Luku 2

Regular languages and finite automata

Tuumasta toimeen...
No niin kone! Käypäs toimeen. Listassa on kaikki merkkijonot, joita koodissa saa olla.
OK
ERR
if...
else...
while...

- e.g. What is legal for variables, constants, different datatypes and so on in the programming languages? How to describe that syntax?

- The problem can be solved by *regular languages* and *finite automata*

- in UNIX ocommand `grep` to search for given patterns in the text. (grep="Global search for Regular Expression and Print")

2.1 Regular expressions and regular languages

- E.g. accept all strings which contain word ”cat” i.e. the strings are of form

  [0 or more letters]*cat*[0 or more letters]

- E.g. Search strings which are of form
  [$x$street or $x$road][*number*] [possible letter]
  [possible flat number][postcode][city]

- How to represent in a compact way all legal strings (i.e. how to describe the language that the recognizing program accepts?)

- Let’s define three operators for combining languages: Let A and B be langauges in alphabet Σ. Then

  – *Union* of A and B is language

  $$A \cup B = \{x \in \Sigma^* \mid x \in A \text{ o r} x \in B\}$$

  – *Product* of A and B

  $$AB = \{xy \in \Sigma^* \mid x \in A,\ y \in B\}$$

  – *Powers* of A A^k, $k \geq 0$, are defined:

  $$\begin{cases} A^0 &= \{\epsilon\}, \\ A^k &= AA^{k-1} \\ &= \{x_1 \dots x_k \mid x_i \in A \quad \forall i = 1, \dots, k\} \qquad (k \geq 1) \end{cases}$$

  – *Closure* of A

  $$\begin{aligned} A^* &= \bigcup_{k=0}^{\infty} A^k \\ &= \{x_1 \dots x_k \mid k \geq 0,\ x_i \in A \quad \forall i = 1, \dots, k\} \end{aligned}$$

- *Definition Regular expression* in Σ are defined by rules:

  (i) $\boldsymbol{\emptyset}$ and $\boldsymbol{\epsilon}$ are regular expressions of Σ;

  (ii) $\boldsymbol{a}$ is regular expression of Σ for all $a \in \Sigma$;

  (iii) If r and s are regular expression in Σ,
  then $(r \cup s)$, (rs) and $r^{\boldsymbol{*}}$ are regular expressions in Σ;

  (iv) Other regular expressions in Σ don’t exist.

- Each regular expression of Σ, r, *describes* a language $L(r)$:

  (i) $L(\boldsymbol{\emptyset}) = \emptyset$;

  (ii) $L(\boldsymbol{\epsilon}) = \{\epsilon\}$;

  (iii) $L(\boldsymbol{a}) = \{a\}$ for all $a \in \Sigma$;

  (iv) $L((r \cup s)) = L(r) \cup L(s)$;

  (v) $L((rs)) = L(r)L(s)$;

  (vi) $L(r^{\boldsymbol{*}}) = (L(r))^*$

- E.g. In alphabet $\{a, b\}$:

$$r_1 = \boldsymbol{((ab)b)}, \quad r_2 = \boldsymbol{(ab)^*},$$

$$r_3 = \boldsymbol{(ab^*)}, \quad r_4 = \boldsymbol{(a(b \cup (bb)))^*}.$$

  The corresponding languages:

$$\begin{aligned}
L(r_1) &= (\{a\}\{b\})\{b\} = \{ab\}\{b\} = \{abb\};\\
L(r_2) &= \{ab\}^* = \{\epsilon, ab, abab, ababab, \dots\}\\
&= \{(ab)^i \mid i \geq 0\};\\
L(r_3) &= \{a\}(\{b\})^* = \{a, ab, abb, abbb, \dots\}\\
&= \{ab^i \mid i \geq 0\};\\
L(r_4) &= (\{a\}\{b, bb\})^* = \{ab, abb\}^*\\
&= \{\epsilon, ab, abb, abab, ababb, \dots\}\\
&= \{x \in \{a,b\}^* \mid \text{ each } a \text{ in } x\\
&\qquad \text{is followed by 1 or 2 } b\text{'s }\}
\end{aligned}$$

- Rules for dropping paranthesis:

  - Priority of operators:

$$\boldsymbol{*} \quad \succ \quad \cdot \quad \succ \quad \cup$$

  - Assosiativity of union and product operations:

$$\begin{aligned}
L(((r \cup s) \cup t)) &= L((r \cup (s \cup t)))\\
L(((rs)t)) &= L((r(st)))
\end{aligned}$$

- We can use common letters if there is no danger of confusion

  Simplier:

$$r_1 = abb, \quad r_2 = (ab)^*, \quad r_3 = ab^*, \quad r_4 = (a(b \cup bb))^*$$

- *Definition:* Language is *regular*, if it can be described by a regular expression.
- E.g. Let alphabet $\Sigma = \{a, b, c, ..., \}$. Let's accept strings of form

$$l^*\text{cat}l^*,$$

  in which l is abreviation for $l = (\text{a} \cup \text{b} \cup ... \cup \text{ö})$ (i.e. $l^* \in \Sigma^*$)

- E.g. unsigned floating point numbers in C (float, double, long double):
  - (integer part).(desimal part) (e or E) [+ or −] (exponent) [suffix]
  - integer part and desimal part consist of digits
  - either integer or desimal part may be missing (but not both)
  - either (i) desimal point or (ii) (e or E) and exponent can be missing (but not both)
  - suffix: F or f: float, L or l: long double, otherwise double
- The recognizing language for unsigned floating point (without suffixes): number $= (d^+.d^* \cup .d^+)(\epsilon \cup ((e \cup E)(+ \cup - \cup \epsilon)d^+)) \cup$ $d^+(e \cup E)(+ \cup - \cup \epsilon)d^+$
- For example strings 12., .12, 1.2, 1.2E3, 1.2e3, 1.2E-3,1E2, 1e23 belong to the language

2.1.1 Simplifying regular expressions

- Often many equivalent expressions, e.g.:

$$\begin{aligned} \Sigma^* &= L((a \cup b)^*) \\ &= L((a^*b^*)^*) \\ &= L(a^*b^* \cup (a \cup b)^*ba(a \cup b)^*). \end{aligned}$$

- *Definition:* Regular expressions r and s are *equivalent*, mark $r = s$, if $L(r) = L(s)$

- Also mark $r \subseteq s$, if $L(r) \subseteq L(s)$
- simplifying the expression= defining the "simplest" equivalent expression
- Notice! We often mark $r^+ = rr^* = r^*r$

2.1.2 Simplifying rules

$$\begin{aligned}
r \cup r &= r \text{ (but } rr \neq r, \text{ when } r \neq \emptyset, \epsilon) \\
r \cup (s \cup t) &= (r \cup s) \cup t \\
r(st) &= (rs)t \\
r \cup s &= s \cup r \\
r(s \cup t) &= rs \cup rt \\
(r \cup s)t &= rt \cup st \\
\emptyset^* &= \epsilon \\
\emptyset r &= \emptyset \text{ (but } \emptyset \cup r = r) \\
\epsilon r &= r \text{ (but } \epsilon \cup r \neq r, \text{ when } r \neq \epsilon) \\
r^* &= r^*r \cup \epsilon = r^+ \cup \epsilon \\
r^* &= (r \cup \epsilon)^* \\
(r^*)^* &= r^*
\end{aligned}$$

- Also holds:
  If $r = rs \cup t$, then $r = ts^*$, when $\epsilon \notin L(s)$
- $L(r) = L(s) \Leftrightarrow L(r) \subseteq L(s) \wedge L(s) \subseteq L(r)$ i.e. $r = s \Leftrightarrow r \subseteq s \wedge s \subseteq r$
- E.g.:
  1. $(a \cup b) \subseteq (a^*b^*) \Rightarrow (a \cup b)^* \subseteq (a^*b^*)^*$
  2. $((a^*b^*)^*) \subseteq a^*b^* \cup (a \cup b)^*ba(a \cup b)^*)$:
     - if of form a^*b^*, then obvious
     - otherwise contains substring ba
  3. $a^*b^* \cup (a \cup b)^*ba(a \cup b)^*) \subseteq (a \cup b)^*$, because $(a \cup b)^*$ describes all strings of Σ
- We can also prove (proof omitted): If L and M are regular languages, then also

1. $L \cap M$
2. $\overline{L} = \Sigma^* \setminus L$
3. $L^R = \{w^R | w \in L\}$

are regular

2.2 Finite automata

- Problem: A coffeemachine, which doesn't give change, accepts coins of 50 cents and one euro. The minimum tax is 2 euros, What kind of input strings does the machine accept?
- Legal input strings e.g. (as cents):
  50 + 50 + 50 + 50
  100 + 100
  50 + 100 +100
  100 + 50 + 50 + 100
- i.e. input strings are of form

```
1 euro +  1 euro +
[0 or more 50 cents or 1 euro coins]
```

or

```
1 euro +  50 cents +
[1 or more 50 cents or 1 euro coins]
```

or

```
50 cents + 1 euro +
[1 or more 50 cents or 1 euro coins]
```

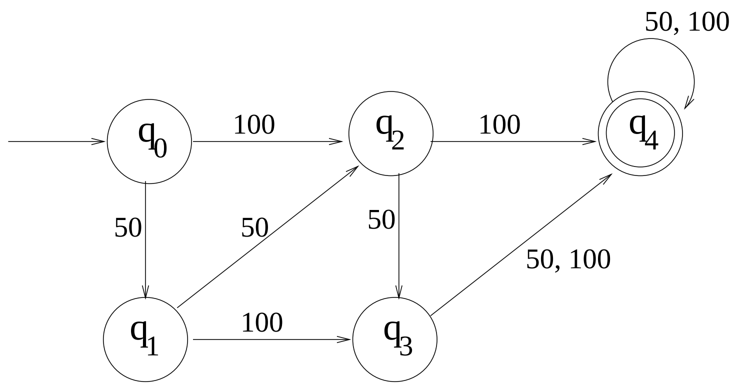
or

```
50 cents +  50 cents + 1 euro +
[0 or more 50 cents or 1 euro coins]
```

or

```
50 cents + 50 cents + 50 cents +
[1 or more 50 cents or 1 euro coins]
```

- The coffeemachine can be described as *a finite automaton*
- input of automaton: 50 cents and 1 euro coins
- the automaton accepts an ”input string”, if the sum of the coins in it is at least 2 euros
- Automaton can be represented as *a transition diagram*

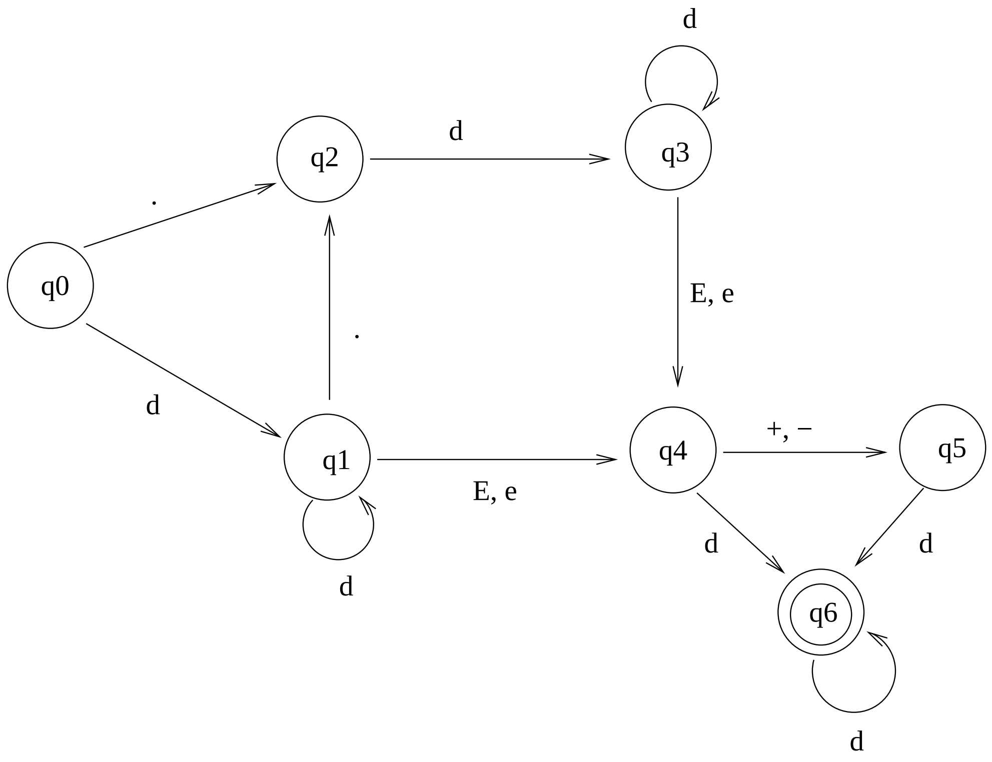


2.2.1 Representations of finite automaton

- transition diagram
- transition matrix

	50 snt	1 euro
$\rightarrow q_0$	q_1	q_2
q_1	q_2	q_3
q_2	q_3	q_4
q_3	q_4	q_4
$\leftarrow q_4$	q_4	q_4

- E.g. unsigned floating point numbers in C:



– Transition matrix:

		d	.	E, e	+, −
$\rightarrow$	q_0	q_1	q_2		
	q_1	q_1	q_3	q_4	
	q_2	q_3			
$\leftarrow$	q_3	q_3		q_4	
	q_4	q_6			q_5
	q_5	q_6			
$\leftarrow$	q_6	q_6			

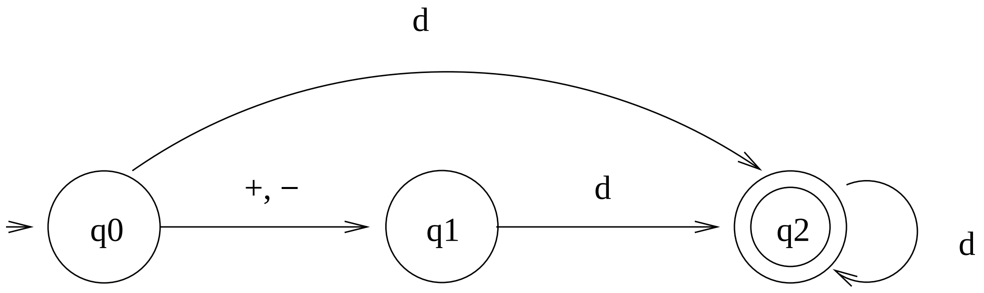
Here $\mathtt{d} = \{0, 1, \ldots, 9\}$. The missing positions of the matrix correspond "Error" states, which are usually not drawn.

– As a program:

```
int IsDigit(char c); /* returns 1, if c is digit, 0 otherwise */

int q = 0
char c while ( (c = fgetc(stdin))!=EOF )
{
   switch ( q )
   {
      case 0: if (IsDigit(c) ) q = 1;
              else if (c=='.') q = 2 else q = 99;
              break;
      case 1: if (IsDigit(c) ) q = 1;
              else if (c=='.') q=3;
              else if (c=='e' || c=='E') q=4; else q=99;
              break;
      case 2: if (IsDigit(c)) q=3; else q=99;
              break;
      case 3: if (IsDigit(c)) q=3;
              else if (c=='e' || c=='E') q = 4 else q = 99;
              break;
      case 4: if (IsDigit(c)) q=6;
              else if (c=='+' || c=='-') q = 5 else q = 99;
              break;
      case 5: if (IsDigit(c)) q=6; else q = 99;
              break;
      case 6: if (IsDigit(c)) q=6; else q = 99;
              break;
      case 99: break;
   }
}
if ( q == 3 || q == 6 ) printf("OK!");
else printf("Error");
```

- We can also add semantic functions into the program based on finite automaton
- E.g. Recognizing a signed integer.



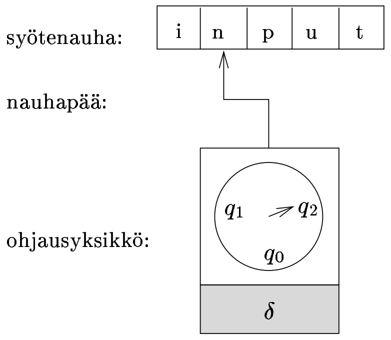
- Corresponding program, which also evaluates the value of number:

```
int IsDigit(char c); /* returns 1, if c is digit, 0 otherwise */

int $q = 0$
char $c$ int $sign = 1$; int $val = 0$; while ( ($c$ = fgetc(stdin))!=EOF )
{
   switch ( $q$ )
   {
      case 0: if ($c$=='+' || $c$=='-') {
              $q$=1;
              if ($c$=='-') $sign = -1$;
              }
              else if (IsDigit($c$) {
              $q$=2;
              $val = c -' 0'$;
              }
              break;
      case 1: if (IsDigit($c$) {
              $q$=2;
              $val = c -' 0'$;
              }
              else $q$=99;
              break;
      case 2: if (IsDigit($c$)) {
              $q$=2;
              $val = 10 * val + (c -' 0')$;
              }
              else $q$=99;
              break;
```

```
        case 99: break;
    }
}
if ( q == 2) printf("The value of the number is %d", sgn * val);
else printf("Errourness number");
```

2.2.2 Formal definition



- Finite automaton M
  - finite *control device*, which is defined by *transition function* δ
  - *input tape*, which is divided into cells
  - *reading head*, which is always positioned to some input charater
- Automaton stops when the last input character is read. If the control device is then in a favourable state the automaton *accepts* the input, otherwise it is *rejected*
- The automaton *recognizes* the language, which consist of all accepted strings
- *Definition: Finite automaton* is a quintuple

  $$M = (Q, \Sigma, \delta, q_0, F),$$

  in which
  - Q is finite set of *states*
  - Σ is *input alphabet*;
  - $\delta : Q \times \Sigma \to Q$ is the *transition function*;

  - $q_0 \in Q$ is the *initial state*;
  - $F \subseteq Q$ is the set of *(favorable) final* states.

- E.g. the formal representation of the real number automaton:

$$\begin{aligned} M &= (\{q_0, \dots, q_6, error\}, \{\texttt{0,1, ... ,9,.,E,e,+,-}\}, \\ &\quad \delta, q_0, \{q_3, q_6\}), \end{aligned}$$

  in which δ is like earlier in the matrix; e.g.

$$\delta(q_0, \texttt{0}) = \delta(q_0, \texttt{1}) = \dots = \delta(q_0, \texttt{9}) = q_1,$$

$$\delta(q_0, .) = q_2, \quad \delta(q_0, E) = error, \quad \delta(q_1, \texttt{E}) = q_4 \quad \text{jne.}$$

- *Configuration* is pair $(q, w) \in Q \times \Sigma^*$
  - *initial configuration* x is pair (q_0, x)
  - q is current configuration w is the unread part of input string

- Configuration (q, w) *leads directly* to configuartion (q', w'), mark

$$(q, w) \underset{M}{\vdash} (q', w'),$$

  if $w = aw'$ $(a \in \Sigma)$ and $q' = \delta(q, a)$.
  Configuration (q', w') is an *immediate successor* of (q, w)

- (q, w) *leads* to (q', w') i.e. configuration (q', w') is a *successor* of (q, w), mark

$$(q, w) \underset{M}{\vdash}^* (q', w'),$$

  if there exists configuration queue $(q_0, w_0), (q_1, w_1), \dots, (q_n, w_n)$, $n \geq 0$, such that

$$(q, w) = (q_0, w_0) \underset{M}{\vdash} (q_1, w_1) \underset{M}{\vdash} \cdots \underset{M}{\vdash} (q_n, w_n) = (q', w').$$

  Special case: $n = 0$, $(q, w) \underset{M}{\vdash}^* (q, w)$ for any (q, w)

- Automaton M *accepts* string $x \in \Sigma^*$, if

$$(q_0, x) \underset{M}{\vdash}^* (q_f, \epsilon) \qquad \text{for some } q_f \in F;$$

  otherwise M *rejects* x.

- *Definition:* Automaton M *recognizes language*

$$L(M) = \{x \in \Sigma^* \mid (q_0, x) \underset{M}{\vdash}^* (q_f, \epsilon) \quad \text{for some } q_f \in F\}$$

- E.g. string "`0.25E2`":

$$\begin{array}{lllll} (q_0, \texttt{0.25E2}) & \vdash & (q_1, \texttt{.25E2}) & \vdash & (q_3, \texttt{25E2}) \\ & \vdash & (q_3, \texttt{5E2}) & \vdash & (q_3, \texttt{E2}) \\ & \vdash & (q_4, \texttt{2}) & \vdash & (q_6, \epsilon). \end{array}$$

  Because $q_6 \in F = \{q_3, q_6\}$, then $\texttt{0.25E2} \in L(M)$

2.3 Minimizing automaton

- Two automata, which recognize exactly the same language, are *equivalent*
- A finite automaton is *minimal* if it has minimum number of states compared to other equivalent automata
- An automaton, which has more states than the minimal equivalent automaton, is *redundant*
- Algorithms, which construct automata, don't always produce minimal automata
- It is easier to "read" a minimal automaton than redundant automaton
- It's useless to save additional states
- It's more efficient to process on a minimal automaton

2.3.1 Some concepts

- Let's define for M an augmented transition function δ^*, which can have a string as its parameter:
  if $q \in Q$, $x \in \Sigma^*$, then

$$\delta^*(q, x) = \text{ such } q' \in Q, \text{ that } (q, x) \underset{M}{\vdash}^* (q', \epsilon)$$

- Equivalence between states: states of M, q and q', are *equivalent*, mark

  $$q \equiv q',$$

  if for all $x \in \Sigma^*$ holds

  $$\delta^*(q,x) \in F \quad \text{if and only if} \quad \delta^*(q',x) \in F$$

  (i.e. if automaton accepts after q and q' exactly same strings)

- *k-equivalence*: states q and q' are k-equivalent, mark

  $$q \overset{k}{\equiv} q',$$

  if for all $x \in \Sigma^*$, $|x| \leq k$, holds

  $$\delta^*(q,x) \in F \quad \text{if and only if} \quad \delta^*(q',x) \in F$$

  (i.e. if string, the length of which is at most k, cannot make difference between states)

- Clearly holds:

  $$\begin{array}{rlll} \text{(i)} & q \overset{0}{\equiv} q', & \text{iff} & \text{both } q \text{ and } q' \text{ are final states} \\ & & & \text{or neither is; and} \\ \text{(ii)} & q \equiv q', & \text{iff} & q \overset{k}{\equiv} q' \text{ for all } k = 0,1,2,\ldots \end{array}$$

- Idea of minimization: the k-equivalence classes of the states of the given automaton are partitioned into $(k+1)$-equivalence classes until we reach the absolut equivalence

2.3.2 Minimization algorithm

- Input: Finite automaton $M = (Q, \Sigma, \delta, q_0, F)$.

  1. (Removing redundant states) Remove from M all states, which cannot be reached from q_0 by any string.
  2. (0-equivalence) Partition the remaining states of M into two classes: non-final and final states
  3. (k-equivalence $\rightarrow$ $(k+1)$-equivalence)

```
while not(transition function compatible with class division) {
    divide states behaving in the different way into different classes;
}
return $\widehat{M} = (\hat{Q}, \Sigma, \hat{\delta}, \hat{q}_0, \widehat{F})$,
```

in which

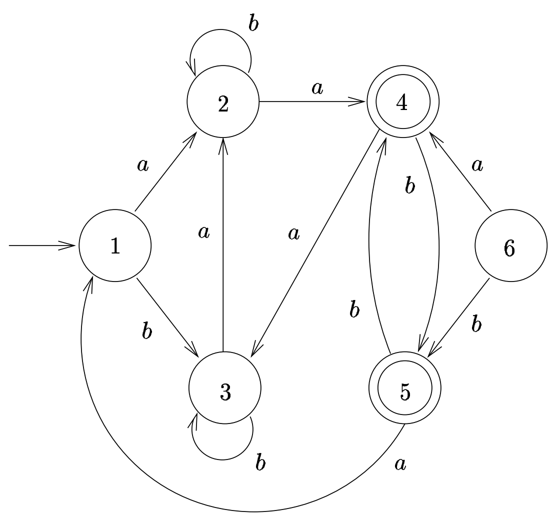
- $\hat{Q}$=state classes of M
- $\hat{\delta}$=transition function between classes
- $\hat{q}_0$=class of the initial state of M
- $\widehat{F}$=classes of the final states of M

- Final result:
  - a finite automaton $\widehat{M}$, which is equivalent with M and in which there is minimum number of states
  - $\widehat{M}$ is uniquely defined (except naming)
- Notice: Initially we had a finite number of states and in each step 3 (except the last one) we divide at least one state class, so the algorithm finishes always

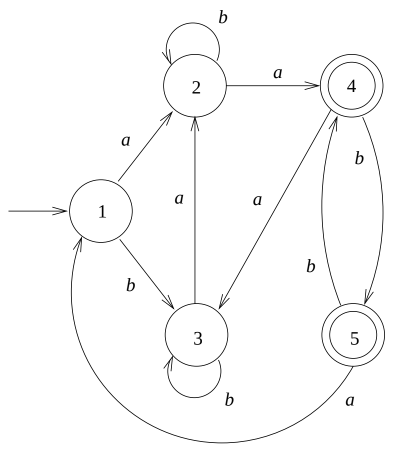
Example

- Let $M = (Q, \Sigma, \delta, q_0, F)$,
  - set of states $Q = \{1, 2, 3, 4, 5, 6\}$,
  - input alphabet $\Sigma = \{a, b\}$,
  - initial state $q_0 = \{1\}$,
  - set of final states $F = \{4, 5\}$ and
  - transition function δ:

		a	b
$\rightarrow$	1	2	3
	2	4	2
	3	2	3
$\leftarrow$	4	3	5
$\leftarrow$	5	1	4
	6	4	5



- Step 1: Removing redundant states



- Step 2: 0-equivalence
  - Divide the remaining states of M into two classes: final and other states

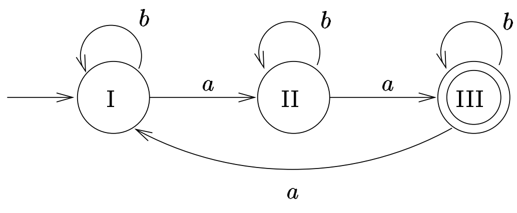
			a	b
I:	$\rightarrow$	1	2, I	3, I
		2	4, II	2, I
		3	2, I	3, I
II:	$\leftarrow$	4	3, I	5, II
	$\leftarrow$	5	1, I	4, II

- Now we can construct an automaton, in which there is
  * one state for each class
  * from each state all those transitions, which the initial states in the class have
- The state is *non-deterministic*, if we can move from it to more than one state with one character
- In example state I is nondeterministic, because by a we can move to state I or state II

- Step 3: k-equivalence $\Rightarrow$ $(k+1)$-equivalence
  - If there are no more nondeterministic states in $\widehat{M}$ then the algorithm finishes and returns $\widehat{M}$
  - Otherwise refine the division of each undeterministic state of $\widehat{M}$ further:
    * Divide the initial states inside the class into separate classes such that from each class there are only similar transitions
    * Repeat step 3
  - In our example we divide the class I
  - After that there are no more nondeterministic states and the algorithm finishes

- Final result

			a	b
I:	$\rightarrow$	1	2, II	3, I
		3	2, II	3, I
II:		2	4, III	2, II
III:	$\leftarrow$	4	3, I	5, III
	$\leftarrow$	5	1, I	4, III



$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[loop above, "a"] \arrow[loop below, "b"] \arrow[r, "a"] & q_1 \arrow[r, "b"] & q_2 \arrow[r, "a"] & \boxed{q_3} \arrow[loop above, "a"] \arrow[loop below, "b"]
\end{tikzcd}$$

2.4 Nondeterministic finite automata

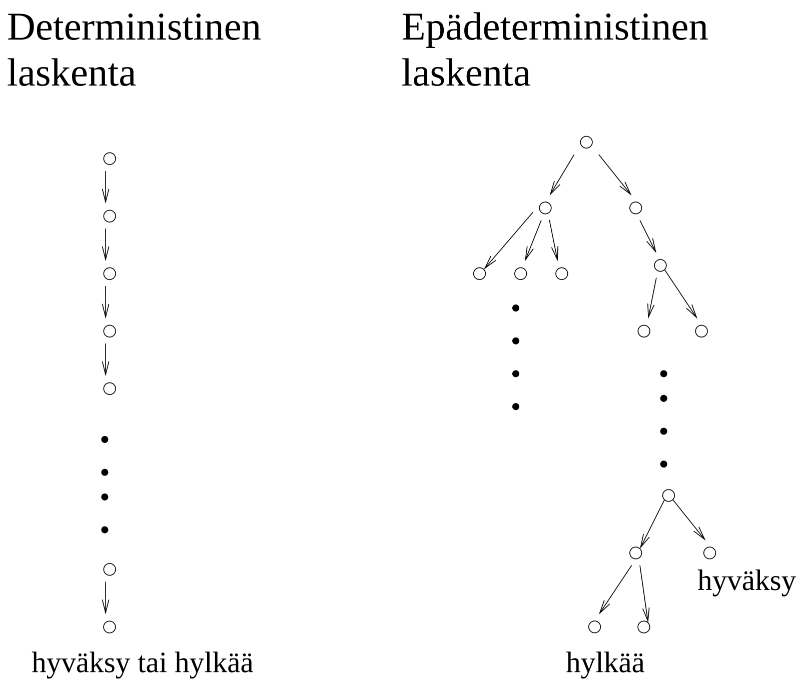
- helps to create conncetion between finite automata and *regular languages*
  - deterministic and nondeterministic automata recognize exactly same languages
  - nondeterministic automata recognize exactly regular languages
    $\Rightarrow$ So deterministic automata recognize exactly regular languages
- In nondeterministic automaton the transition function connects for a pair of previous state and input character (q, x) *a set* of possible successor states
- Non-Deterministic automaton accepts a string if some path of possible states leads to final state. If such a path doesn't exist the nondeterministic automaton rejects the input string
- E.g. in the figure the automaton accept string *abbaba*, because it can be handled in the following way:

$$(q_0, abbaba)(q_0, bbaba)(q_0, baba)$$
$$(q_0, aba)(q_1, ba)(q_2, a)(q_3, \epsilon)$$

- On the other hand we can also reach rejecting state

$$(q_0, abbaba)(q_0, bbaba)(q_0, baba)$$
$$(q_0, aba)(q_0, ba)(q_0, a)(q_0, \epsilon)$$

- Non-Deterministic automaton can be thought to process all possible derivations parallelly.



- *Definition:* Nondeterministic finite automaton is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$, in which
  - Q is a finite set of states
  - Σ is input alphabet,
  - $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is (setvalued) transition function,
  - $q_0 \in Q$ is initial state and
  - $F \subseteq Q$ a set of final states.
- The transition function of the automaton in the fig.

	a	b
$\to q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_3\}$	$\emptyset$
$\leftarrow q_3$	$\{q_3\}$	$\{q_3\}$

- Now the error state can be expressed by emptyset
- (q, w) *can* lead directly to configuration (q', w'), $(q, w) \underset{M}{\vdash} (q', w')$, if $w = aw'$ and $q' \in \delta(q, a)$. Configuration (q', w') is a *possible* immediate successor of (q, w)

- Otherwise definitions same as earlier

- Deterministic automata are special case of the nondeterministic ones $\Rightarrow$ All languages, which can be recognized by former, can also be recognized by latter

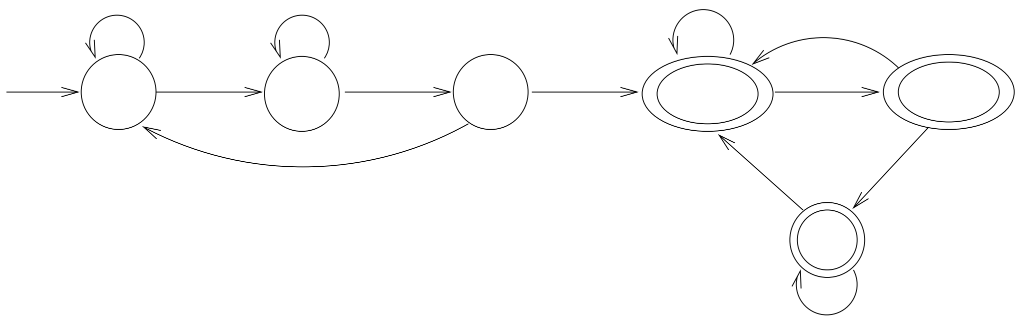
- But also vice versa: *deterministic and nondeterministic finite automata are equally powerful*

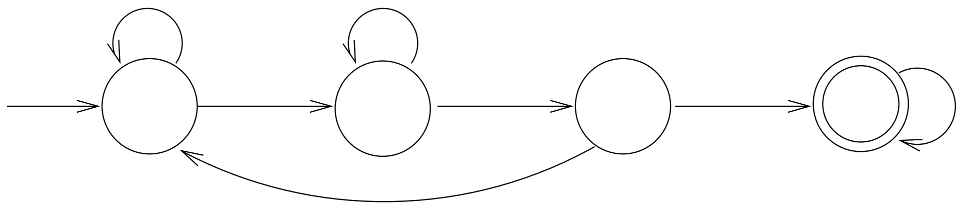
2.4.1 Determinization

- Given nondeterministic automaton M, construct the corresponding deterministic automaton $\widehat{M}$:

  1. Create the states of $\widehat{M}$ $S \subseteq \mathcal{P}(Q)$
     - i.e. the powerset of the states of M: $\mathcal{P}(Q)$
     - Mark set of states $\mathcal{P}(Q) = \{\emptyset, s_1, s_2, ..., s_m\}$, in which empty set corresponds error state and $k = 2^n - 1$
  2. Add the transitions between states of $\widehat{M}$:
     - $s_i \xrightarrow{a} s_j$, in which $s_j = \bigcup\{q' | f(q,a) = q', q \in s_i$
     - i.e. the successor set of s_i by reading a consists of all such states q', which can be reached from the states of s_i
  3. Initial state $\{q_0\}$
  4. Favorable states: all such states, which contain the original favorable state q_f i.e. all s_i, $q_f \in q_i$
  5. Remove all states, which cannot be reached from the initial state
  6. Minimize automaton
     - divide into favorable and other states
     - refine the class division until compatible with transition function

- Example:





- *Clause:* Let $A = L(M)$ language recognized by some nondeterministic finite automaton M. Then there exists a deterministic automaton $\widehat{M}$, for which $L(\widehat{M}) = A$.
  *Proof: Let $A = L(M)$, $M = (Q, \Sigma, \delta, q_0, F)$. Let's construct a determ. automaton $\widehat{M} = (\hat{Q}, \Sigma, \hat{\delta}, \hat{q_0}, \hat{F})$, which simulates the behaviour of M in its all possible states parallelly. States of $\widehat{M}$ correspond state sets of M

$$\begin{aligned} \hat{Q} &= \mathcal{P}(Q), \\ \hat{q_0} &= \{q_0\}, \\ \hat{F} &= \{S \subseteq Q \mid S \text{ contains some } q_f \in F\}, \\ \hat{\delta}(S,a) &= \bigcup_{q \in S} \delta(q,a). \end{aligned}$$

  Let's check that $L(\widehat{M}) = L(M)$. The equivalence follows, when we prove that for all $x \in \Sigma^*$ and $q \in Q$:

$$(q_0, x) \underset{M}{\vdash^*} (q, \epsilon) \Leftrightarrow (\{q_0\}, x) \underset{\widehat{M}}{\vdash^*} (S, \epsilon) \text{ and } q \in S.$$

  Induction by the length of x:

  1. $|x| = 0$: $(q_0, \epsilon) \underset{M}{\vdash^*} (q, \epsilon) \Leftrightarrow q = q_0$.
     Also $(\{q_0\}, \epsilon) \underset{\widehat{M}}{\vdash^*} (S, \epsilon) \Leftrightarrow S = \{q_0\}$.

2. *Induction assumption*: claim holds when $|x| \leq k$.

3. $|x| = k+1$: then $x = ya$ for some y, $|y| = k$, for which the claim holds by induction assumption. Now

$$\begin{aligned}
& (q_0, x) = (q_0, ya) \underset{M}{\vdash^*} (q, \epsilon) \\
\Leftrightarrow\ & \exists q' \in Q \text{ s.e. } (q_0, ya) \underset{M}{\vdash^*} (q', a) \text{ and } (q', a) \underset{M}{\vdash} (q, \epsilon) \\
\Leftrightarrow\ & \exists q' \in Q \text{ s.e. } (q_0, y) \underset{M}{\vdash^*} (q', \epsilon) \text{ and } (q', a) \underset{M}{\vdash} (q, \epsilon) \\
\Leftrightarrow\ & \exists q' \in Q \text{ s.e. } (\{q_0\}, y) \underset{\widehat{M}}{\vdash^*} (S', \epsilon) \text{ and } q' \in S' \text{ and } q \in \delta(q', a) \\
\Leftrightarrow\ & (\{q_0\}, y) \underset{\widehat{M}}{\vdash^*} (S', \epsilon) \text{ and } \exists q' \in S' \text{ s.e. } q \in \delta(q', a) \\
\Leftrightarrow\ & (\{q_0\}, y) \underset{\widehat{M}}{\vdash^*} (S', \epsilon) \text{ and } q \in \bigcup_{q' \in S'} \delta(q', a) = \hat{\delta}(S', a) \\
\Leftrightarrow\ & (\{q_0\}, ya) \underset{\widehat{M}}{\vdash^*} (S', a) \text{ and } q \in \hat{\delta}(S', a) = S \\
\Leftrightarrow\ & (\{q_0\}, ya) \underset{\widehat{M}}{\vdash^*} (S', a) \text{ and } (S', a) \underset{\widehat{M}}{\vdash} (S, \epsilon) \text{ and } q \in S \\
\Leftrightarrow\ & (\{q_0\}, x) = (\{q_0\}, ya) \underset{\widehat{M}}{\vdash^*} (S, \epsilon) \text{ and } q \in S.
\end{aligned}$$

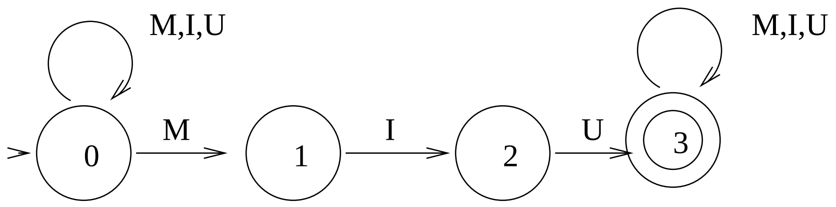
□

2.4.2 Pattern matching by nondeterministic automaton

Nondeterministic automata are suitable for describing pattern matching problems. However, for program implementation we have to determinize the automaton. Earlier we noticed that in the worst case the corresponding deterministic automaton consists of exponential number of states compared to the original one. Fortunately in pattern matching problems the determinization algorithm produces an automaton, which contains *equal number of states* with the original nondeterministic automaton, and which is also *minimal automaton!*.

E.g. Let's have alphabet $\{M, I, U\}$. Does the string contain pattern MIU? Corresponding nondeterministic automaton:

Let's determinize:

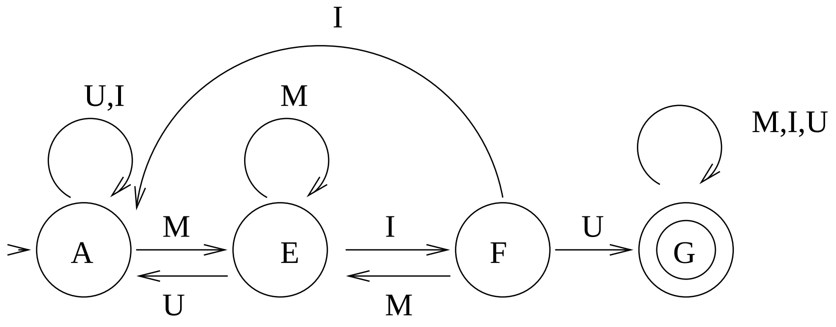


Kuva 2.1: Nondeterministic MIU-automaton.

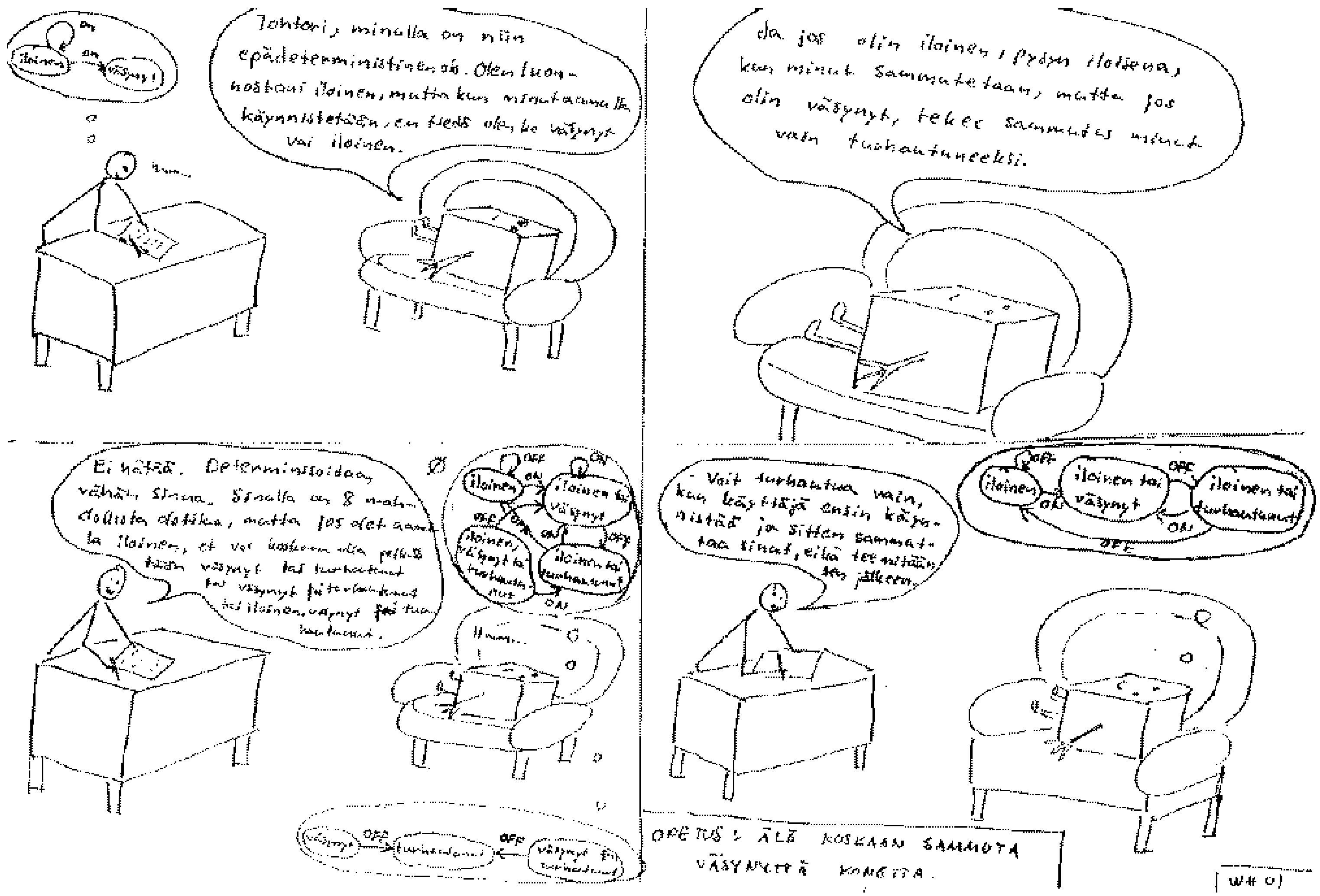
		M	I	U
	∅	∅	∅	∅
A	{0}	{0,1}=E	{0}=A	{0}=A
B	{1}	∅	{2}=C	∅
C	{2}	∅	∅	{3}=D
D	{3}	{3}=D	{3}=D	{3}=D
E	{0,1}	{0,1}=E	{0,2}=F	{0}=A
F	{0,2}	{0,3}=E	{0}=A	{0,3}=G
G	{0,3}	{0,1,3}=L	{0,3}=G	{0,3}=G
H	{1,2}	∅	{2}=C	{3} = D
I	{1,3}	{3}=D	{2,3}=J	{3}=D
J	{2,3}	{3}=D	{3}=D	{3}=D
K	{0,1,2}	{0,1}=E	{0,2}=F	{0,3}=G
L	{0,1,3}	{0,1,3}=L	{0,2,3}=M	{0,3}=G
M	{0,2,3}	{0,1,3}=L	{0,3}=G	{0,3}=G
N	{1,2,3}	{3}=D	{2,3}=J	{3}=D
O	{0,1,2,3}	{0,1,3}=L	{0,2,3}=M	{0,3}=G

Draw the diagram!

Remove the redundant states, which cannot be reached from the initial state, and combain the final states (according to minimization algorithm).



Kuva 2.2: Minimal deterministic MIU-automaton.


on
iloinen
on
väsynyt!
Hmm...
Tohtori, minulla on niin epädeterministinen olo. Olen luonnostani iloinen, mutta kun minut aamulla käynnistetään, en tiedä olenko väsynyt vai iloinen.
Ja jos olin iloinen, pysyn iloisena, kun minut sammutetaan, mutta jos olin väsynyt, tekee sammutus minut vain turhautuneeksi.
Ei hätää. Determinisoidaan vähän sinua. Sinulla on 8 mahdollista olotilaa, mutta jos olet aamulla iloinen, et voi koskaan olla pelkästään väsynyt tai turhautunut tai väsynyt ja turhautunut tai iloinen, väsynyt ja turhautunut.
Ø
OFF
iloinen
ON
iloinen tai väsynyt
OFF
OFF
ON
ON
OFF
iloinen, väsynyt tai turhautunut
iloinen tai turhautunut
ON
Hmm...
väsynyt
OFF
turhautunut
OFF
väsynyt tai turhautunut
- Voit turhautua vain, kun käyttäjä ensin käynnistää ja sitten sammuttaa sinut, eikä tee mitään sen jälkeen.
OFF
iloinen
ON
iloinen tai väsynyt
OFF
iloinen tai turhautunut
ON
OFF
OPETUS: ÄLÄ KOSKAAN SAMMUTA VÄSYNYTTÄ KONETTA.
WH 01

A comic about nondeterministic automaton consulting the doctor.

1. The automaton has two states: happy and tired. "Doctor, I have so undeterministic feeling. I am naturally happy, but when I am turned on in the morning, I don't know, if I am tired or happy"
2. "If I was happy, I stay happy, but if I was tired, turning off makes me thrustrated."
3. "Don't worry. Let's determinize you a little. You have eight possible states, but if you are in the morning happy, you can never be only tired or thrustrated or (tired or thrustrated) or (happy or tired or thrustrated)." In the upper automaton states: happy, (happy or tired), (happy or thrustrated), (happy, tired or thrustrated). In the other automaton states: tired, thrustrated, (tired or thrustrated).
4. "You can get thrustrated only if the user first turns you on and then turns you off, and does nothing afterwords" In the automaton states: happy, (happy or tired), (happy or thrustrated). What we learnt: Never turn off a tired machine.