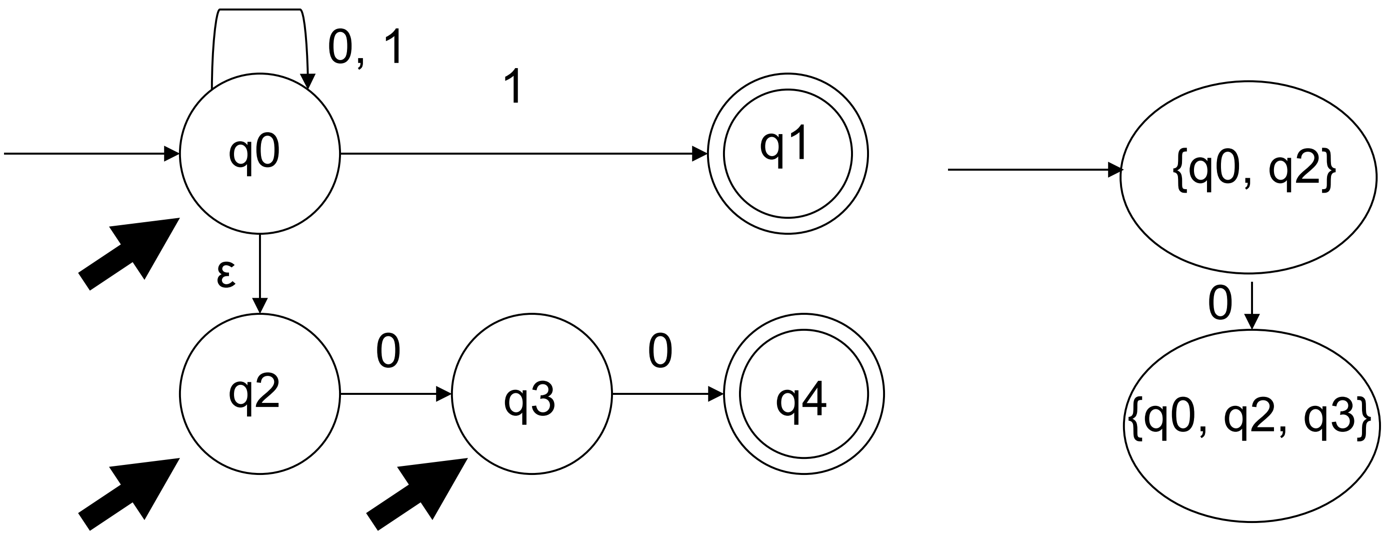
Example

- Look at where q0, q2 can go to using 0-transitions …



Example

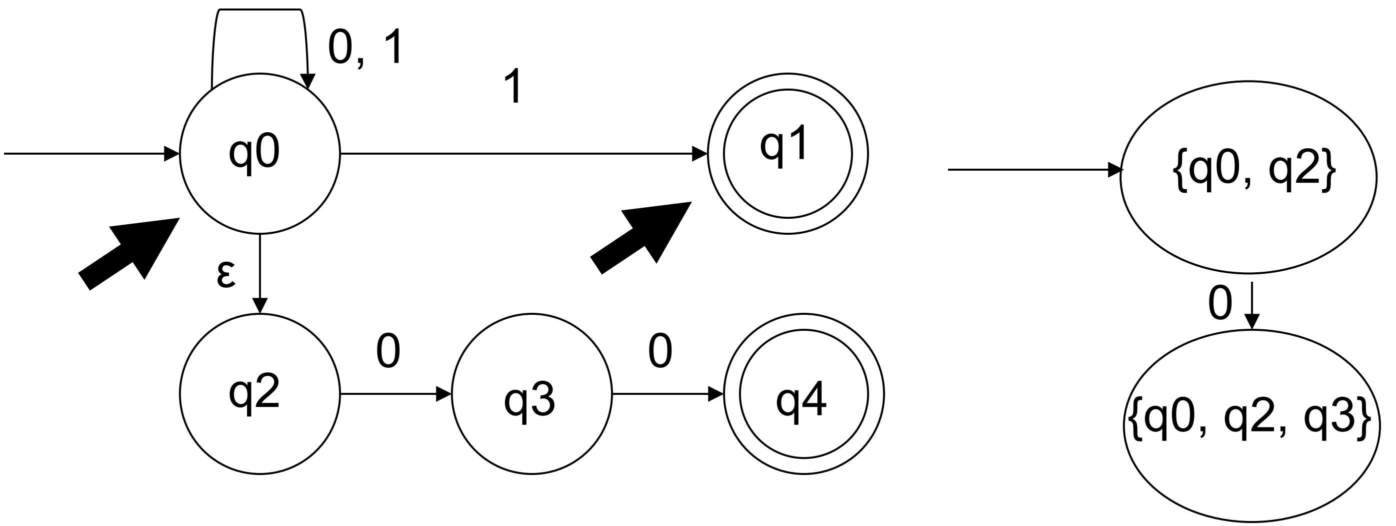
- … and include the ε-transitions to the ε-closure



- Now for the 1-transition of {q0,q2}

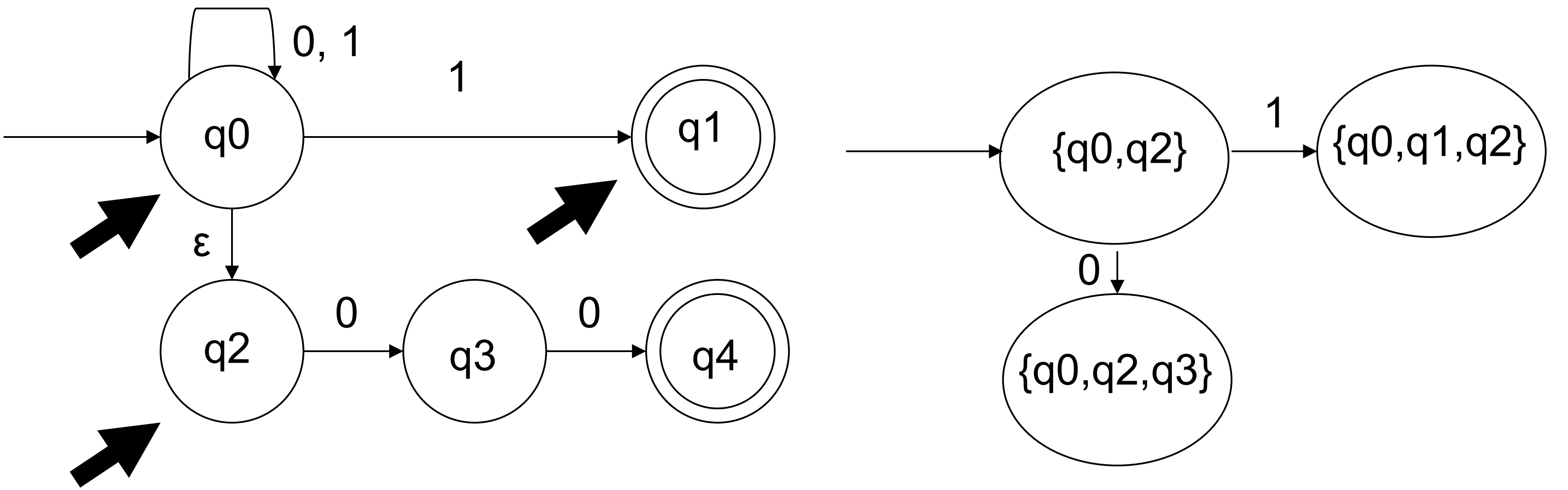
Example

- q0, q2 go to q0, q1 with 1-transitions …



Example

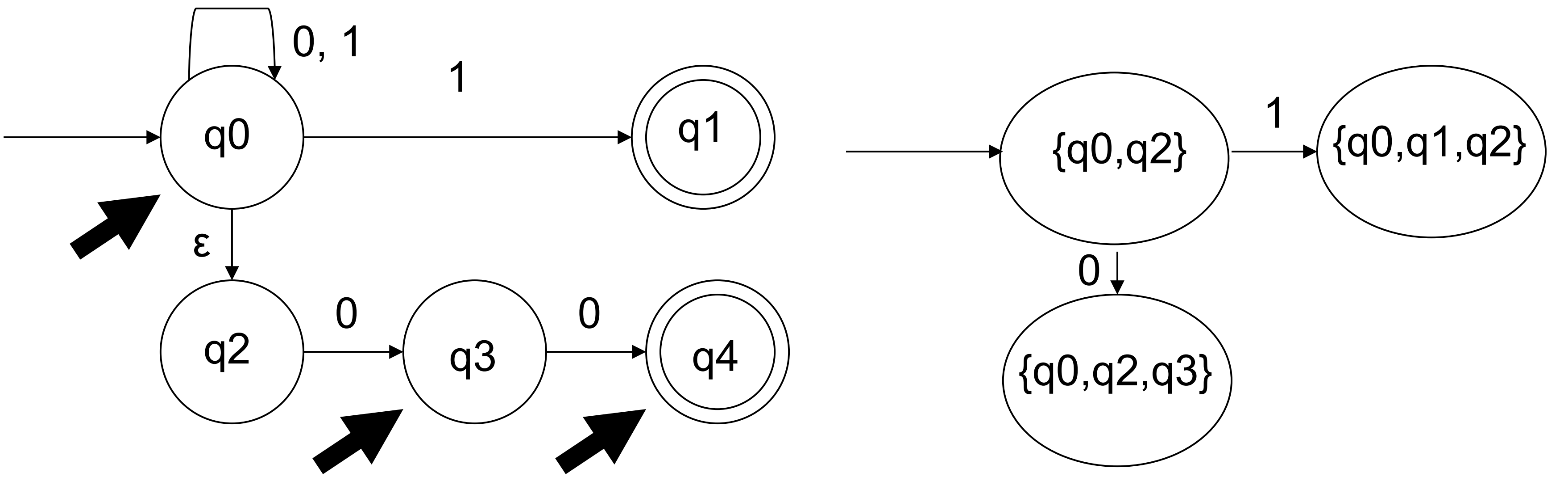
- … including the ε-transitions …



- Go on to {q0, q2, q3}. First the 0-transition.

Example

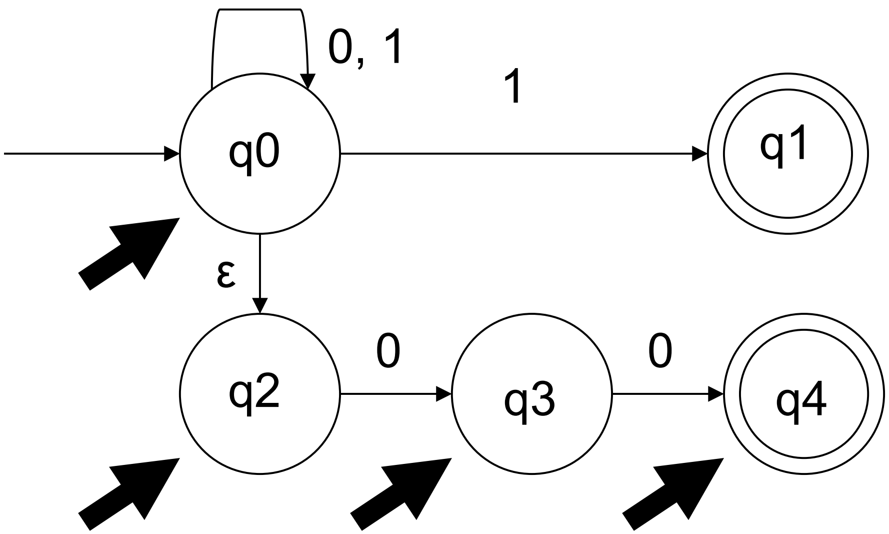
- q0,q2,q3 goes to q0, q3, q4

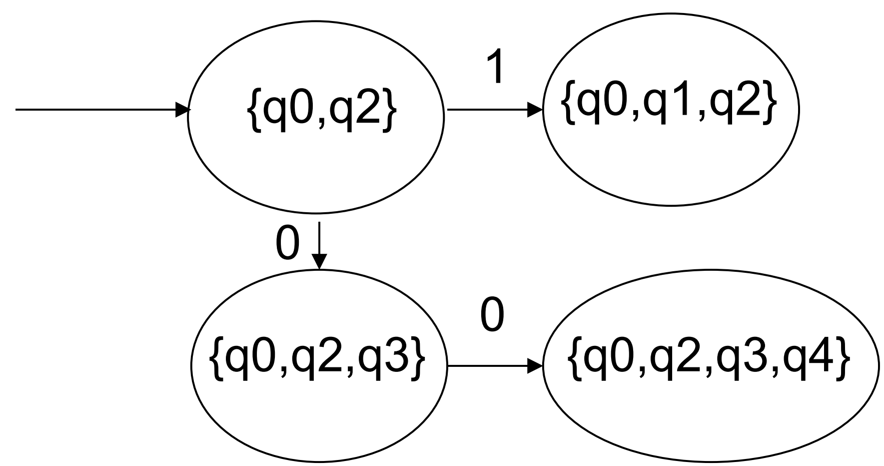


- And including ε-transitions …

Example

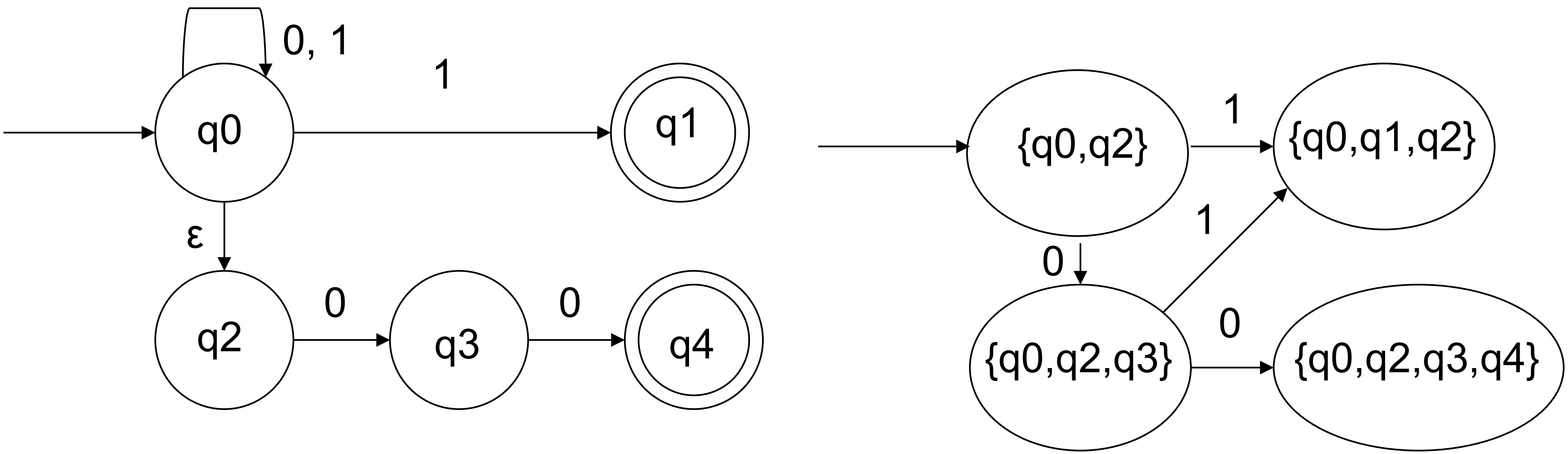
- and we get q0, q2, q3, q4





Example

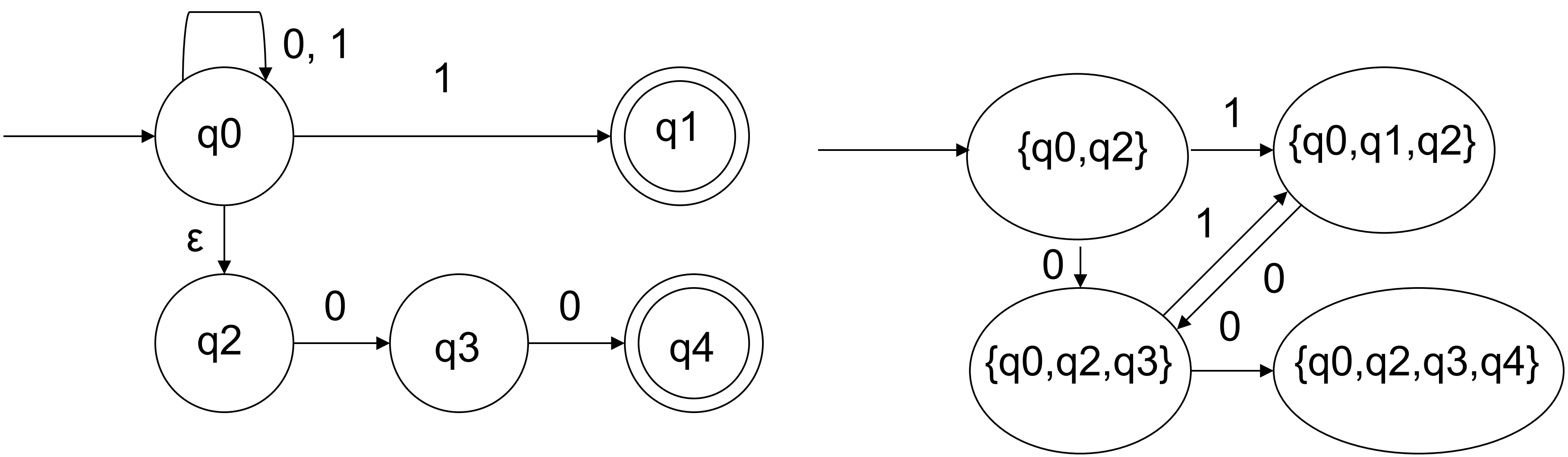
Using 1-transitions q0,q2,q3 can go to q0, q1. The ε-closure is q0, q1, q2



- Now for q0,q1,q2

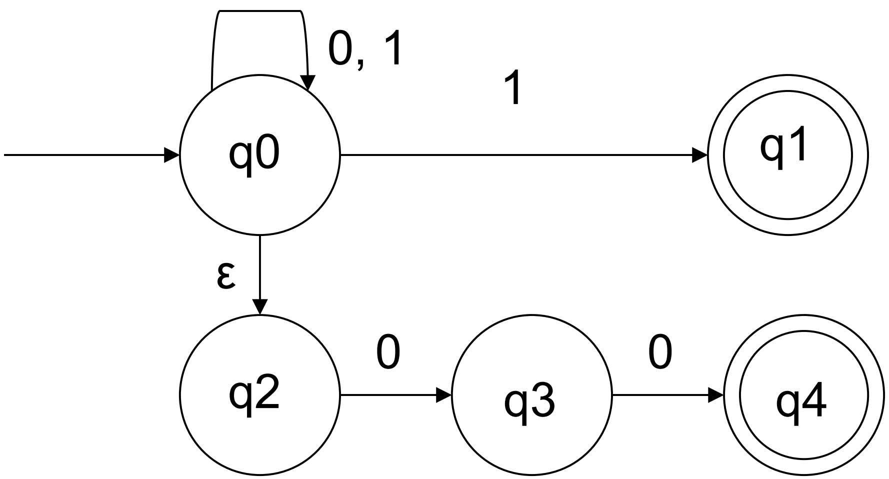
Example

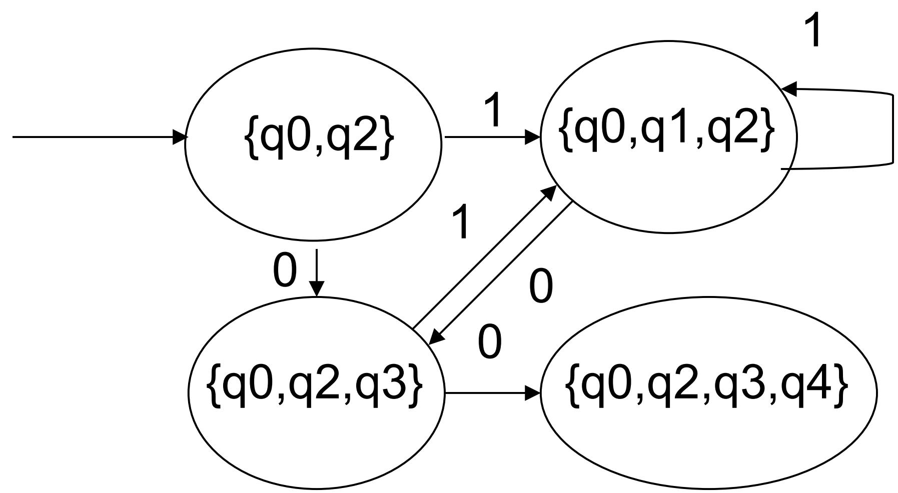
- Using 0-transitions, q0,q1,q2 go to q0,q3. The ε-closure is q0,q1,q3.



Example

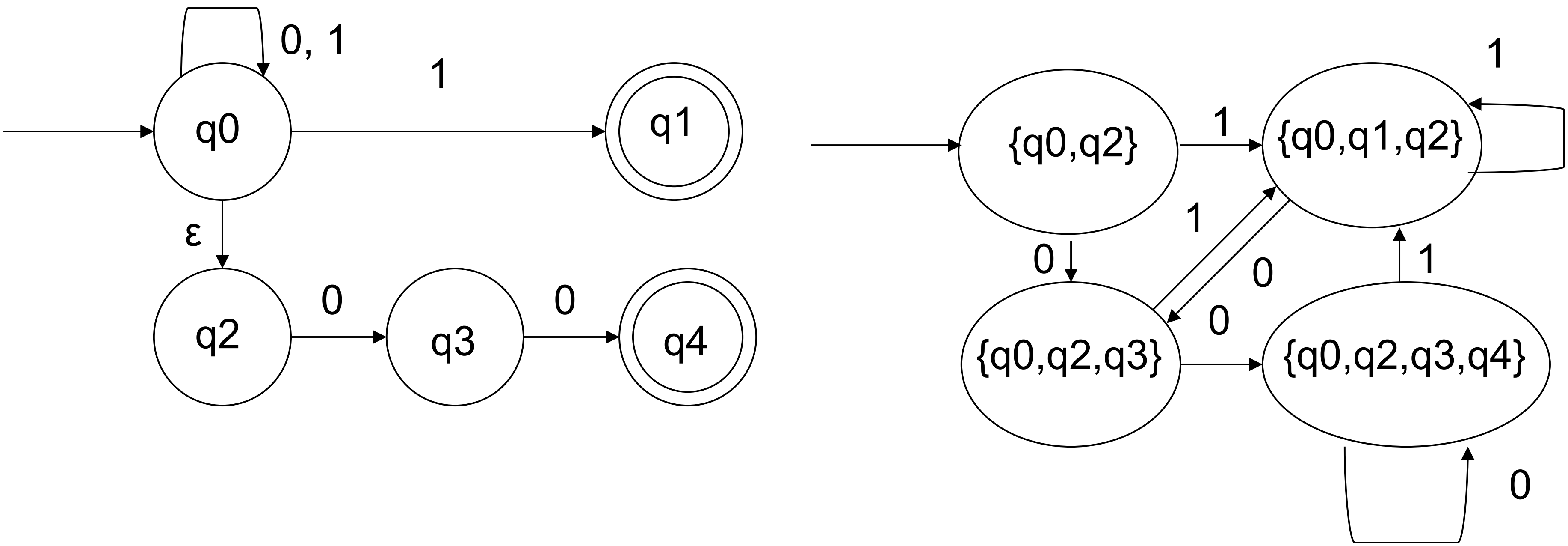
- Using 1-transitions, q0,q1,q2 go to q0. The ε-closure is q0,q1,q2.





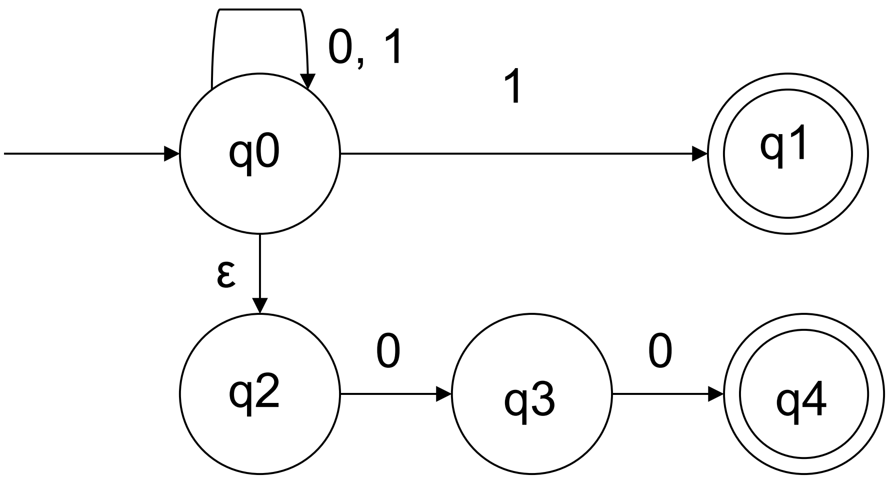
- Now for q0,q2,q3,q4.

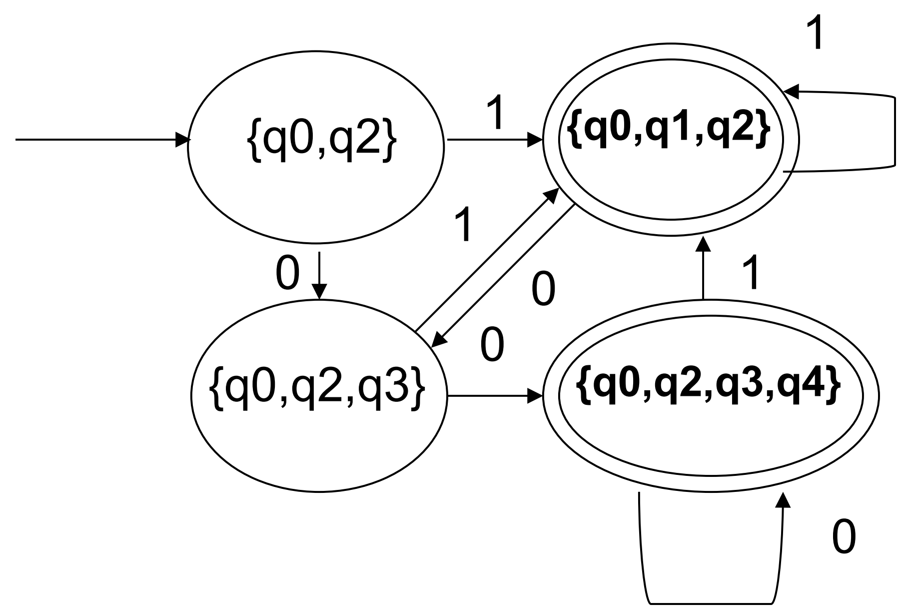
Example



Example

The final states of the NFA are those containing at least one final state of the DFA, i.e. q1 or q4.

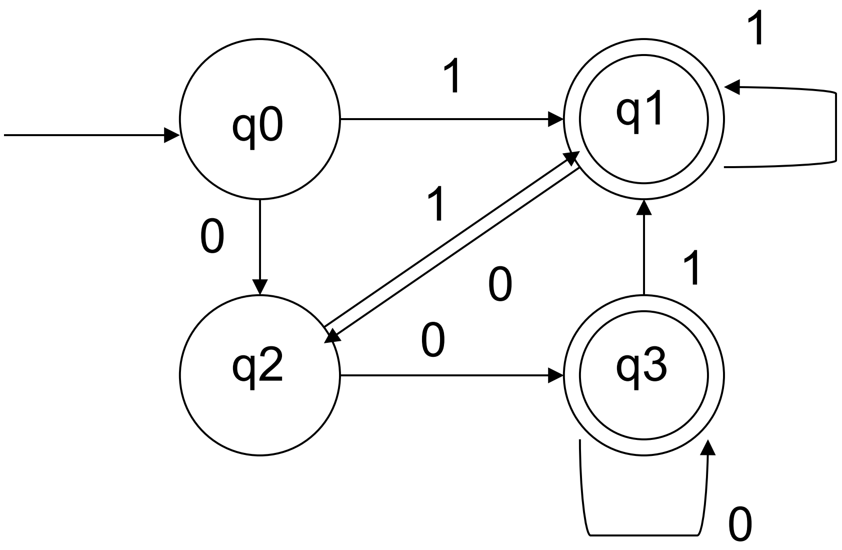




Example

C/C++ pseudocode:

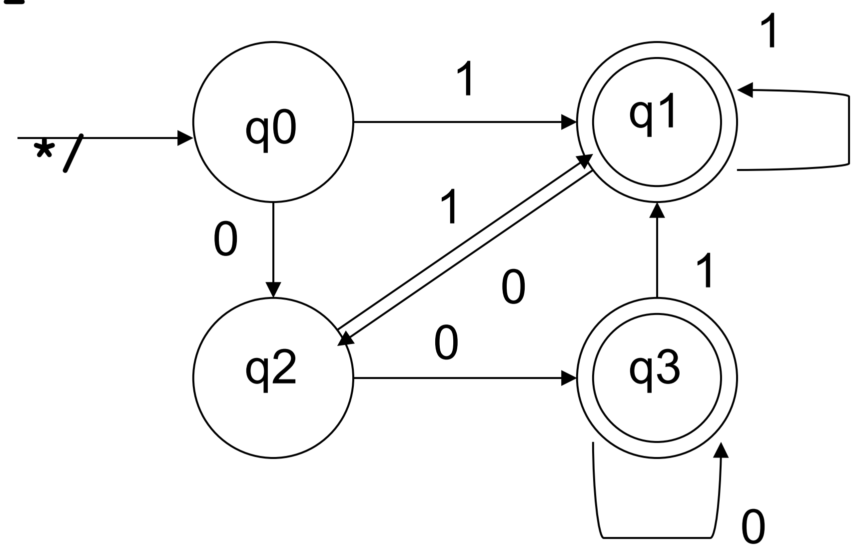
```
q0: if EOF goto REJECT
    c = getchar();
    if (c=='0') goto q2;
    else if (c=='1') goto q1;
q1: if EOF goto ACCEPT
    c = getchar();
    if (c=='0') goto q2;
    else if (c=='1') goto q1;
...
REJECT: cout << "reject";
        goto EXIT
ACCEPT: cout << "accept";
EXIT:
...
```



Example

C/C++ pseudocode:

```
Or use a transition table where
rows = states (q0 = row 0, q1 = row
1, etc.) and columns = character
read.
Let 2D array t be
   {{2,1},/* at q0,read 0 => q2 */
    {2,1},
    {3,1},
    {3,1}}
state = 0
for character c in input:
    state = t[state][c];
return state == 1 or state == 3
```

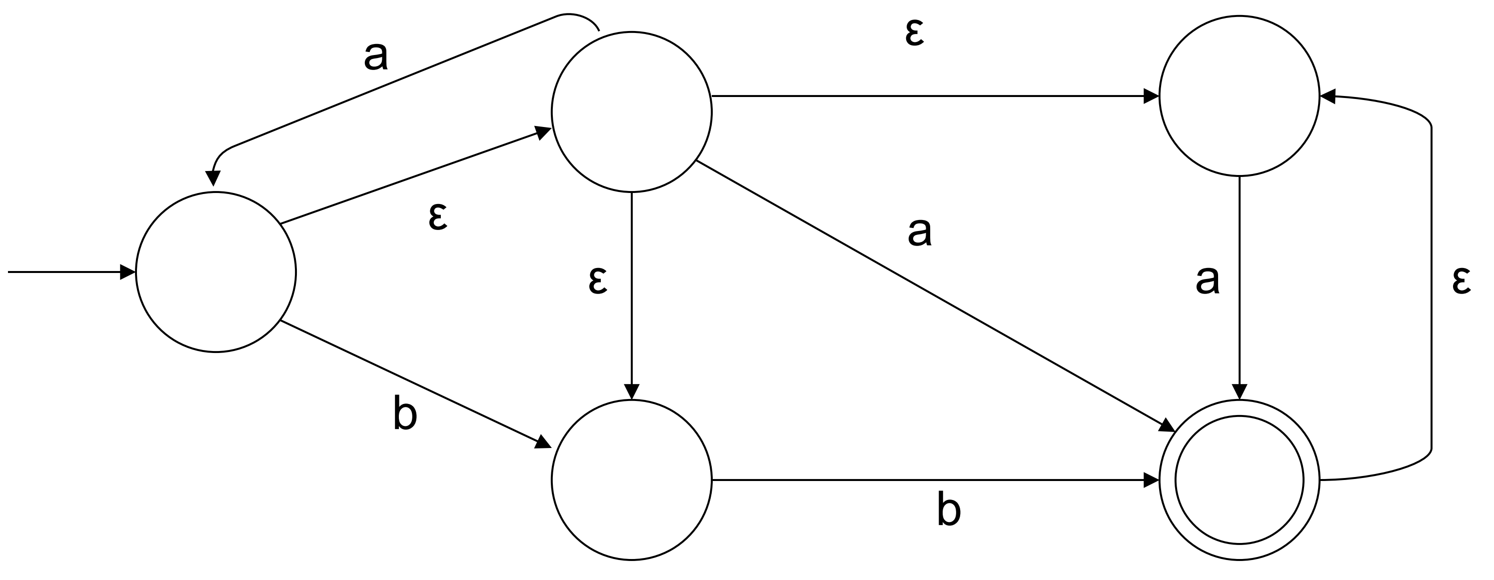


Exercise

- Exercise. What is the NFA for (0U1)*0? What is the DFA?

Exercise

- What is a DFA that accepts the same words as the follow NFA?



Big Picture

- The next thing will be to take a program as a string and
  - Cut it up into tokens
  - Build the abstract syntax tree
- The words (substrings) of a program fall into various categories. Each category is described by a regex. Words are recognized by DFA.
- Writing such a program is time consuming. There are language tools (lexers) for doing this.