Wyrażenia regularne i regexpy

dr inż. Marcin Ciura

Wydział Informatyki, Akademia Górniczo-Hutnicza

Motywujące przykłady

„W” czy „we”? (1)

we Francji, we Wrocławiu, we foyer

we dwoje, we troje, we czworo

we mnie

we Lwowie, we łbie, we łzach, we mgle, we śnie

„W” czy „we”? (2)

Należy używać „we” przed:

```
^([fvw]([bcdfghjklłmnprstwzż]|o[iy])
|dwoj|dwó|troj|tró|czwor|czwó|mnie$
|lwow|łb|łz|mg[lł]|sn|śn)
```

Rząd liczebników (1)

1 sprzęt	2–4 sprzęty	0, 5–9 sprzętów
11 sprzętów	12–14 sprzętów	10, 15–19 sprzętów
21 sprzętów	22–24 sprzęty	20, 25–29 sprzętów
91 sprzętów	92–94 sprzęty	90, 95–99 sprzętów

101 sprzętów

3 141 592 sprzęty

31 415 926 sprzętów

Rząd liczebników (2)

```
^0*1$ sprzęt
^([0-9]*[02-9])?[2-4]$ sprzęty
^([0-9]*[05-9]|1[0-9])$ sprzętów
```

Wołacz imion

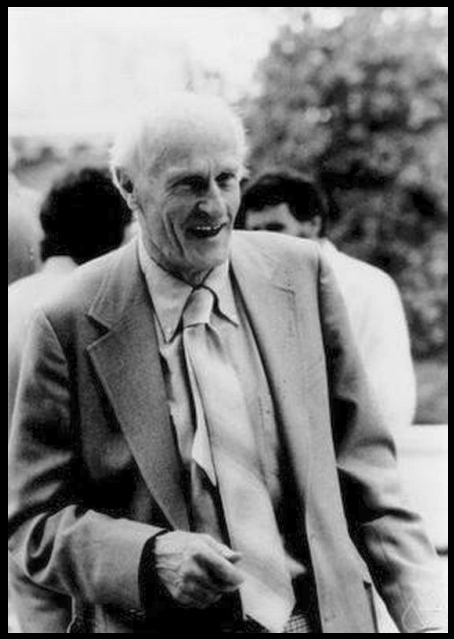
[cnsz]ia$	-len(a)	+u
ja$	-len(a)	+u
a$	-len(a)	+o
ek$	-len(ek)	+ku
eł$	-len(eł)	+le
er$	-len(er)	+rze
ł$	-len(ł)	+le
ś$	-len(ś)	+siu
niec$	-len(niec)	+ńcu
[crs]z$	-0	+u
[bfmnpswxz]$	-0	+ie
[cghjkl]$	-0	+u
d$	-0	+zie
r$	-0	+ze
t$	-len(t)	+cie

Plan na dziś: 114 slajdów, w tym 12 zagadek

- Motywujące przykłady
- Wyrażenia regularne
- Automaty skończone
- Regexpy

Wyrażenia regularne

Stephen Cole Kleene (5.1.1909–25.1.1994)



Amerykański matematyk. Wspinał się po górach, działał na rzecz ochrony przyrody. W 1951 roku, badając sieci neuronowe, wynalazł wyrażenia regularne. Jego nazwisko nosi gwiazdka Kleene'a.

Wyrażenia regularne a regex(p)y

- Wyrażenia regularne: pojęcie matematyczne
- Regex, regexp: implementacja wyrażeń regularnych w języku programowania z rozszerzeniami, które ułatwiają pracę programistom (lukier składniowy, czyli syntactic sugar)

Wyrażenia regularne: definicje (1)

- Alfabet: skończony zbiór znaków,
  na przykład {`a`, `b`}, ASCII, Unicode...
- Łańcuch znaków, krócej łańcuch: to samo, co ciąg znaków,
  na przykład `Ala␣ma␣kota`
- Długość łańcucha: liczba znaków w tym łańcuchu. Długość łańcucha oznaczamy otaczając ten łańcuch kreskami pionowymi,
  na przykład $|\texttt{Ala␣ma␣kota}| = 11$
- Łańcuch pusty: łańcuch o długości 0 znaków. Oznaczamy go grecką literą epsilon: ε

Wyrażenia regularne: definicje (2)

- Wyrażenie regularne: zgodny z pewnymi regułami łańcuch, który opisuje pewien zbiór łańcuchów. Mówimy też, że ten zbiór łańcuchów pasuje do tego wyrażenia regularnego.

Elementarne wyrażenia regularne

- Do symbolu zbioru pustego $\emptyset$ pasuje pusty zbiór łańcuchów: $\emptyset$
- Do symbolu pustego łańcucha ε pasuje zbiór, który zawiera tylko łańcuch pusty: $\{\varepsilon\}$
- Do wyrażenia regularnego złożonego z jednego znaku pasuje zbiór, który zawiera tylko jeden łańcuch o długości 1 złożony tylko z tego znaku

  Na przykład do wyrażenia regularnego a pasuje zbiór łańcuchów $\{a\}$

Złożone wyrażenia regularne: konkatenacja (1)

Jeśli R i S są wyrażeniami regularnymi, to:

- Do wyrażenia regularnego RS, czyli do konkatenacji wyrażeń regularnych R i S, pasuje zbiór takich łańcuchów, które można otrzymać, dopisując dowolny łańcuch pasujący do S tuż za dowolnym łańcuchem pasującym do R

Złożone wyrażenia regularne: konkatenacja (2)

Przykłady konkatenacji wyrażeń regularnych:

- Jeśli do R pasuje zbiór łańcuchów {d},
  a do S pasuje zbiór łańcuchów {o},
  to do RS pasuje zbiór łańcuchów {do}
- Jeśli do R pasuje zbiór łańcuchów {do, od},
  a do S pasuje zbiór łańcuchów {dać, pisać},
  to do RS pasuje zbiór łańcuchów {dodać, dopisać, oddać, odpisać}

Złożone wyrażenia regularne: alternatywa (1)

Jeśli R i S są wyrażeniami regularnymi, to:

- Do wyrażenia regularnego $R|S$, czyli do alternatywy wyrażeń regularnych R i S, pasuje suma dwóch zbiorów: zbioru takich łańcuchów, które pasują do R i zbioru takich łańcuchów, które pasują do S

Złożone wyrażenia regularne: alternatywa (2)

Przykłady alternatywy wyrażeń regularnych:

- Jeśli do R pasuje zbiór łańcuchów {do},
  a do S pasuje zbiór łańcuchów {od},
  to do $R|S$ pasuje zbiór łańcuchów {do, od}
- Jeśli do R pasuje zbiór łańcuchów {cegła, dach},
  a do S pasuje zbiór łańcuchów {cecha, cegła},
  to do $R|S$ pasuje zbiór łańcuchów {cecha, cegła, dach}

Złożone wyrażenia regularne: gwiazdka Kleene'a (1)

Jeśli R jest wyrażeniem regularnym, to:

- Do wyrażenia regularnego R^*, czyli do domknięcia Kleene'a wyrażenia regularnego R) pasuje zbiór takich łańcuchów, które powstały przez połączenie 0 lub więcej łańcuchów pasujących do R. Domknięcie Kleene'a nazywamy też gwiazdką Kleene'a.

Złożone wyrażenia regularne: gwiazdka Kleene'a (2)

Przykłady użycia gwiazdki Kleene'a:

- Jeśli do R pasuje zbiór łańcuchów {x}, to do R^* pasuje zbiór łańcuchów

  {ε, x, xx, xxx, xxxx, xxxxx, xxxxxx, xxxxxxx, ...}
- Jeśli do R pasuje zbiór łańcuchów {fa, sol}, to do R^* pasuje zbiór łańcuchów

  {ε,
  fa, sol,
  fafa, fasol, solfa, solsol,
  fafafa, fafasol, fasolfa, fasolsol,
  solfafa, solfasol, solsolfa, solsolsol,
  ...
  }

Złożone wyrażenia regularne: nawiasy

Jeśli R jest wyrażeniem regularnym, to:

- wyrażenie regularne (R) oznacza to samo, co R

Złożone wyrażenia regularne: kolejność działań

Żeby pisać mniej nawiasów, wykonujemy działania w takiej kolejności:

- najpierw wykonujemy działania w nawiasach
- potem stosujemy gwiazdkę Kleene'a
- potem konkatenujemy wyrażenia regularne
- potem budujemy alternatywy wyrażeń regularnych

Zagadka 1

1. `ab*` = `a(b*)` czy `(ab)*`?
2. `a|b*` = `a|(b*)` czy `(a|b)*`?
3. `ab|cd` = `(ab)|(cd)` czy `a(b|c)d`?

Zagadka 1: rozwiązanie

1. `ab*` = `a(b*)`
2. `a|b*` = `a|(b*)`
3. `ab|cd` = `(ab)|(cd)`

Zagadka 2

Które łańcuchy pasują do wyrażenia regularnego `(b*(a|ε)b)*`?

1. `ε`
2. `a`
3. `b`
4. `c`
5. `ab`
6. `aa`
7. `bb`
8. `ba`
9. `bbbbb`
10. `bbbba`
11. `abbbb`
12. `aaabb`
13. `bbabb`
14. `baabb`
15. `ababab`

Zagadka 2: rozwiązanie

Które łańcuchy pasują do wyrażenia regularnego `(b*(a|ε)b)*`?

1. `ε`
2. `a`
3. `b`
4. `c`
5. `ab`
6. `aa`
7. `bb`
8. `ba`
9. `bbbbb`
10. `bbbba`
11. `abbbb`
12. `aaabb`
13. `bbabb`
14. `baabb`
15. `ababab`

Automaty skończone

Automaty skończone (1)

Każdy automat skończony (finite automaton, liczba mnoga: finite automata) może wczytywać kolejne symbole i zmieniać swój stan zależnie od tych symboli. Liczba stanów automatu skończonego jest skończona.

Automaty skończone (2)

Zanim automat skończony wczyta jakikolwiek symbol, jest w stanie początkowym. Potem ten automat wczytuje kolejne symbole i przechodzi do kolejnych stanów.

Każdy stan, do którego przechodzi automat skończony, zależy od bieżącego stanu tego automatu i od tego symbolu, który ten automat właśnie wczytuje, a nie zależy od tych symboli, które ten automat wczytał, zanim przeszedł do bieżącego stanu.

Automat skończony, który jest w pewnym stanie końcowym, rozpoznał wczytany ciąg symboli. Automat skończony może mieć wiele stanów końcowych.

Automaty skończone (3)

Każdy automat skończony można przedstawić jako taki graf skierowany, w którym:

- wierzchołki nazywamy stanami automatu
- krawędzie nazywamy przejściami między stanami
- każda krawędź jest oznaczona co najmniej jednym symbolem alfabetu

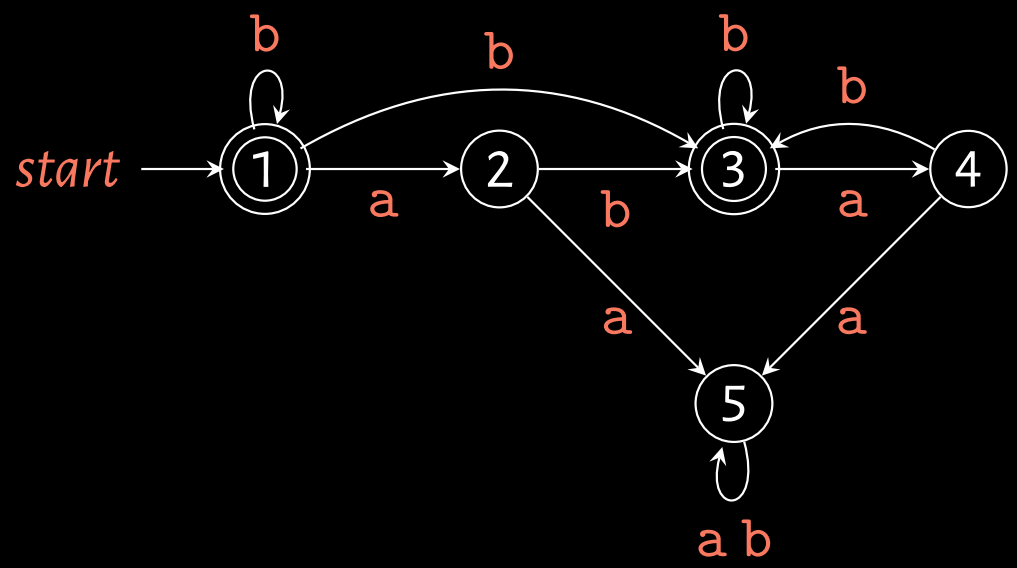
Niedeterministyczne automaty skończone

NFA (Nondeterministic Finite Automaton)

NFA, który wczytuje pewien symbol, będąc w danym stanie, może przejść do jednego z wielu różnych stanów.

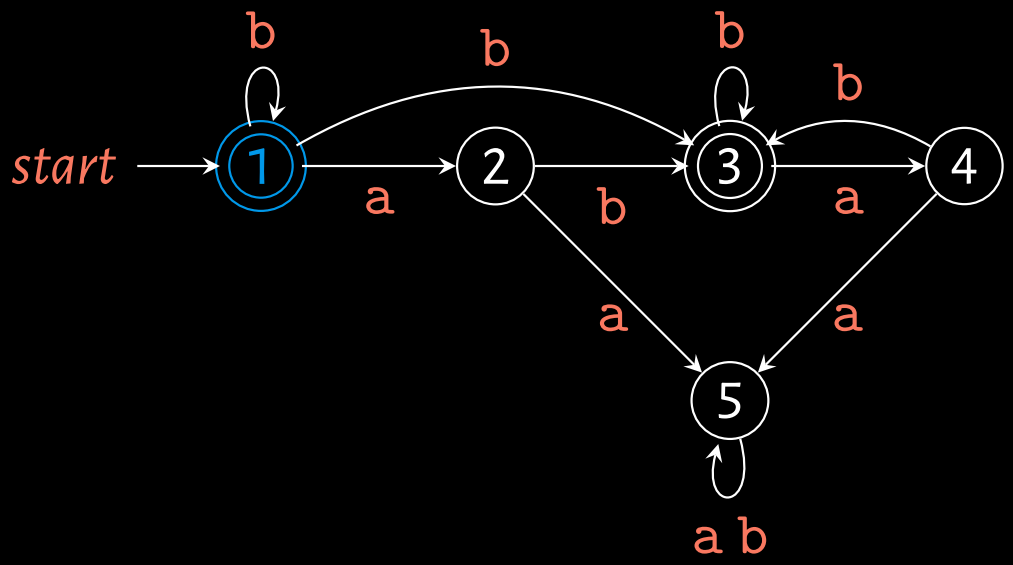
Ze stanów NFA może wychodzić więcej niż jedno przejście oznaczone tym samym symbolem alfabetu.

NFA: przykład



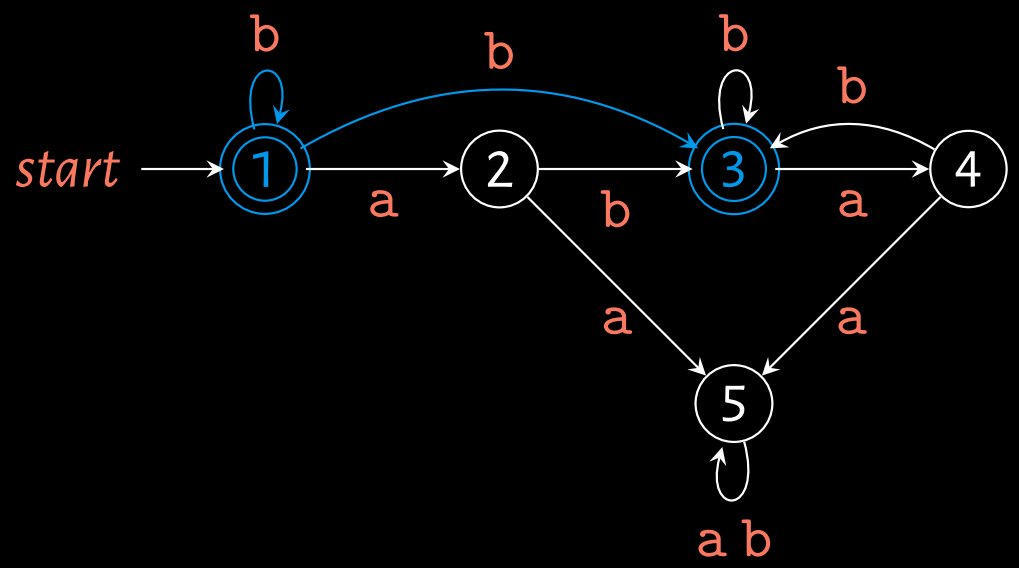
- babb

NFA: rozpoznawanie łańcucha (1)



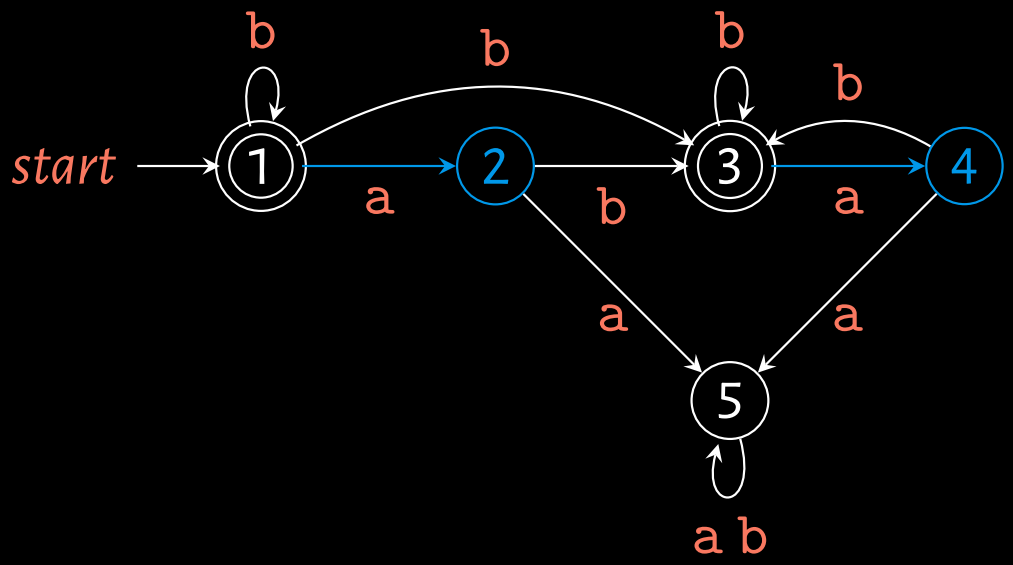
- babb

NFA: rozpoznawanie łańcucha (2)



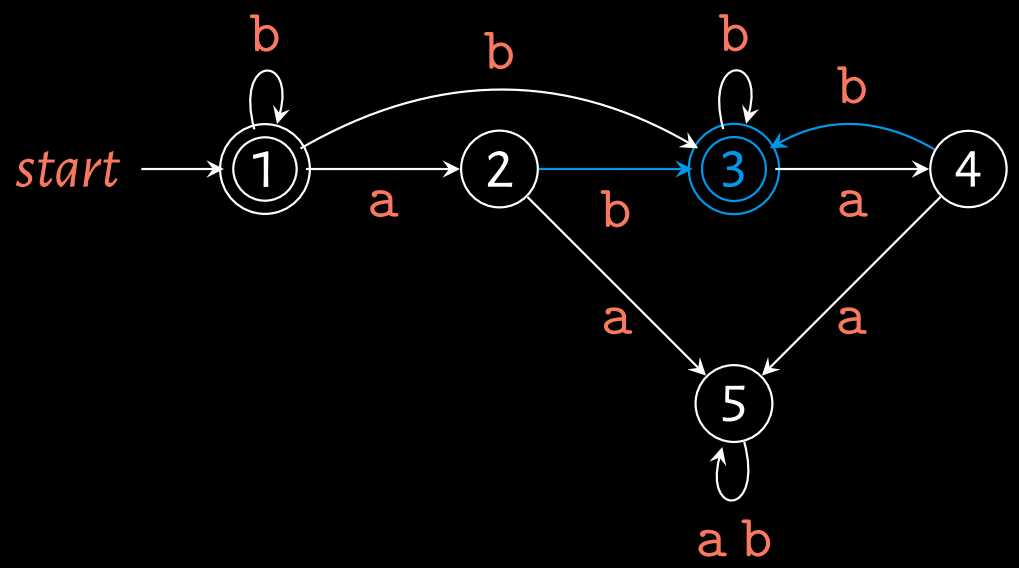
$\mathtt{b} \cdot \mathtt{abb}$

NFA: rozpoznawanie łańcucha (3)



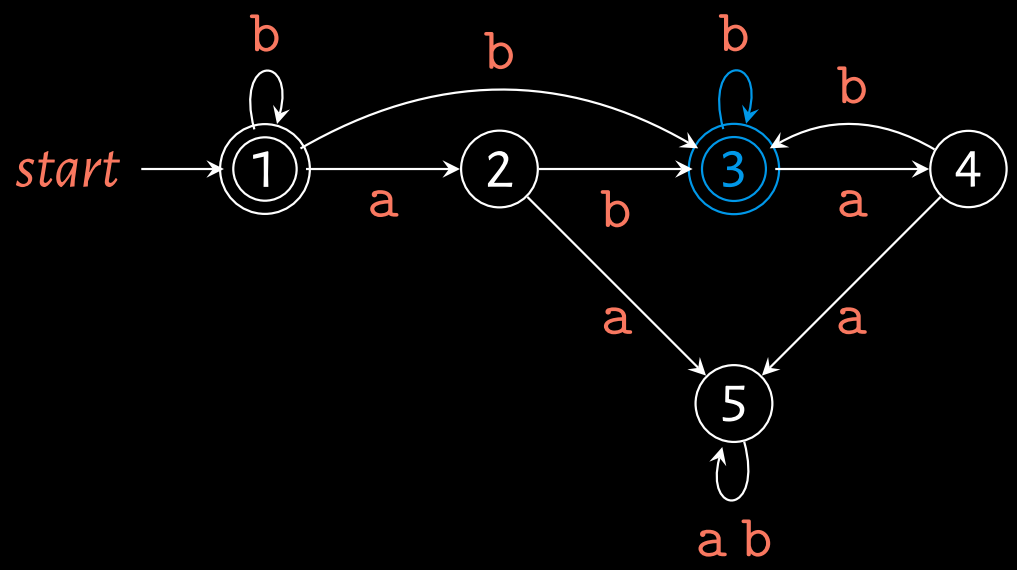
$\mathtt{ba} \cdot \mathtt{bb}$

NFA: rozpoznawanie łańcucha (4)



bab · b

NFA: rozpoznawanie łańcucha (5)



babb·

Sukces!

Automat zatrzymał się w stanie końcowym

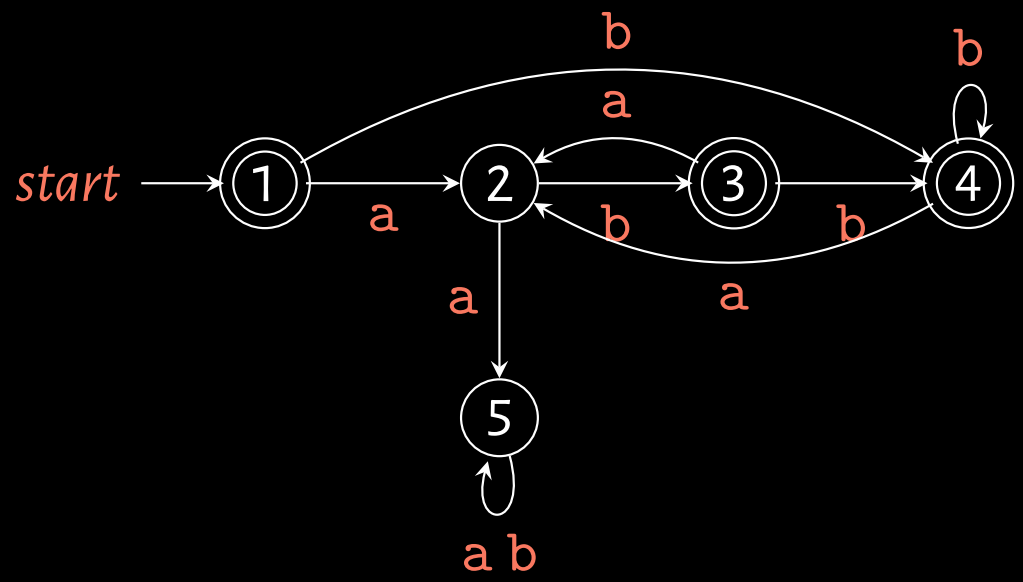
Deterministyczne automaty skończone

DFA (Deterministic Finite Automaton)

DFA, który wczytuje pewien symbol, będąc w danym stanie, zawsze przechodzi do tego samego stanu.

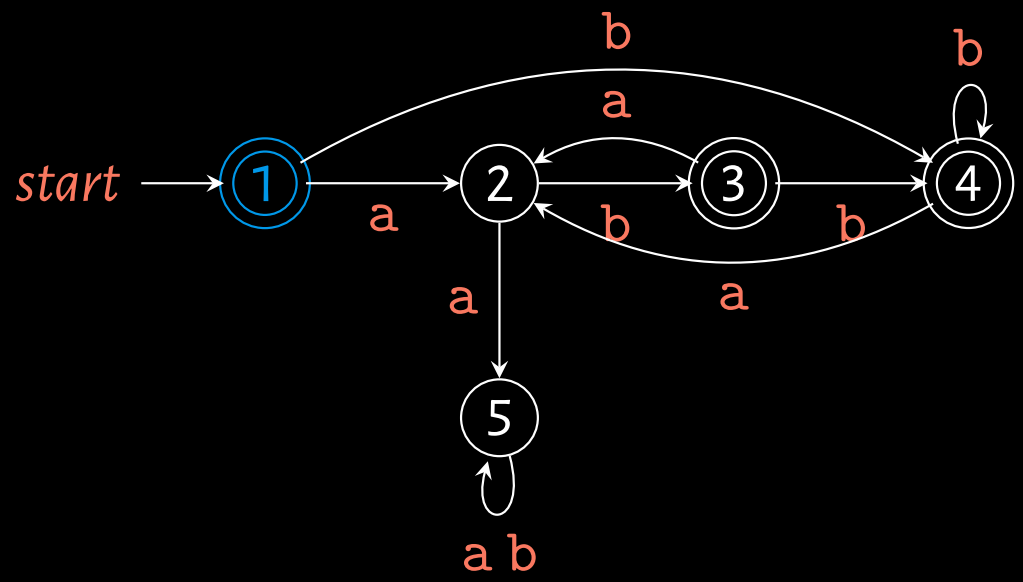
Z każdego stanu DFA wychodzi po jednym przejściu oznaczonym każdym z symboli alfabetu.

DFA: przykład



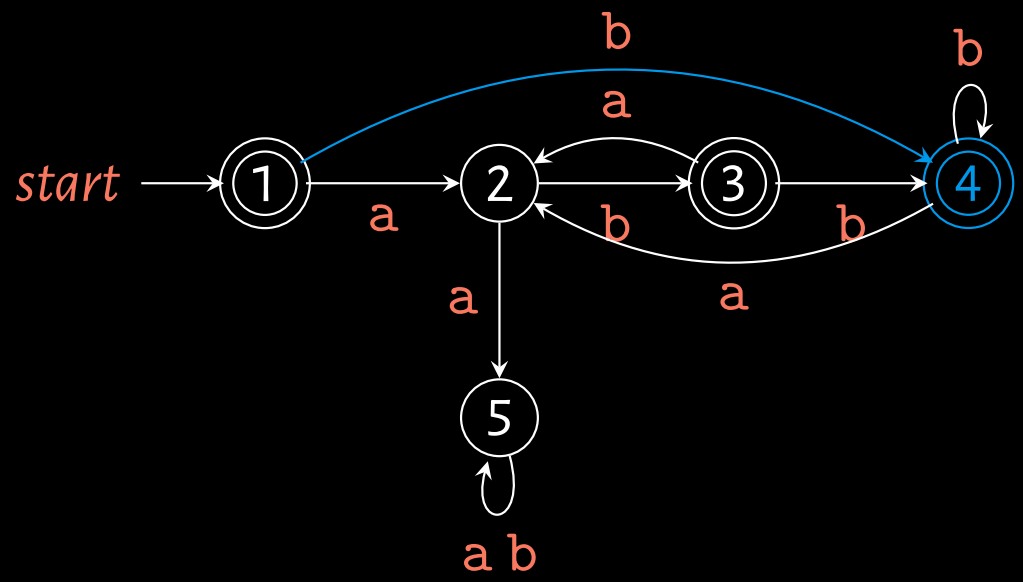
- babb

DFA: rozpoznawanie łańcucha (1)



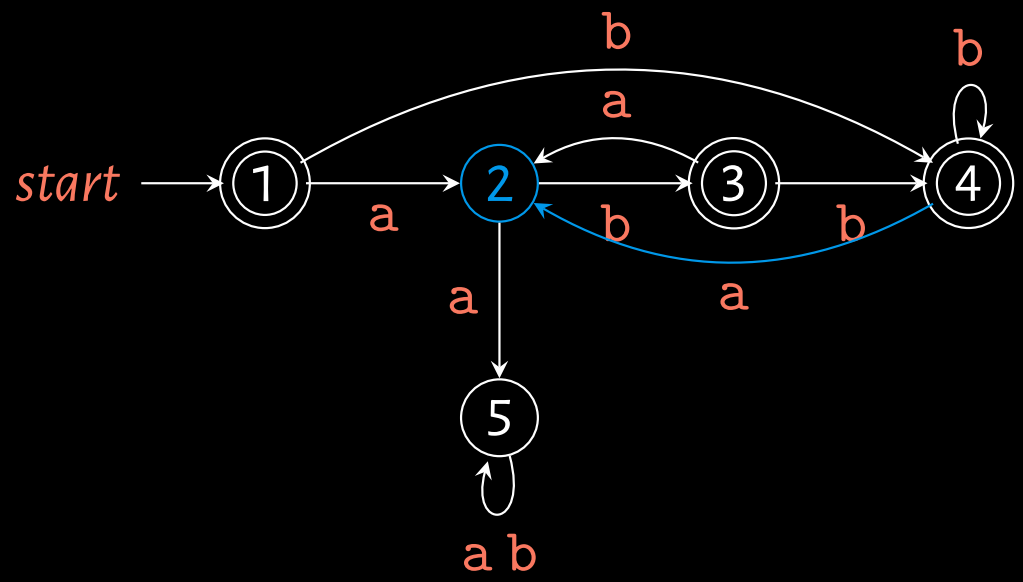
- babb

DFA: rozpoznawanie łańcucha (2)



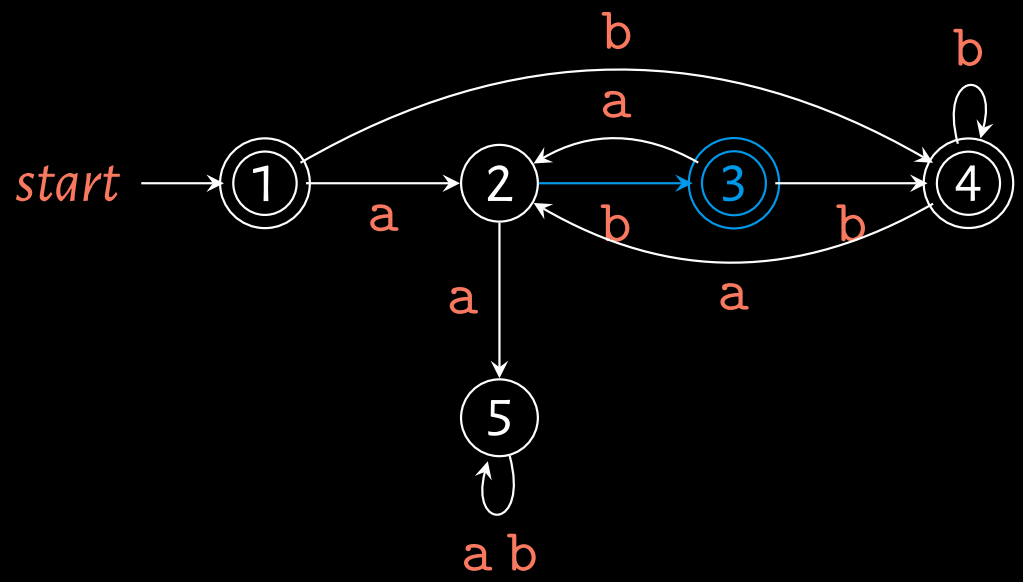
$\mathtt{b} \cdot \mathtt{abb}$

DFA: rozpoznawanie łańcucha (3)



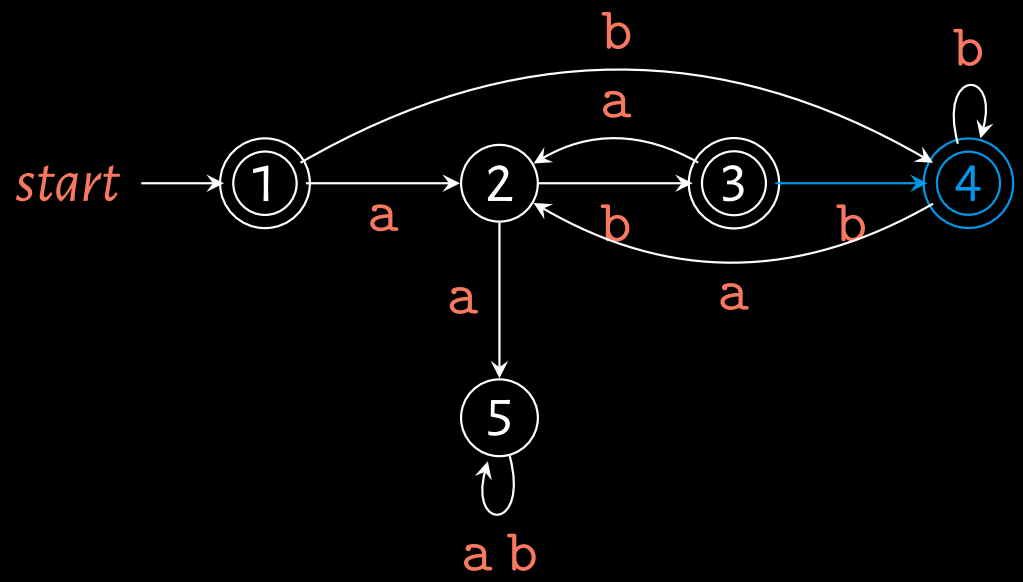
ba · bb

DFA: rozpoznawanie łańcucha (4)



bab · b

DFA: rozpoznawanie łańcucha (5)

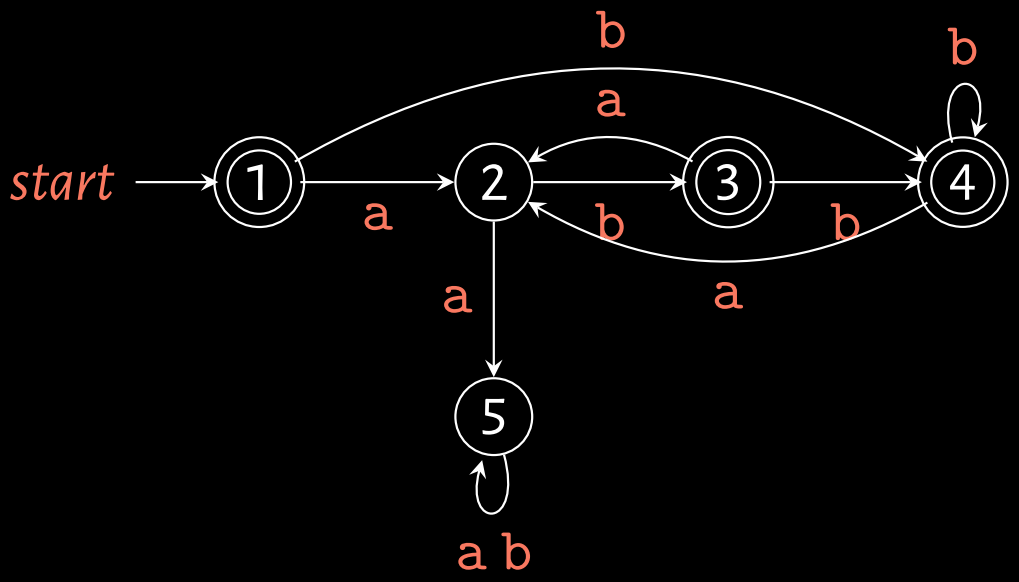


babb·

Sukces!

Automat zatrzymał się w stanie końcowym

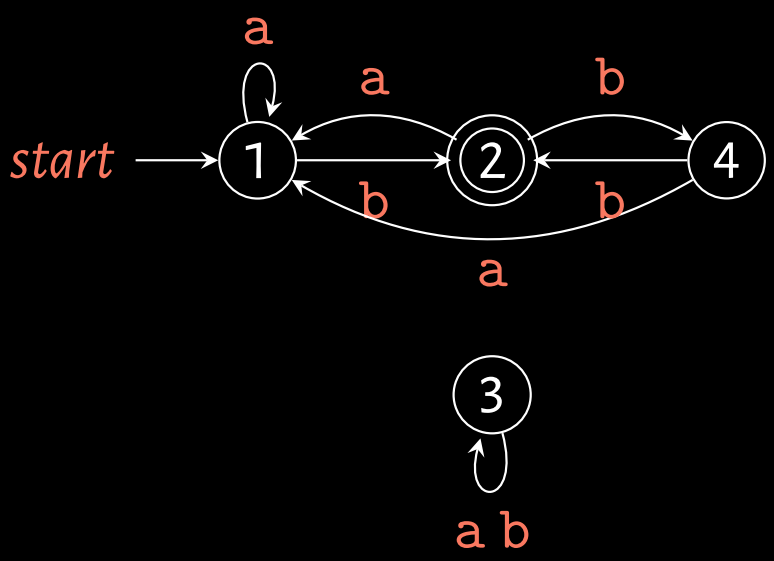
DFA: postać tablicowa

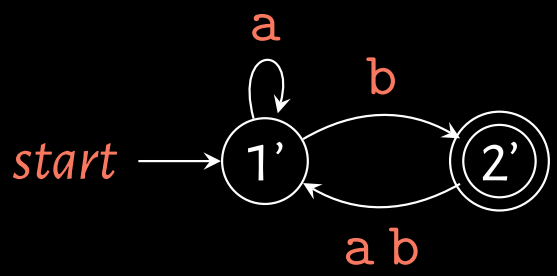


stan	końcowy?	a	b
1	tak	2	4
2	nie	3	5
3	tak	2	4
4	tak	2	4
5	nie	5	5

Równoważne automaty skończone

Automaty skończone, które rozpoznają te same zbiory łańcuchów





Minimalny DFA

W minimalnym DFA nie ma żadnych zbędnych stanów, czyli:

- takich stanów, które można połączyć w jeden stan bez zmiany zbioru łańcuchów rozpoznawanych przez DFA
- takich stanów, do których nie można dotrzeć ze stanu początkowego

Dopasowywanie łańcucha do wyrażenia regularnego

- Bezpośrednio interpretować wyrażenie regularne
- Zbudować NFA, odpowiadający wyrażeniu regularnemu (na przykład algorytmem Thompsona), po czym:
  - przechodzić przez NFA, pamiętając bieżący zbiór stanów
  - przechodzić przez NFA z nawrotami (backtracking)
  - zbudować DFA równoważny temu NFA
- Od razu zbudować DFA, odpowiadający danemu wyrażeniu regularnemu, korzystając z:
  - algorytmu Myhilla-Nerode'a
  - algorytmu DeRemera
  - pochodnych Brzozowskiego

Pochodne Brzozowskiego

Janusz Antoni Brzozowski (10.5.1935–24.10.2019)

Polsko-kanadyjski informatyk. Urodził się w Warszawie. W 1964 roku wynalazł pochodną Brzozowskiego.

Pochodna Brzozowskiego zbioru łańcuchów

Dane:

- zbiór łańcuchów $\mathcal{L}$
- symbol a

Obliczanie pochodnej Brzozowskiego $\partial_a \mathcal{L}$:

- znaleźć w zbiorze $\mathcal{L}$ takie łańcuchy, które zaczynają się od symbolu a
- odciąć pierwszy symbol a od każdego ze znalezionych łańcuchów

Zagadka 3

$$\partial_{\texttt{w}}\{\texttt{się}, \texttt{i}, \texttt{w}, \texttt{wszystko}, \texttt{więc}\} = ?$$

$$\partial_{\texttt{n}}\{\varepsilon, \texttt{nie}, \texttt{na}, \texttt{o}\} = ?$$

$$\partial_{\texttt{z}}\{\varepsilon, \texttt{z}, \texttt{za}, \texttt{abrakadabra}\} = ?$$

Zagadka 3: rozwiązanie

$$\partial_{\texttt{w}}\{\texttt{się}, \texttt{i}, \texttt{w}, \texttt{wszystko}, \texttt{więc}\} = \{\varepsilon, \texttt{szystko}, \texttt{ięc}\}$$

$$\partial_{\texttt{n}}\{\varepsilon, \texttt{nie}, \texttt{na}, \texttt{o}\} = \{\texttt{ie}, \texttt{a}\}$$

$$\partial_{\texttt{z}}\{\varepsilon, \texttt{z}, \texttt{za}, \texttt{abrakadabra}\} = \{\varepsilon, \texttt{a}\}$$

Pochodna Brzozowskiego wyrażeń regularnych (1)

$\partial_a \emptyset = ?$

Pochodna Brzozowskiego wyrażeń regularnych (2)

$\partial_a \emptyset = \emptyset$

$\partial_a \varepsilon = ?$

Pochodna Brzozowskiego wyrażeń regularnych (3)

$\partial_a \emptyset = \emptyset$

$\partial_a \varepsilon = \emptyset$

$\partial_a a = ?$

Pochodna Brzozowskiego wyrażeń regularnych (4)

$$\partial_a \emptyset = \emptyset$$

$$\partial_a \varepsilon = \emptyset$$

$$\partial_a a = \varepsilon$$

$$\partial_a b = ?$$

Pochodna Brzozowskiego wyrażeń regularnych (5)

$$\partial_a \emptyset = \emptyset$$

$$\partial_a \varepsilon = \emptyset$$

$$\partial_a a = \varepsilon$$

$$\partial_a b = \emptyset$$

$$\partial_a aR = ?$$

Pochodna Brzozowskiego wyrażeń regularnych (6)

$$\partial_a \emptyset = \emptyset$$

$$\partial_a \varepsilon = \emptyset$$

$$\partial_a a = \varepsilon$$

$$\partial_a b = \emptyset$$

$$\partial_a aR = R$$

$$\partial_a bR = ?$$

Pochodna Brzozowskiego wyrażeń regularnych (7)

$$\partial_a \emptyset = \emptyset$$

$$\partial_a \varepsilon = \emptyset$$

$$\partial_a a = \varepsilon$$

$$\partial_a b = \emptyset$$

$$\partial_a aR = R$$

$$\partial_a bR = \emptyset$$

$$\partial_a (R|S) = ?$$

Pochodna Brzozowskiego wyrażeń regularnych (8)

$$\partial_a \emptyset = \emptyset$$

$$\partial_a \varepsilon = \emptyset$$

$$\partial_a a = \varepsilon$$

$$\partial_a b = \emptyset$$

$$\partial_a aR = R$$

$$\partial_a bR = \emptyset$$

$$\partial_a (R|S) = \partial_a (R) | \partial_a (S)$$

$$\partial_a R^* = ?$$

Pochodna Brzozowskiego wyrażeń regularnych (9)

$$\partial_a \emptyset = \emptyset$$

$$\partial_a \varepsilon = \emptyset$$

$$\partial_a a = \varepsilon$$

$$\partial_a b = \emptyset$$

$$\partial_a aR = R$$

$$\partial_a bR = \emptyset$$

$$\partial_a (R \mid S) = \partial_a (R) \mid \partial_a (S)$$

$$\partial_a R^* = (\partial_a R) R^*$$

Zagadka 4

$\partial_{w}\texttt{się|i|nie|w(ε|szystko|ięc)} = ?$

$[\partial_a R^* = (\partial_a R)R^*]$

$\partial_{x}\texttt{x}^* = ?$

$\partial_{x}((\texttt{xy})^*) = ?$

$\partial_{x}((\texttt{x}|\texttt{y})^*) = ?$

$\partial_{x}((\texttt{x}^*|\texttt{y})^*) = ?$

Zagadka 4: rozwiązanie

$$\partial_w \texttt{się|i|nie|w(ε|szystko|ięc)} = \texttt{ε|szystko|ięc}$$

$$[\partial_a R^* = (\partial_a R) R^*]$$

$$\partial_x x^* = x^*$$

$$\partial_x ((xy)^*) = \partial_x (xy)(xy)^* = y(xy)^*$$

$$\partial_x ((x|y)^*) = \partial_x (x|y)(x|y)^* = (\partial_x x|\partial_x y)(x|y)^* = (x|y)^*$$

$$\partial_x ((x^*|y)^*) = \partial_x (x^*|y)(x^*|y)^* = x^*(x^*|y)^*$$

Algorytm Brzozowskiego

Dane: alfabet Σ i wyrażenie regularne R

V := $\emptyset$;
E := $\emptyset$;
W := $\{R\}$;
while $W \neq \emptyset$ **do**
    wybierz dowolny stan $w \in W$;
    $W := W \setminus \{w\}$;
    foreach $s \in \Sigma$ **do**
        $d := \partial_s w$;
        if $d \notin V$ **then**
            $W := W \cup \{d\}$;
            $V := V \cup \{d\}$;
    $E := E \cup \{(w, d, s)\}$;

Wyniki: V: zbiór stanów DFA, E: zbiór przejść DFA

Algorytm Brzozowskiego: przykład (1)

1: `(b*(a|ε)b)*`

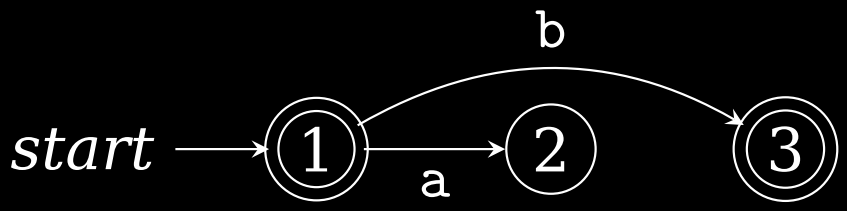
Algorytm Brzozowskiego: przykład (2)

1: `(b*(a|ε)b)*`

2: `b(b*(a|ε)b)*`

∂_a`(b*(a|ε)b)*` = `b(b*(a|ε)b)*`

Algorytm Brzozowskiego: przykład (3)



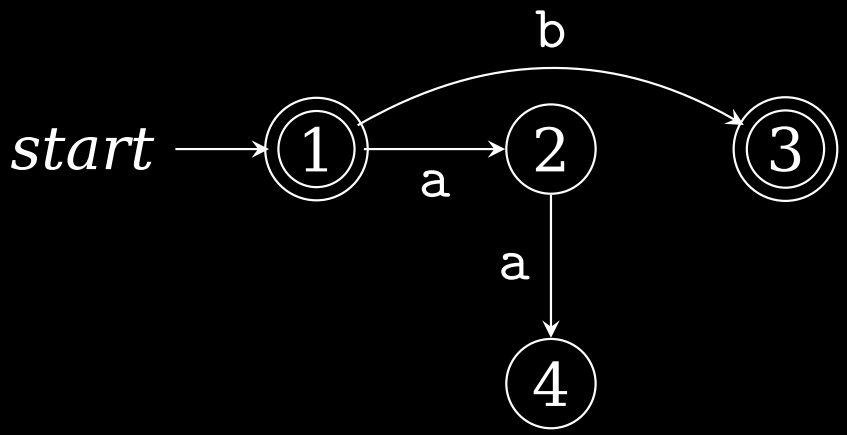
1: `(b*(a|ε)b)*`

2: `b(b*(a|ε)b)*`

3: `b*(b*(a|ε)b)*`

$$\partial_b(\texttt{b*(a|ε)b)*} = \texttt{b*(b*(a|ε)b)*}$$

Algorytm Brzozowskiego: przykład (4)



1: `(b*(a|ε)b)*`

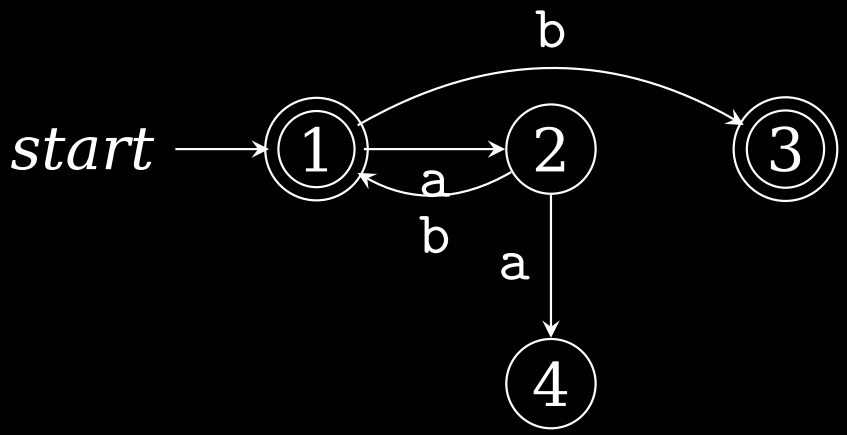
2: `b(b*(a|ε)b)*`

3: `b*(b*(a|ε)b)*`

4: Ø

∂_a`b(b*(a|ε)b)*` = Ø

Algorytm Brzozowskiego: przykład (5)



1: `(b*(a|ε)b)*`

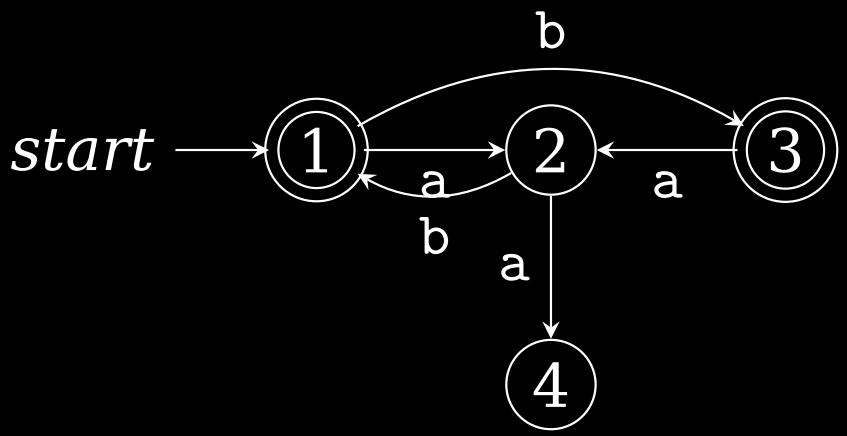
2: `b(b*(a|ε)b)*`

3: `b*(b*(a|ε)b)*`

4: Ø

∂_b`b(b*(a|ε)b)*` = `(b*(a|ε)b)*`

Algorytm Brzozowskiego: przykład (6)



1: `(b*(a|ε)b)*`

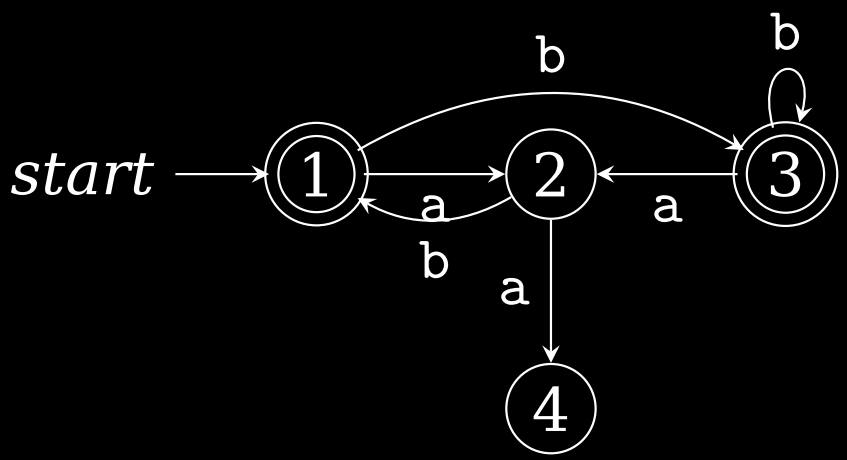
2: `b(b*(a|ε)b)*`

3: `b*(b*(a|ε)b)*`

4: Ø

$$\partial_a \texttt{b*(b*(a|ε)b)*} = \texttt{b(b*(a|ε)b)*}$$

Algorytm Brzozowskiego: przykład (7)



1: `(b*(a|ε)b)*`

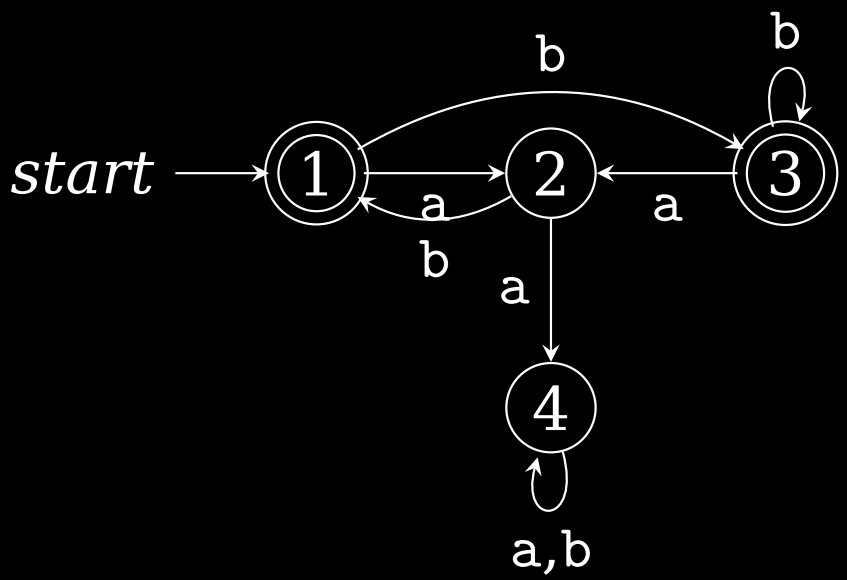
2: `b(b*(a|ε)b)*`

3: `b*(b*(a|ε)b)*`

4: Ø

$$\partial_b \texttt{b*(b*(a|ε)b)*} = \texttt{b*(b*(a|ε)b)*}$$

Algorytm Brzozowskiego: przykład (8)



$$\partial_a \emptyset = \partial_b \emptyset = \emptyset$$

1: `(b*(a|ε)b)*`

2: `b(b*(a|ε)b)*`

3: `b*(b*(a|ε)b)*`

4: $\emptyset$

Regexpy

Regexpy

Regexpy, regexy – implementacja wyrażeń regularnych z dodatkiem:

- zbiorów znaków między nawiasami kwadratowymi `[...]`
- specjalnych znaków `^` i `$`, które oznaczają początek i koniec łańcucha
- specjalnego znaku kropki `.`, która oznacza dowolny znak
- opisu wyrażeń, które występują co najwyżej jeden raz, co najmniej jeden raz lub określoną liczbę razy
- opisu łańcucha, który dopasował się do jednego z poprzednich wyrażeń (referencja wsteczna, czyli backreference)
- innych udogodnień, często różnych w różnych implementacjach

Regexpy: działania na łańcuchach

- Dopasowywanie
- Wyszukiwanie
- Zamiana części łańcucha
- Podział łańcucha na części

Regexpy w Pythonie

```
import re

r1 = re.compile('kot|pies')
r1.match(text)    # Dopasowanie
r1.search(text)   # Wyszukiwanie
r1.findall(text)  # Wyszukiwanie
r1.sub('królik', text)  # Zamiana części łańcucha

r2 = re.compile(',|;')
r2.split(text)    # Podział łańcucha na części
```

Regexpy: zbiory znaków `[...]` (1)

Nawiasy kwadratowe `[...]` otaczają takie wyrażenia, do których pasują zbiory znaków. Takim wyrażeniem może być:

- łańcuch znaków, na przykład regexp `[ąę]` jest równoważny regexpowi `(ą|ę)`
- przedziały znaków połączone znakiem minusa `-`, na przykład regexp `[A-D]` jest równoważny regexpowi `(A|B|C|D)`
- zbiory znaków poprzedzone znakiem daszku, na przykład do regexpa `[^x-z]` pasuje każdy pojedynczy znak oprócz `x`, `y` i `z`

Regexpy: zbiory znaków `[...]` (2)

Przykład:

Do regexpa `[A-ZĄĆĘŁŃÓŚŹŻa-ząćęłńóśźż]` pasują pojedyncze litery alfabetu polskiego wraz z literami `q`, `v` i `x`

Regexpy: znaki specjalne `^` `]` – wewnątrz zbiorów znaków `[...]` (1)

Znak daszku `^` pasuje:

- do części regexpa `[\^...]`
- do części regexpa `[...^...]`

Regexpy: znaki specjalne `^` `]` – wewnątrz zbiorów znaków `[...]` (2)

Znak nawiasu kwadratowego zamykającego `]` pasuje:

- do części regexpa `[...\]...]`
- do części regexpa `[]...]`

Regexpy: znaki specjalne `^` `]` `-` wewnątrz zbiorów znaków `[...]` (3)

Znak minusa `-` pasuje:

- do części regexpa `[...\-...]`
- do części regexpa `[-...]`
- do części regexpa `[...-]`

Regexpy: daszek `^` i dolar `$`

Początek łańcucha pasuje do znaku daszku `^`

Koniec łańcucha pasuje do znaku dolara `$`

Znaki daszku i dolara pasują do regexpów `\^` i `\$`

Częsty błąd: Regexp `^kot|pies$` jest równoważny regexpowi `(^kot)|(pies$)`
Taki regexp, do którego pasują tylko łańcuchy `kot` i `pies`, to `^(kot|pies)$`

Zagadka 5

```
import re

r1 = re.compile('dom')
r1.search('wiadomo') is not None

r2 = re.compile('^dom')
r2.search('domownik') is not None

r3 = re.compile('dom$')
r3.search('świadom') is not None

r4 = re.compile('^dom$')
r4.search('domek') is not None
```

Zagadka 5: rozwiązanie

```
import re

r1 = re.compile('dom')
r1.search('wiadomo') is not None
True
r2 = re.compile('^dom')
r2.search('domownik') is not None
True
r3 = re.compile('dom$')
r3.search('świadom') is not None
True
r4 = re.compile('^dom$')
r4.search('domek') is not None
False
```

Zagadka 6

```
import re

r = re.compile('dom')
r.match('wiadomo') is not None

r.match('domownik') is not None

r.match('świadom') is not None

r.match('domek') is not None
```

Zagadka 6: rozwiązanie

```
import re
r = re.compile('dom')
r.match('wiadomo') is not None
False
r.match('domownik') is not None
True
r.match('świadom') is not None
False
r.match('domek') is not None
True
```

Regexpy: kropka `.`

Do znaku kropki `.` pasuje dowolny znak oprócz znaku nowego wiersza

Znak kropki pasuje do regexpa `\.` lub `[.]`

Zagadka 7

```
import re

r1 = re.compile('^d.m$')
r1.findall(słownik, re.MULTILINE)


r2 = re.compile('^K.*ów$')
r2.findall(słownik, re.MULTILINE)
```

Zagadka 7: rozwiązanie

```
import re

r1 = re.compile('^d.m$')
r1.findall(słownik, re.MULTILINE)
['dam', 'dem', 'dom', 'dum', 'dym']

r2 = re.compile('^K.*ów$')
r2.findall(słownik, re.MULTILINE)
['Kraków', 'Knurów', 'Kryspinów', 'Krzeszów',
 'Kijów', 'Kiszyniów',...
```

Regexpy: powtórzenia

- Znak zapytania ? oznacza, że poprzednie wyrażenie może się powtórzyć albo 0 razy albo 1 raz:
  - $R? = R|\varepsilon$
- Znak plus + oznacza, że poprzednie wyrażenie może się powtórzyć co najmniej 1 raz:
  - $R+ = R|R^*$
- Część regexpa między nawiasami klamrowymi {...} oznacza to, ile razy może się powtórzyć poprzednie wyrażenie:
  - $R\{3\} = RRR$
  - $R\{3,5\} = RRR|RRRR|RRRRR$
  - $R\{2,\} = RRR^*$
  - $R\{,3\} = \varepsilon|R|RR|RRR$

Dygresja 1: „syndrom pochylonych wykałaczek” (1)

Leaning Toothpick Syndrome

W regexpach, które opisują znak lewego ukośnika `\`, trzeba poprzedzić ten znak kolejnym lewym ukośnikiem.

Dodatkowo w wielu językach programowania lewe ukośniki wewnątrz stałych łańcuchowych trzeba poprzedzać lewymi ukośnikami.

Przykład regexpa, do którego pasuje łańcuch `C:\`

```
import re

root_dir = re.compile('^[A-Za-z]:\\\\$')
```

Dygresja 1: „syndrom pochylonych wykałaczek” (2)

Leaning Toothpick Syndrome

W regexpach, które opisują znak lewego ukośnika `\`, trzeba poprzedzić ten znak kolejnym lewym ukośnikiem.

Dodatkowo w wielu językach programowania lewe ukośniki wewnątrz stałych łańcuchowych trzeba poprzedzać lewymi ukośnikami.

Przykład regexpa, do którego pasuje łańcuch `C:\`

```
import re

root_dir = re.compile('^[A-Za-z]:\\\\$')
root_dir = re.compile(r'^[A-Za-z]:\\$')
```

Regexpy: grupy (1)

nazywamy część regexpa między nawiasami okrągłymi `(...)`

Zagadka 7

Co pasuje do takiego regexpa?

```
^(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])[.]
 (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])[.]
 (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])[.]
 (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])$
```

Regexpy: grupy (2)

```
import re

r = re.compile(
  '^(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])[.]'
  '(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])[.]'
  '(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])[.]'
  '(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])$')
m = r.match('127.0.0.1')
m.group(1), m.group(2), m.group(3), m.group(4)
```

Regexpy: grupy (3)

```
import re

r = re.compile(
  '^(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])[.]'
  '(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])[.]'
  '(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])[.]'
  '(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])$')
m = r.match('127.0.0.1')
m.group(1), m.group(2), m.group(3), m.group(4)
('127', '0', '0', '1')
```

Regexpy: referencje wsteczne

Do wyrażeń `\1`...`\99` pasuje ten sam łańcuch, który pasuje do grupy o odpowiednim numerze kolejnym

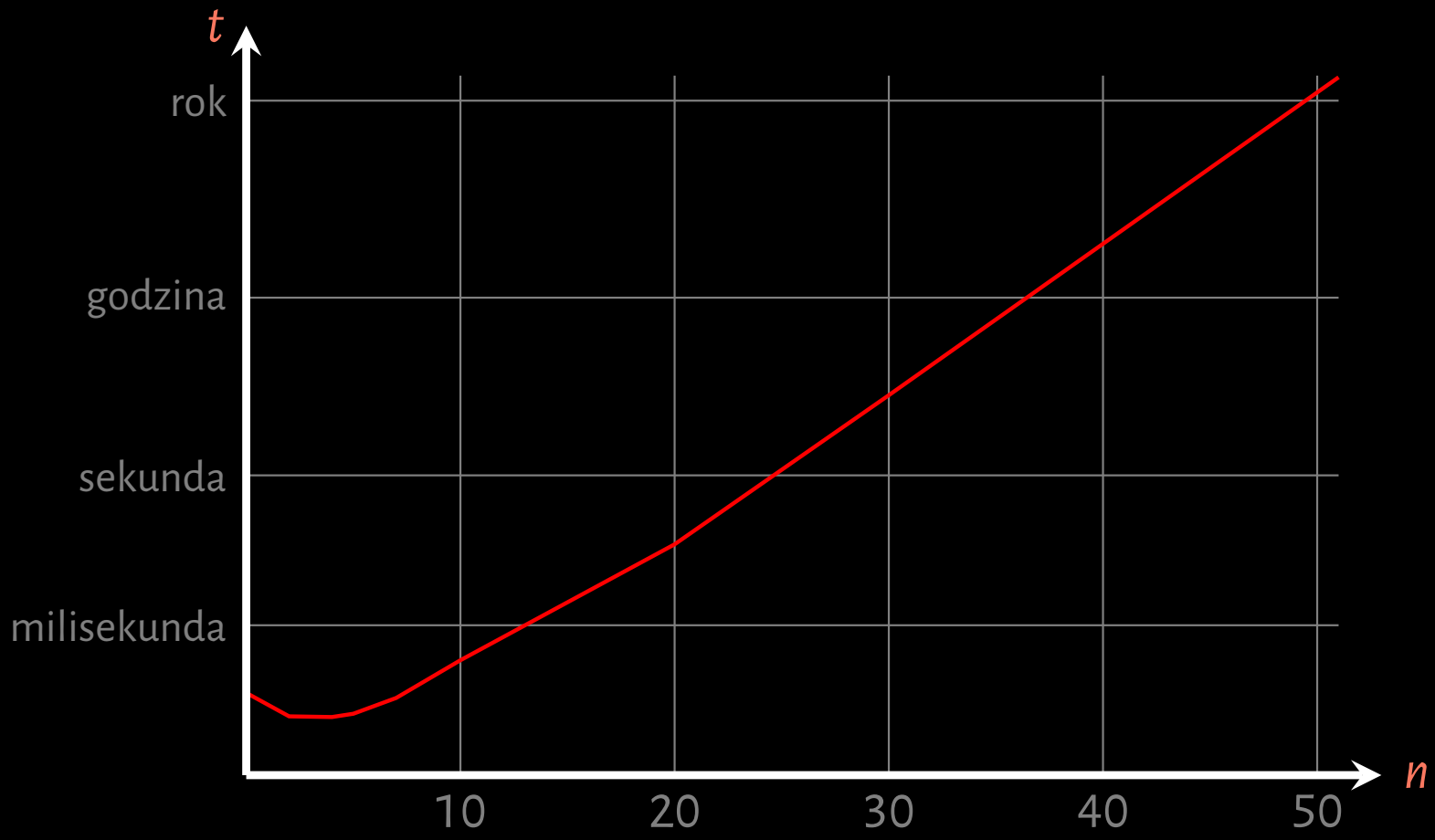
Zagadka 8

```
import re

r = re.compile(r'^(.+)\1$')
r.findall(słownik, RE.MULTILINE)
```

Zagadka 8: rozwiązanie

```
import re

r = re.compile(r'^(.+)\1$')
r.findall(słownik, RE.MULTILINE)
['baba', 'berber', 'bobo', 'dada', 'dodo',
 'dowodowo', 'dudu', 'dziadzia', 'dzidzi', 'gogo',
 'jaja', 'jojo', 'kankan', 'kuku', 'kuskus', 'lala',
 'lulu', 'mama', 'niania', 'ojoj', 'papa', 'psipsi',
 'rowerowe', 'siusiu', 'tata', 'toto']
```

Dygresja 2: nieliniowy wzrost liczby nawrotów (1)




```
r = re.compile(f'(a?){{{n}}}a{{{n}}}')
r.match(n * 'a')
```

Dygresja 2: blokada usług przez regexpy (1)

DoS, czyli Denial of Service

Gdy użytkownicy mogą:

- wprowadzać dowolne regexpy
- wprowadzać dowolne łańcuchy znaków, które prowadzą do nieliniowej liczby nawrotów regexpa użytego w programie

Dotyczy tylko regexpów korzystających z NFA!

Dygresja 2: blokada usług przez regexpy (2)

W 150 najpopularniejszych javowych aplikacjach na GitHubie, które zawierały jakiekolwiek regexpy, znaleziono:

2868 regexpów

522 regexpy z nieliniową liczbą nawrotów, w tym

37 regexpów z wykładniczą liczbą nawrotów

Badaczom udało się zablokować na ponad 10 minut 27 z tych aplikacji.

(V. Wüstholz i inni, Static Detection of DoS Vulnerabilities in Programs that use Regular Expressions, w: TACAS'17)

Zagadka 9

```
^[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?$
```

Zagadka 9: rozwiązanie

```
^[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?$
```

Do tego regexpa pasują liczby w notacji naukowej,
na przykład `3.7698294e7`

Zagadka 10

```
^M{,3}(C[MD]|D?C{,3})(X[CL]|L?X{,3})(I[XV]|V?I{,3})$
```

Zagadka 10: znaleźć bug

```
^M{,3}(C[MD]|D?C{,3})(X[CL]|L?X{,3})(I[XV]|V?I{,3})$
```

Do tego regexpa pasują liczby rzymske,

na przykład `MMXXIV`

Zagadka 10: rozwiązanie

```
^M{,3}(C[MD]|D?C{,3})(X[CL]|L?X{,3})(I[XV]|V?I{,3})$
```

Do tego regexpa pasują liczby rzymske,

na przykład `MMXXIV`, ale pasuje do niego również łańcuch pusty

Zagadka 11

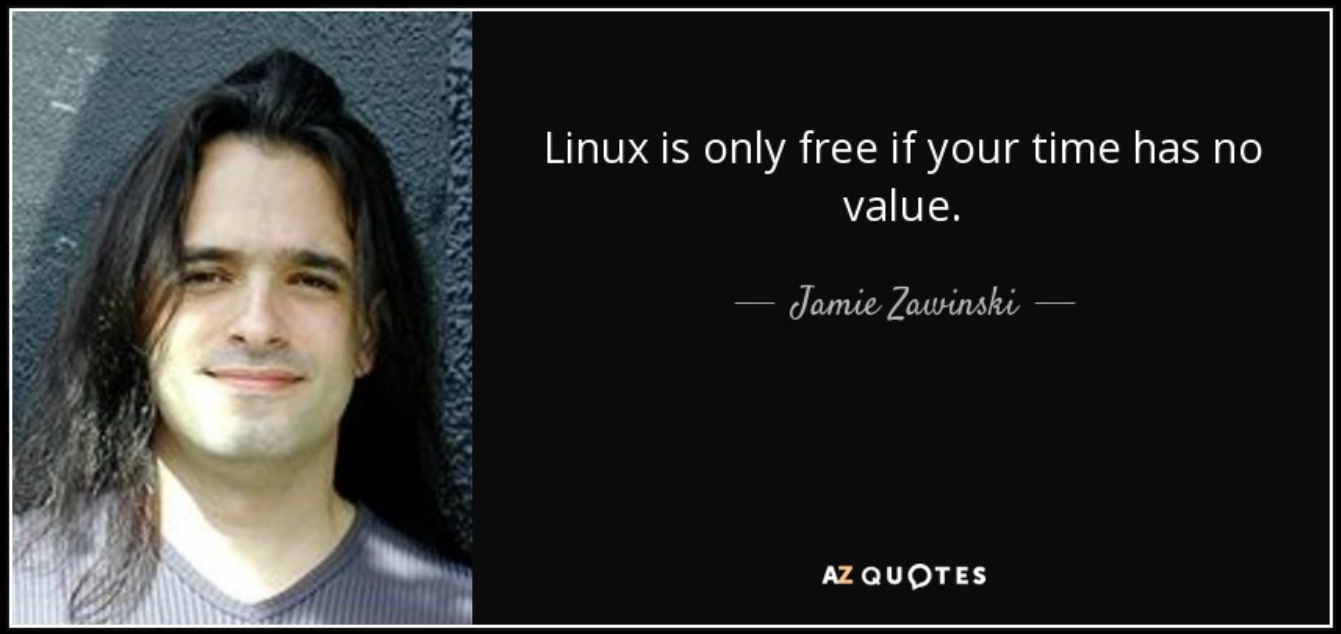
```
^[0-9]{4}-([0][0-9]|1[0-2])-([0-2][0-9]|3[01])$
```

Zagadka 11: rozwiązanie

```
^[0-9]{4}-([0][0-9]|1[0-2])-([0-2][0-9]|3[01])$
```

Do tego regexpa pasują daty w formacie ISO 8601,
na przykład `1791-05-03`

Jamie Zawinski (3.11.1968–)



Amerykański programista. Twórca przeglądarki Netscape Navigator. Właściciel klubu nocnego DNA Lounge w San Francisco. Autor wielu trafnych powiedzeń.

Netscape Navigator → Mozilla → Firefox

Trzeba uważać, żeby nie przedobrzyć

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

Jamie Zawinski (1997)

Regexp, który dokładniej opisuje daty w formacie ISO 8601

```
^((((19|20)(([02468][048])|([13579][26]))-02-29))|
((20[0-9][0-9])|(19[0-9][0-9]))-
((((0[1-9])|(1[0-2]))-((0[1-9])|(1[0-9])|(2[0-8])))|
((((0[13578])|(1[02]))-31)|(((0[1,3-9])|
(1[0-2]))-(29|30)))))$
```

Zagadka 12

```
^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+[A-Za-z]{2,}$
```

Zagadka 12: rozwiązanie

```
^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+[A-Za-z]{2,}$
```

Pasuje do adresów email,

na przykład do `mgc@agh.edu.pl`

Regexp, który dokładniej opisuje adresy email

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

Silniki regexpów

- C – zewnętrzne biblioteki: PCRE (Perl-Compatible Regular Expressions) (NFA), biblioteka opakowująca RE2 (DFA)
- C++11 – `std::basic_regex<char>`, `std::basic_regex<wchar_t>` (NFA), zewnętrzna biblioteka RE2 (DFA)
- Go – `import ''regexp''` (DFA)
- Java – `import java.util.regex.Matcher; import java.util.regex.Pattern;` (NFA)
- Javascript – `let r = /[Rr]egexp?/;` (NFA)
- Python – `import re` (NFA)

Regexpy: zalety i wady

+ zwięzłe: jeden regexp może zastępować wiele wierszy kodu
+ przenośne: podstawowe konstrukcje są takie same w różnych językach programowania
− bardziej złożone regexpy mogą być nieczytelne
− jeśli silnik regexpów używa NFA, czas dopasowania regexpa do łańcucha znaków może rosnąć szybciej niż liniowo w stosunku do długości tego łańcucha

Podsumowanie

Wyrażenia regularne

- konkatenacja, alternatywa, gwiazdka Kleene'a
- automaty skończone: NFA, DFA, minimalny DFA
- algorytm budowania DFA oparty na pochodnych Brzozowskiego

Regexpy

- znaki specjalne `[...]`, `^`, `$`, `.`
- znaki specjalne `?`, `+`, `{...}`
- metody w Pythonie: `.match`, `.search`, `.sub`
- dostęp do grup w Pythonie: `.group`

Do zobaczenia
na następnym wykładzie

Źródła zdjęć

Wikipedia

https://www.azquotes.com/author/16151-Jamie_Zawinski