Non-Deterministic Automata

What problem we will be solving. In the previous lecture note, we showed how we can transform automata for recognizing two languages A and B into an automaton for recognizing their union $A \cup B$ and intersection $A \cap B$. This construction was not easy.

There can be other operations with languages – we will denote them by A op B, where op is the symbol of the corresponding operation (such as $\cup$). For these operations, we would also want to transform automata for recognizing A and B into an automaton recognizing the language A op B. The corresponding constructions are even more difficult.

How can we make the description easier? A natural way to explain any complex algorithm is to represent it as a sequence of steps:

- first do this,
- then do that, etc.

This is exactly what we will do now. Instead of immediately presenting a complex automaton for recognizing the language A op B, we will do it in two steps:

- first, we describe a construction called a *non-deterministic automaton* (NDA) for recognizing A op B;
- then, we will show how to transform NDA into a finite automaton.

What is non-deterministic automaton. In a real finite automaton, for each state q and for each symbol s, we must describe where we should go if in the state q, we see the symbol s. In graphical terms, for each state q and for each symbols s, there is exactly one arrow coming out of the state q which has the symbol s on top. (This description, by the way, is what constitutes the function δ in the formal description of the finite automaton.)

In the non-deterministic case:

- for some state q and some symbol s, we can have no arrows going out of the state q with the symbol s on top; this means that we do not describe at all into what state the automaton should go;
- for some state q and some symbol s, we can have several arrows going out of q with the same symbol s on top; this means that upon seeing the symbol s in the state q, this automaton can go to different states;

- we can also have what is called *jumps*: arrows with ε on top (pronounced *epsilon*), meaning that we *can* "jump" to the next state without the need to see any symbol at all (can but do not have to).

In all these cases, the next state of the automaton is *not* uniquely *determined* by what symbol we see. This is where the term *non-deterministic* comes from.

Since the automaton is non-deterministic, for the same word, we may have different sequences of states that this automaton will follow as it reads the symbols. A word is *accepted* if there is a sequence leading to the final state – and it is OK if some other sequences do not lead to the final state.

We will have a few examples now, but first, an important notice:

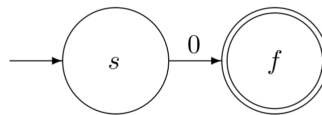
- a usual (deterministic) finite automaton is a *real* computational device, it is actively used in compiling;

- in contrast, a non-deterministic finite automaton is *not* a real computational device, it is an intermediate step in the design of the actual finite automaton.

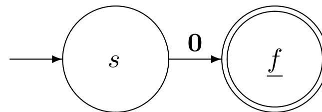
Note. Please note that while there may be several possible sequences of states, at any given moment of time, we can be in only one state: we cannot jump to two different states at the same time.

Simple examples. Let us assume that our alphabet consists of 0 and 1, and no other symbols are allowed. Under this assumption, we will present three non-deterministic automata that will illustrate all three features that we described above.

First example: when for some state q and some symbol s, there are no s-marked arrows coming out of the state q. Let us consider the following example:



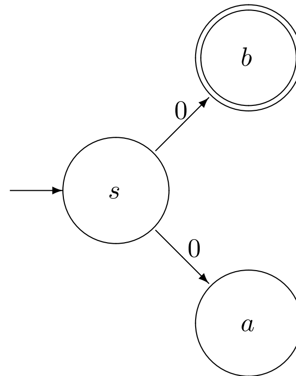
If we are in the start state s, and we see 0, we go to the final state, so 0 is accepted.



However, if we see 1 in the start state, we do not know where to go, since there is no arrow coming out of the state s that has 1 on top.

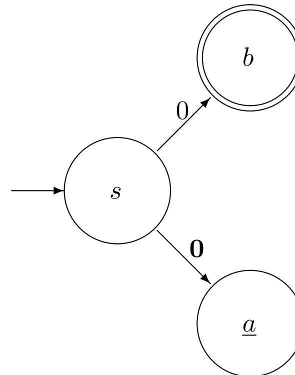
If we are in the final state, then no matter what symbol we see, we cannot go anywhere, since there are no arrows going out of the final state at all.

Second example: when we have several arrows with the same symbol going out of the same state.

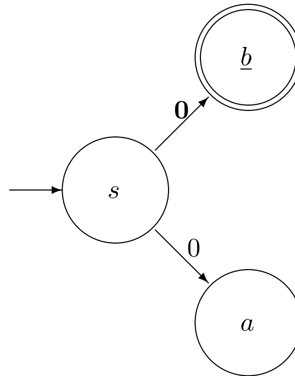


In this case, if in the starting state s, we see 0, we have two options:

- we can go to the state a which is not final:



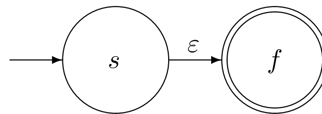
- or we can go to the state b which is final:



One of these two possible paths leads us to the final state, so the word 0 is *accepted*.

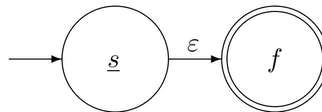
On the other hand, if we see 1, there is no way to go, since there are no arrows going from the start state s that have 1 on top. So, the word 1 is *not accepted*.

Third example: when we have a jump. Here is an example:

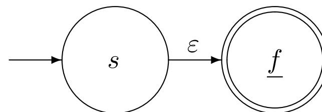


In this example, if we are in the start state s, then before we see any symbols we have two options:

- we can stay in this state, which is not final,



- or we can jump to the final state f.



In one of these options, we get to the final state. Thus, the empty string Λ can lead to the final state and is, therefore, accepted by this non-deterministic automaton.

What is the empty string: reminder. In Java, we can write

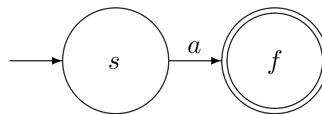
```
String a = "class";
```

then we will have a string consisting of 5 characters. We can also write

```
String a = "";
```

this will create a string with no characters at all; this string is known as the *empty string*.

Examples that will be our building blocks. For each symbol a, we can easily build a non-deterministic automaton that accepts only the word a and no other words, i.e., for which the corresponding language $L = \{a\}$ consists of a single 1-symbol word a:



One can easily check that the only way to get from the start state s to the final state f is to read the symbol a.

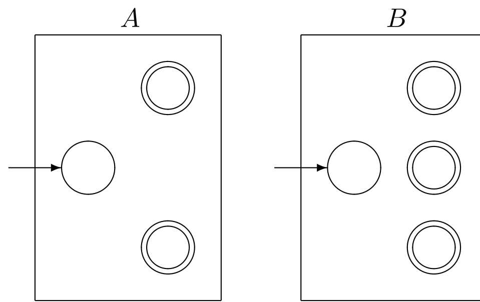
Comment about notations. For simplicity, this language will also be denoted simply by the letter a. So, e.g.:

- $\{a\}$ or a are notations for the language consisting of a single word a;
- $\{b\}$ or b are notations for the language consisting of a single word b;
- $\{0\}$ or 0 are notations for the language consisting of a single word 0, etc.

Practice. Design a similar non-deterministic finite automaton for the language $L = \{+\}$, i.e., an automaton that recognizes only one word: the word $+$.

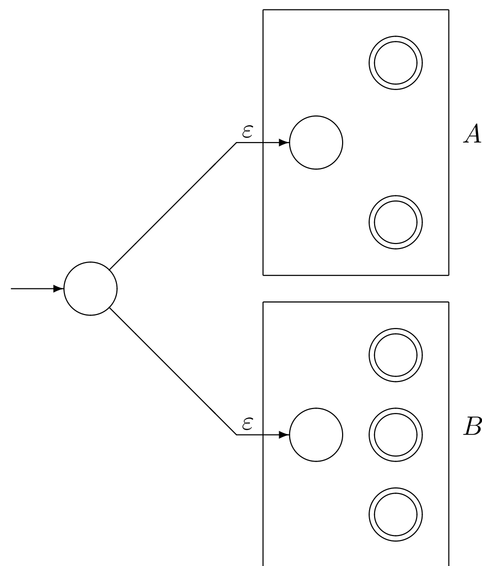
Hint: all you have to do is replace a in the above non-deterministic automaton with $+$.

How to design a non-deterministic automaton for the union. Suppose that we have two non-deterministic automata A and B:



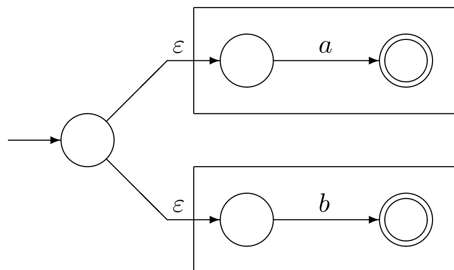
Then, to form a non-deterministic automaton for recognizing the union $A \cup B$, we need to do the following:

- make a new start state,
- add jumps from the new start state to both old start states.



Example. Let us apply this algorithm to automata for recognizing the languages $A = \{a\}$ and $B = \{b\}$:

Then, we get the following non-deterministic automaton for recognizing the language $A \cup B = \{a, b\}$:



Practice. Apply the same algorithm to two different automata: e.g., automaton A for recognizing unsigned integers and automaton B for recognizing binary sequences with odd number of symbols.

Note. We have shown how to use non-deterministic automata to describe the union. Since in the previous lecture, we dealt both with union and intersection, one would expect intersection as well; however, it is not clear how to describe intersection in terms of non-deterministic automata.

Concatenation. In the description of programming languages, we often use the words "followed by". For example:

- a signed integer is a sign *followed by* an integer;
- a floating point number is an integer *followed by* the period *followed by* an unsigned integer, etc.

In all these cases, we place one word after another; this is called *concatenation.*

In theory of computations, if we write a word w' after the word w, we simply write ww'. In Java, concatenation is described by the plus sign +: if we write

```
System.out.println("Vla" + "dik");
```

the system will print the word

```
Vladik
```

For two languages A and B, concatenation AB is the set of all the words that can be obtained if we first place a word from the language A, and then a word from the language B:

$$AB \stackrel{\text{def}}{=} \{ab : a \in A, b \in B\}.$$

Example. Let $A = \{\Lambda, +, -\}$ and let B be the set of all binary sequences of length 1 or 2:

$$B = \{0, 1, 00, 01, 10, 11\}.$$

Then:

- if we first place the empty string Λ, this will not change what we write after that, so we will get the words

$$0, 1, 00, 01, 10, 11;$$

- if we first place the plus sign, then we will get the words:

$$+0, +1, +00, +01, +10, +11;$$

- finally, if we first place the minus sign, then we will get the words:

$$-0, -1, -00, -01, -10, -11.$$

Thus, in this case,

$$AB = \{0, 1, 00, 01, 10, 11, +0, +1, +00, +01, +10, +11, -0, -1, -00, -01, -10, -11\}.$$

Important notice. Concatenation is *not* commutative:

```
System.out.println("Vla" + "dik");
```

print the word

```
Vladik
```

but if we change the order:

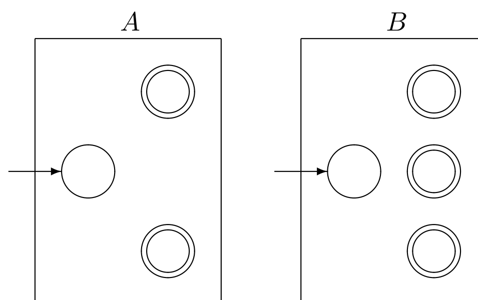
```
System.out.println("dik" + "Vla");
```

the system will print a different word

```
dikVla
```

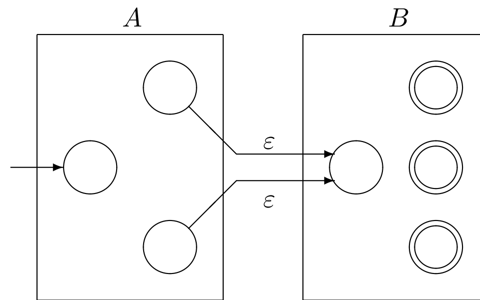
Practice. Write down what will the set AB looks like if $A = \{red, blue\}$ and $B = \{cat, dog\}$.

How to recognize the concatenation. Suppose that we have two automata that recognize two languages A and B:

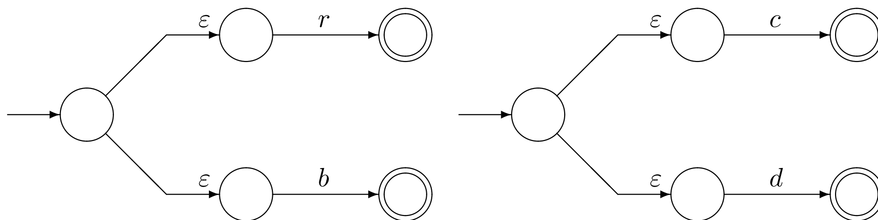


Then, to recognize the concatenation AB, we first need to recognize the first part as belonging to A, i.e., get to the final state of the A-automaton. However, this is not enough: after that, we need to recognize the remaining part of the input as belonging to B. So:

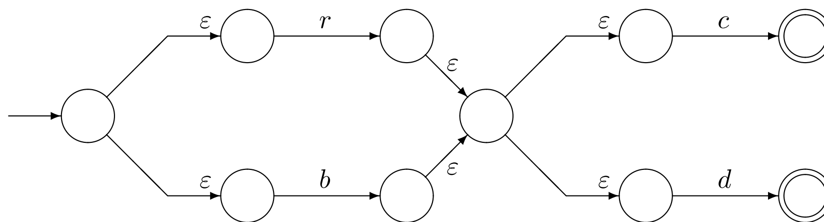
- final states of A are no longer final, and
- we add jumps from each previously final state of A to the starting state of B.



Example. Let us take two non-deterministic automata for recognizing the sets $A = \{r, b\} = \{r\} \cup \{b\}$ and $B = \{c, d\} = \{c\} \cup \{d\}$. (Feel free to interpret these letters as *red*, *blue*, *cat*, and *dog*):



By using the above algorithm – i.e., by making previously final states of A not final and adding jumps from these states to the starting state of B – we get the following non-deterministic automaton for the language $AB = \{rc, rd, bc, bd\}$:



Practice. Try this algorithm on other pairs of automata.

Concatenation and union: idea. What if we have both concatenation and union? In this case, we start from the original languages and apply both algorithms again and again.

Example. Suppose that we have a language $01 \cup 23$. The notation AB for concatenation is similar to the usual notation for the product, so:

- just like in an arithmetic expression $2 + 3 \cdot 4$, multiplication is performed first,
- here, concatenation is performed first – unless it is in parentheses.

If we have $A \cup B \cup C$, then we first design an automaton for $A \cup B$, then combine it with the automaton for C.

Similar to arithmetic expressions, if we want to change the order of operations, we need to use parentheses; e.g.:

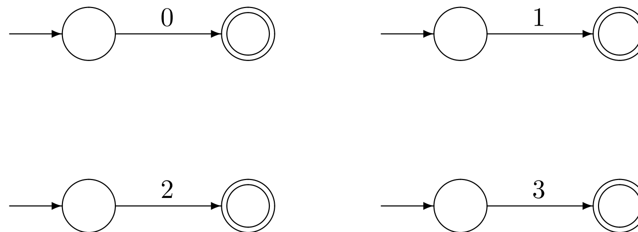
- $0(1 \cup 2)3$ means that we first perform the union $1 \cup 2$, and only then perform concatenation;
- $A \cup (B \cup C)$ means that we first design an automaton for $B \cup C$, and only then combine it with an automaton for A.

It should be noted that while the sets $(A \cup B) \cup C$ and $A \cup (B \cup C)$ are the same, the resulting automata may be different – so we will get two different automata for recognizing the same language.

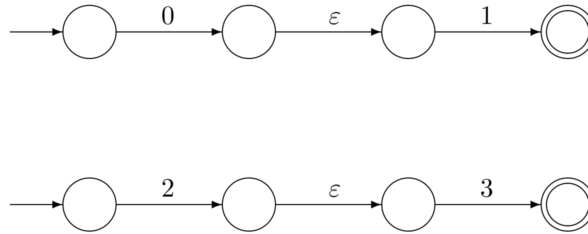
In the above example:

- first, we perform concatenations, i.e., design automata for 01 and 23;
- then, we apply the union operation to transform automata for 01 and 23 into an automaton for the union $01 \cup 23$.

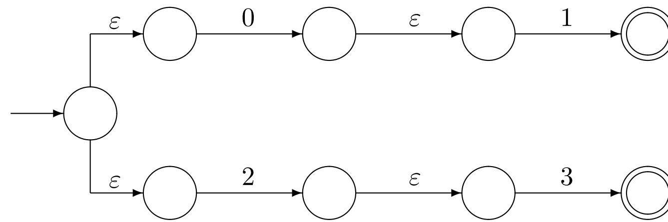
We start with automata for recognizing 0, 1, 2, and 3:



Then, we apply the concatenation algorithm to design automata for recognizing languages 01 and 23:



Then, we apply the union algorithm to get the automaton for recognizing the union $01 \cup 23$:



Practice. Try some other expressions, e.g., $0 \cup 0(1 \cup 2)$.

Kleene star. Some descriptions of programming language use plural. For example, a signed integer can be described as sign followed by digits. A valid variable name is a letter followed by letters, digits, and underscore. This plural is formally described as a *Kleene star*

$$A^* = \{\Lambda; a \in A; a_1 a_2, a_i \in A; a_1 a_2 a_3, a_i \in A, \ldots\}.$$

In other words, we can have an empty string, we can have a word from A, we can have a concatenation of two words from A, three, etc.

For example, if $A = \{b\}$, then A^* is the set of all the words that have only letters b:

$$A^* = \{\Lambda, b, bb, bbb, bbbb, \ldots\}.$$

If $A = \{0, 1\}$, then A^* is the set of all binary sequences:

$$A^* = \{\Lambda, 0, 1, 00, 01, 10, 11, 000, \ldots\}.$$

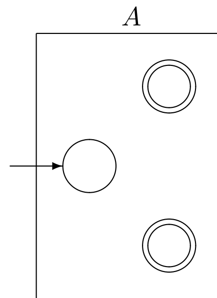
If $A = \{South, West\}$, then

$$A^* = \{\Lambda, South, West, SouthSouth, SouthWest, WestSouth, WestWest,$$

$$SouthSouthSouth, \ldots\}.$$

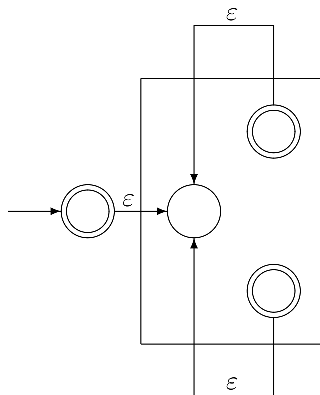
Practice. Describe the Kleene star of some other language.

How to recognize a Kleene star. Suppose that we have an automaton for recognizing a language A:

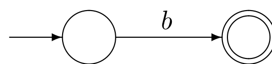


To form an automaton that recognizes the language A^*, we need to do the following:

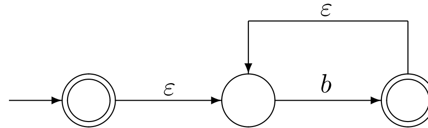
- add a new starting state which is also final;
- add jump from the new starting state to the old starting state; and
- add jumps from the old final states to the old starting state.



Example. Let us start with the language $A = \{b\}$. For this language, the automaton has the following form:



By applying the above algorithm, we get the following automaton for recognizing the language $A^* = \{\Lambda, b, bb, \ldots\}$:

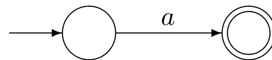


Practice: try applying this algorithm to some other languages.

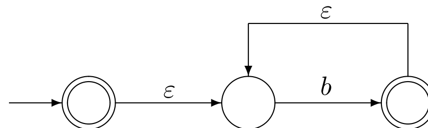
Union, concatenation, and Kleene star. We can have expressions containing union, concatenation, and Kleene star. In these expressions, Kleene star is performed first – since it is a unary operation, and unary operations are usually performed first: e.g., ab^2 means that we first compute b^2.

So, $a \cup b^*$ means that we first find the set b^*, and then find the union of this set and the set a. If we want to describe the Kleene star of the union $a \cup b$, we describe it as $(a \cup b)^*$.

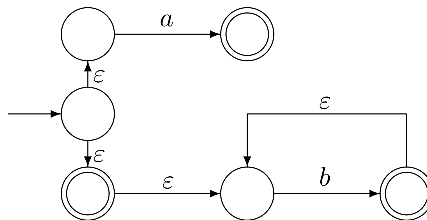
How can we recognize the language $a \cup b^*$? We have an automaton for recognizing the language a:



We also have an automaton for recognizing the language b^*:

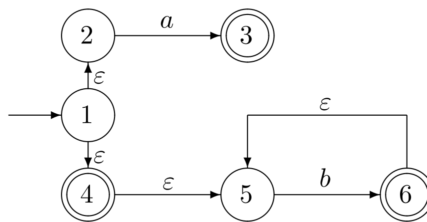


To these two automata, we apply the union-algorithm. As a result, we get the following automaton for recognizing the language $a \cup b^*$:

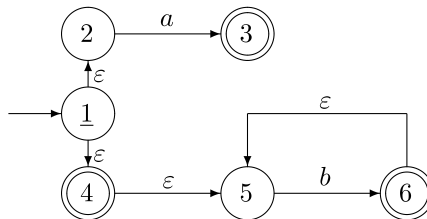


How to transform a non-deterministic finite automaton into a deterministic one? Let us recall that non-deterministic finite automata are *not* real computational devices, they are auxiliary constructions that still need to be transformed into real (deterministic) finite automata. How can we do it? Let us illustrate this on the example of the above non-deterministic finite automaton.

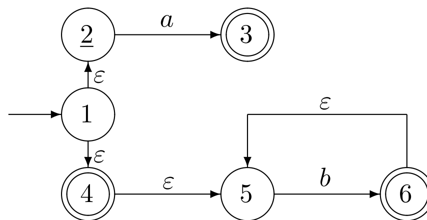
To make our analysis clearer, let us number the state of this non-deterministic automaton. How we number them, does not really matter. It is usually convenient to assign No. 1 to the starting state, but how we assign numbers after that will not affect the resulting deterministic automaton. If we assign top-to-bottom and left-to-right within each horizontal layer, we get the following numbering:

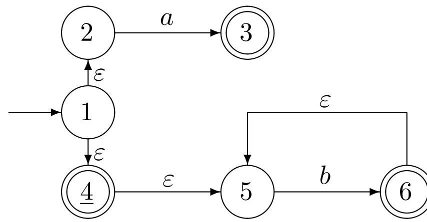


Let us analyze what will happen if we feed a word symbol-by-symbol to this automaton. In the beginning, we are in the starting state 1.

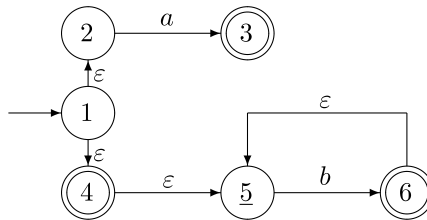


We can stay in this state, or we can jump to the state 2 or to the state 4.



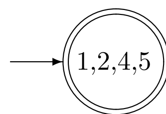


In the state 2, we cannot jump anywhere, but from state 4, we can jump to state 5:



From state 5, we cannot jump anywhere. So, before we start reading symbols, we can end up in one of the four states: 1, 2, 4, and 5. The set $\{1, 2, 4, 5\}$ consisting of these four possible states of the NDA will be the starting state of our deterministic finite automaton.

Is this state final? A non-deterministic automaton accepts a word if one of the resulting states is final. In our case, one of these four states – namely, the state 4 – is final, so the state $\{1, 2, 4, 5\}$ is a final state for the deterministic finite automaton.



For a deterministic finite automaton, for each state and for each symbol, we need to determine what will be the next state.

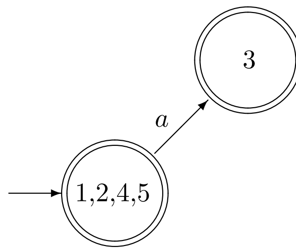
If we are in the state $\{1, 2, 4, 5\}$ and we see a symbol a, where can we go?

- From the state 1, there are no arrows with a on top;

- from state 2, there is an arrow with a on top, it leads to state 3;
- for states 4 and 5, there are no arrows with a on top.

Thus, we can go to state 3. From state 3, there are no jumps, so the only state that we can reach is the state 3.

So, if we are in the starting state $\{1, 2, 4, 5\}$, and we see a symbol a, we can end up only in one state: the state 3. The set of possible states is thus the set $\{3\}$ consisting of one state 3. The state 3 is a final state of the non-deterministic finite automaton, so it is final for the deterministic finite automaton as well:

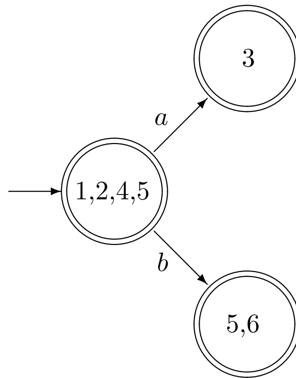


What if in the state $\{1, 2, 4, 5\}$, we see the symbol b?

- From the states 1, 2, and 4, there are no arrows with b on top;
- from state 5, there is an arrow with b on top, it leads to state 6.

Thus, we can go to state 6. From state 6, we can jump to state 5, and from state 5, there are no jumps. So the only states that we can reach are the states 5 and 6.

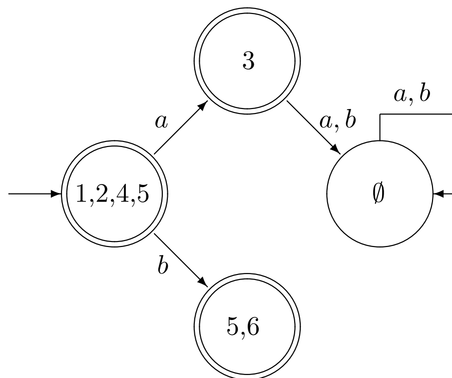
So, if we are in the starting state $\{1, 2, 4, 5\}$, and we see a symbol b, we can end up in state 5 or state 6. The set of possible states is thus the set $\{5, 6\}$. The state 6 is a final state of the non-deterministic finite automaton, so the state $\{5, 6\}$ containing 6 is final for the deterministic finite automaton:



For the state $\{1, 2, 4, 5\}$, we considered all possible symbols: a and b. Let us now consider the state 3. From this state, there are no arrows at all, so no matter what next symbol we see, whether it is a or b, we cannot go to any state. This means that the set of possible states is the *empty set*. The empty set is usually denoted by $\emptyset$ or by $\{\}$.

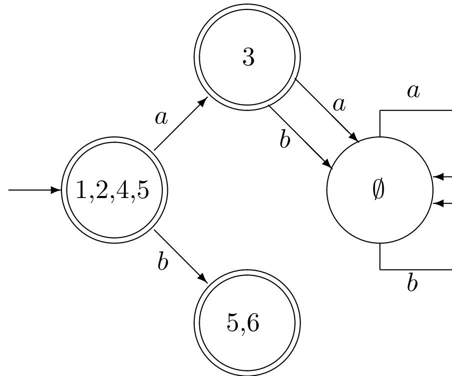
Comment about notation. Note that the usual mathematical sign for the empty set is a crossed 0. The cross is important: the empty set is *never* described by simply a 0: 0 means a language which has exactly one word 0, *not* an empty set.

Back to constructing a deterministic finite automaton. Of course, if we are in an empty set (i.e., no states are possible), we cannot go anywhere, so whatever we see in the empty set state will bring us back to the empty set. In other words, here, the empty set is a sink state.



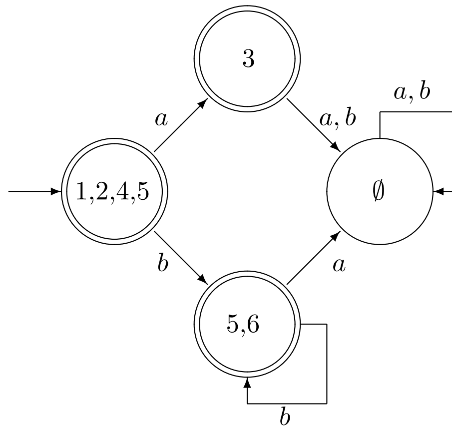
Another comment about notation. Instead of having two symbols a and b on a single arrow, we could have two arrows:

- one arrow corresponding to letter a and
- another arrow corresponding to letter b.



Back to constructing a deterministic finite automaton. Finally, let us consider the state $\{5, 6\}$. Neither state 5 nor state 6 have any arrows with a on top. So, if in this state, we see a symbol a, we cannot go anywhere – so the set of possible states is the empty set.

If we see symbol b, then from state 5, there is an arrow with b on top, it leads to state 6. From state 6, we can jump to state 5. Thus, the set of possible states after seeing the symbol b is the set $\{5, 6\}$:

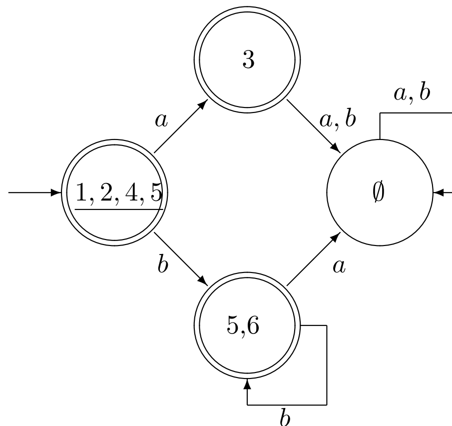


Now, for every state and for every symbol, we have a description of where to go, so we have a deterministic finite automaton.

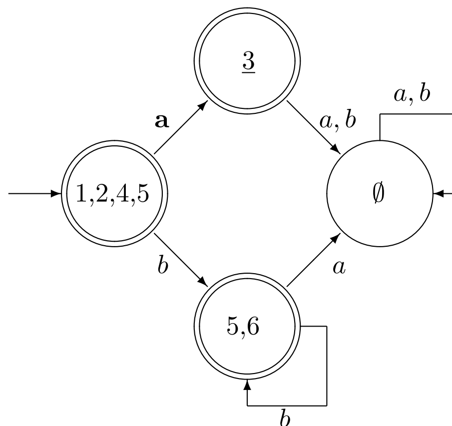
Let us show that this automaton indeed recognizes the language with which we started: the language

$$a \cup b^* = \{a\} \cup \{\Lambda, b, bb, \ldots\} = \{\Lambda, a, b, bb, \ldots\}.$$

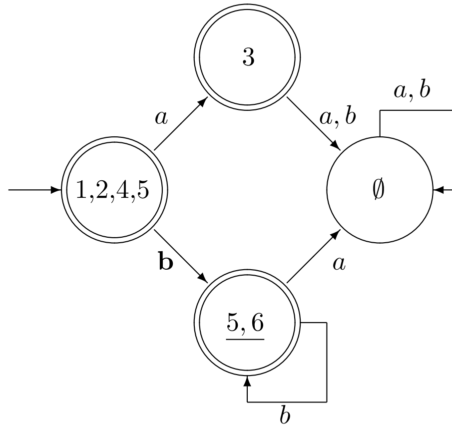
For this automaton, the starting state is also final. So, if the input is the empty string, with no symbols to read, we stay in the starting state. Thus, the empty string Λ is accepted.



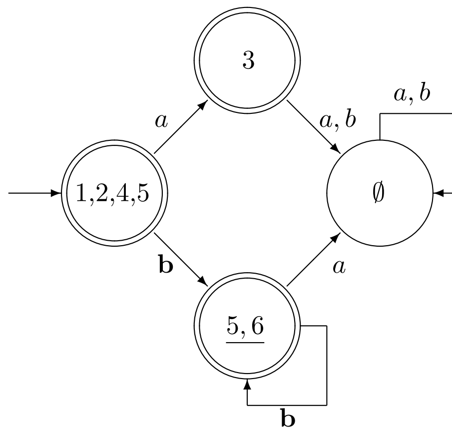
The only way to get to the final state $\{3\}$ is to read the symbol a. After this, any other symbol will bring us to a non-final state $\emptyset$. Thus, the only word that can be accepted this way is the word consisting of a single symbol a.



To get to the final state $\{5, 6\}$, we need to read a letter b.



This can be followed by other letters b, as many as we want (but if read letter a, to go to a non-final stare and stay there):



So, the words that can be accepted are: Λ, a, and the words b, bb, $\ldots$, consisting of letters b only. This is indeed the language that we want to recognize.

Practice: try the same transition for some other non-deterministic finite automaton.

Note. States of the deterministic automata are subsets of the set $\{1, 2, 3, 4, 5, 6\}$ of all the states of the non-deterministic automaton. Good news is that we do not need *all* subsets:

- in the 6-element set, there are $2^6 = 64$ subsets,
- but in our deterministic automaton, we used only four of them.

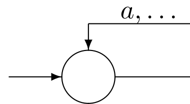
Summarizing. Anything that can be obtained from symbols by using union, concatenation, and the Kleene star is known as a *regular expression.* What we have shown in this lecture is that every regular expression can be represented by a deterministic finite automaton.

Note. Regular expressions can also be two special symbols:

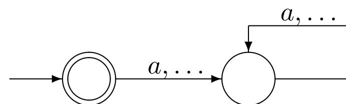
- the symbol $\emptyset$ (or $\{\}$) for the empty set, and
- the symbol Λ means the set that contains only one word – the empty string.

Both sets can easily be represented by automata:

- the empty set can be represented by an automaton in which no state is final:



- and the empty string can be represented by an automaton in which only the starting state is final, and any symbol moves us away from this state:



What we will do next. In the following lecture, we will show that, vice versa, every language recognized by a deterministic finite automaton can be represented as a regular expression.