Szymon Toruńczyk

Języki, automaty i obliczenia

Spis treści

O tym skrypcie

Ten skrypt zawiera materiały z wykładu prowadzonego w latach 2018-2020 na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego. Polecanym materiałem pomocniczym do nauki przedmiotu jest zbiór zadań[1] Niwińskiego i Ryttera *200 Problems in Formal Languages and Automata Theory*. Książka ta zawiera również rozwiązania zadań.

[1] Książka jest dostępna w bibliotece wydziałowej; zobacz też https://sites.google.com/view/200problems, gdzie dostępne są treści zadań.
Przez NRXX oznaczam zadanie numer XX ze zbioru zadań.

Podział skryptu na rozdziały nie pokrywa się podziałem wykładów. Początki kolejnych wykładów oznaczone są na marginesie. Poza materiałami z wykładu, w tym skrypcie pojawia się czasem materiał omówiony na ćwiczeniach, lub dodatkowe materiały, w rozdziałach oznaczonych $*$. Na marginesach umieszczam przypisy lub komentarze związane z tekstem głównym, w tym ćwiczenia.

Podziękowania. Dziękuję Piotrowi Ambroszczykowi, Michałowi Balcerzakowi, Kamilowi Cywińskiemu, Kamilowi Ćwintalowi, Mateuszowi Danowskiemu, Krzysztofowi Drużyckiemu, Marcinowi Gałązce, Resulowi Hangeldiyevowi, Krzysztofowi Małysie, Filipowi Murlakowi, Marcie Nowakowskiej, Jackowi Olczykowi, Hubertowi Pełczyńskiemu, Krzysztofowi Piesiewiczowi, Wojciechowi Przybyszewskiemu, Kamilowi Rychlewiczowi, Wojciechowi Rytterowi, Jakubowi Skrajnemu, Damianowi Werpachowskiemu i Karolowi Węgrzyckiemu za uwagi dotyczące skryptu.

Szymon Toruńczyk, Warszawa, 13 września 2021

1
Słowa i języki

WYKŁAD 1

Słowa. Przez *alfabet* rozumiemy dowolny zbiór, którego elementy nazywane są *literami* bądź *symbolami*. W tym wykładzie, będziemy jedynie rozważali alfabety *skończone*, choć są również ciekawe i przydatne zastosowania alfabetów nieskończonych. Alfabety oznaczamy literam A, B, C itd. lub greckimi Σ, Γ, Δ itd. Przykładowe alfabety to $\{0,1\}$ lub $\{a,b,c\}$. *Słowo* nad alfabetem A jest to skończony ciąg liter z alfabetu A. Przykładowo,

011101 jest słowem nad alfabetem $\{0,1\}$,

baba jest słowem nad alfabetem $\{a,b,c\}$.

Jeśli słowo w ma długość n, to piszemy $|w| = n$; mówimy też, że słowo w ma n *pozycji*, liczonych od 1. Jeśli $1 \leqslant i \leqslant |w|$, to przez $w[i]$ oznaczamy i-tą literę słowa w. Jest tylko jedno słowo długości 0, nazywane słowem *pustym* i oznaczone symbolem ε. *Konkatenacja* słów $u = a_1 \cdots a_k$ oraz $v = b_1 \cdots b_l$ to słowo $a_1 \cdots a_k b_1 \cdots b_l$, oznaczane $u \cdot v$ lub uv. Przez A^* oznaczamy zbiór wszystkich słów nad alfabetem A.

Formalnie, słowa długości $n \geqslant 0$ nad alfabetem A to n-krotki liter, tzn. elementy zbioru A^n. Zbiór wszystkich słów to $A^* = \bigcup_{n \geqslant 0} A^n$.

Języki. W informatyce teoretycznej, jednym z głównych obiektów badań są zbiory słów, zwane *językami*. Formalnie, *językiem* nad alfabetem A nazywamy dowolny podzbiór $L \subseteq A^*$. Przykładowe języki to:

1. Skończone zbiory słów, np. $\{aba, bab\} \subseteq \{a,b\}^*$, a także język pusty $\emptyset \subseteq \{a,b\}^*$ oraz język składający się wyłącznie z słowa pustego, $\{\varepsilon\} \subseteq \{a,b\}^*$.
2. Zbiór wszystkich słów nad alfabetem A, tj. A^*.
3. Zbiór wszystkich słów zawierających literę a (gdzie $a \in A$), oznaczany $A^* a A^*$.
4. Zbiór wszystkich słów postaci $abab \ldots ab$, oznaczany $(ab)^* \subseteq \{a,b\}^+$.

5. Zbiór wszystkich słów w nad alfabetem $\{a\}$ o parzystej długości, oznaczany $(aa)^*$.

6. Język składający się ze wszystkich słów w nad alfabetem $\{0,1\}$ t.że w jest binarnym zapisem[1] liczby parzystej, tj.

$$\{\text{bin}(2n) \mid n \in \mathbb{N}\} \subseteq \{0,1\}^* = \{0, 10, 100, 110, 1000, 1010, \ldots\}.$$

Można by ten język oznaczyć wyrażeniem

$$\{0\} \cup 1 \cdot \{0,1\}^* \cdot 0 \qquad \text{lub} \qquad 0 + 1(0+1)^*0.$$

Te notacje zostaną opisane poniżej.

7. Język składający się ze wszystkich słów w nad alfabetem $\{0,1\}$ t.że w jest binarnym zapisem liczby podzielnej przez trzy, tj.

$$\{\text{bin}(3n) \mid n \in \mathbb{N}\} \subseteq \{0,1\}^*.$$

8. Język składający się ze wszystkich słów w nad alfabetem $A = \{a, b\}$ które mają tyle samo liter[2] a co liter b, tj.

$$\{w \in A^* \mid \#_a(w) = \#_b(w)\}.$$

9. Język składający się ze wszystkich palindromów nad alfabetem A, tj. słów w które są równe swojemu lustrzanemu odbiciu[3],

$$\{w \in A^* \mid w = w^{\mathsf{R}}\}.$$

10. Język składający się ze wszystkich słów w nad alfabetem $\{0,1\}$ t.że w jest binarnym zapisem liczby kwadratowej, tj.

$$\{\text{bin}(n^2) \in A \mid n \in \mathbb{N}\}.$$

11. Język składający się ze wszystkich słów w nad alfabetem $\{0,1\}$ takich, że w jest binarnym zapisem liczby pierwszej, tj.

$$\{\text{bin}(n) \in A \mid n \in \mathbb{N} \text{ jest liczbą pierwszą}\}.$$

12. Zbiór wszystkich słów w nad alfabetem $\{0,1\}$ będących opisem[4] grafu G o wierzchołkach $\{1, \ldots, n\}$, dla pewnej liczby $n \in \mathbb{N}$.

13. Zbiór wszystkich słów w nad alfabetem $\{0,1\}$ opisujących graf G o wierzchołkach $\{1, \ldots, n\}$ który zawiera ścieżką eulerowską, t.j. ścieżkę przechodzącą przez każdą krawędź grafu dokładnie raz. Jest to podzbiór języka z poprzedniego punktu.

14. Zbiór wszystkich słów w nad alfabetem $\{0,1\}$ opisujących macierz incydencji grafu G o wierzchołkach $\{1, \ldots, n\}$ który zawiera ścieżką hamiltonowską, t.j. ścieżkę przechodzącą przez każdy wierzchołek grafu dokładnie raz.

[1] Dla $n \in \mathbb{N}$, przez $\text{bin}(n)$ oznaczamy słowo nad alfabetem $\{0,1\}$ będące binarną reprezentacją liczby n bez wiodących zer, z tym że $\text{bin}(0) = 0$.

[2] przez $\#_x(w)$ oznaczamy liczbę wystąpień litery x w słowie w

[3] przez w^{R} oznaczamy słowo w czytane od tyłu, np. $(bbaa)^{\mathsf{R}} = aabb$

Każdy zna ten ciąg: 10, 11, 101, 111, 1011, 1101, 10001, 10011, 10111, 11101, 11111, 100101

[4] Graf G o wierzchołkach $V = \{1, \ldots, n\}$ oraz zbiorze krawędzi E jest opisany słowem długości n^2 które na pozycji $n \cdot (i-1) + j$ ma cyfrę 1 jeżeli wierzchołki i oraz j są połączone krawędzią, oraz 0 w przeciwnym przypadku, dla $1 \leqslant i, j \leqslant n$ (zakładamy przy tym, że żaden wierzchołek nie jest połączony krawędzią ze sobą samym). Np. słowo 0110 opisuje graf o dwóch wierzchołkach i jednej krawędzi, a słowo 011101110 opisuje trójkąt.

15. Zbiór wszystkich słów w nad alfabetem $\{0,1\}$ opisujących macierz incydencji grafu G którego wierzchołki da się pokolorować trzema kolorami w taki sposób, by każde dwa sąsiadujące wierzchołki miały ten sam kolor.

16. Język Posta[5] nad alfabetem $\{a,b,\#\}$, składający się ze wszystkich słów postaci
$$u_1 \# v_1 \# u_2 \# v_2 \# \ldots \# u_k \# v_k,$$
gdzie $k \geqslant 1$ oraz $u_1, v_1, \ldots, u_k, v_k \in \{a,b\}^*$, dla których istnieje taki skończony ciąg liczb $i_1, \ldots, i_\ell \in \{1,\ldots,k\}$, gdzie $\ell \geqslant 1$, że[6] $u_{i_1} u_{i_2} \cdots u_{i_\ell} = v_{i_1} v_{i_2} \cdots v_{i_\ell}$.

[5] Emil Post (1897-1954) był matematykiem polsko-żydowskim pochodzącym z Augustowa.

[6] Przykładowo, rozważmy następujące trzy pary słów (u_i, v_i), ułożone w następującą macierz:

i	1	2	3
u_i	a	ab	bba
v_i	baa	aa	bb

Ten układ reprezentowany jest przez słowo $a\#baa\#ab\#aa\#bba\#bb$. Słowo to należy do języka Posta, o czym zaświadcza ciąg $3,2,3,1$: konkatenacja słów u_3, u_2, u_3, u_1 jest równa konkatenacji słów v_3, v_2, v_3, v_1. Można to zilustrować następująco:

3	2	3	1
bba	ab	bba	a
bb	aa	bb	baa

gdzie każda kolumna jest jedną z kolumn powyższej macierzy, oraz obydwa wiersze mają równe konkatenacje.

Kluczową obserwacją w informatyce teoretycznej jest to, że języki różnią się *złożonością* – są pewne miary pozwalające stwierdzić, że jeden język jest bardziej "skomplikowany" niż inny język. Intuicyjnie, język L jest skomplikowany jeżeli stwierdzenie, czy dane słowo należy do L wymaga dużo obliczeń.

W tym wykładzie, rozpoczniemy od badania najprostszych języków, zwanych *językami regularnymi*. Stwierdzenie przynależności słowa w do języka regularnego, w pewnym uproszczeniu, nie wymaga *żadnych* rachunków, jedynie wymaga przeczytania słowa w od lewej do prawej, trzymając w pamięci jedynie bardzo ograniczoną ilość informacji, coś w rodzaju "przeniesienia" w algorytmie dodawania. Są to na przykład języki w punktach 1-7. Przykładowo, żeby stwierdzić czy dana liczba (zapisana binarnie) jest podzielna przez dwa wystarczy spojrzeć na ostatnią cyfrę tej liczby. Na pierwszy rzut oka, mogłoby się wydawać, że stwierdzenie podzielności przez trzy liczby zapisanej w zapisie binarnym wymaga rachunków bardziej skomplikowanych niż stwierdzenie, czy w słowie jest tyle samo liter a co b. Jednak, jak zobaczymy, te rachunki są bardzo proste – wymagają jedynie przenoszenia informacji która jest liczbą z zakresu $\{0,1,2\}$. Z kolei porównanie liczby wystąpień liter a i b wymaga przenoszenia informacji potencjalnie nieograniczonej. Istotnie, zobaczymy, że język z punktu 8 nie jest regularny.

W Rozdziale 2 zdefiniujemy języki regularne i rozwiniemy podstawową ich teorię. W Rozdziale 3 będziemy badali języki *bezkontekstowe*. Są to na przykład języki w punktach 8 i 9, ale już nie w kolejnych punktach. W Rozdziale 4 będziemy badali maszyny Turinga oraz definiowane przez nie języki, i wprowadzimy różne miary ich złożoności. Okaże się, że języki w punktach 10, 11, 12, 13 są podobnej trudności, języki 14, 15 są też tej samej trudności, ale przypuszczalnie dużo trudniejsze niż wcześniejsze języki, choć nikt jak do tej pory nie potrafił tego udowodnić. Wreszcie, język w punkcie 16 jest zdecydowanie najtrudniejszy ze wszystkich podanych języków.

Jak zobaczymy, ten język jest tak trudny, że nie da się napisać programu komputerowego który by poprawnie stwierdzał, czy dane słowo w należy do języka Posta. Żeby to udowodnić, trzeba będzie wpierw sformalizować pojęcie programu komputerowego, i właśnie temu służą maszyny Turinga.

2
Języki regularne

Języki regularne stanowią najprostsze rodzaje języków. Mogą być zdefiniowane na kilka równoważnych sposobów. Poznamy trzy zasadnicze definicje: za pomocą automatów, za pomocą wyrażeń regularnych, oraz abstrakcyjną charakteryzację. Zaczniemy od "konkretnych" definicji języków regularnych, potem poznamy ich rozmaite własności oraz definicję abstrakcyjną.

Przypomina to sytuację w algebrze liniowej, gdzie podprzestrzenie liniowe w $\mathbb{R}^n$ można definiować na trzy równoważne sposoby: jako zbiór rozwiązań układu równań liniowych, jako zbiór wszystkich kombinacji liniowych danego skończonego zbioru wektorów, i abstrakcyjnie, jako zbiór wektorów zamknięty na dodawanie i mnożenie przez skalary. Podobnie jak w przypadku przestrzeni liniowych, każda z definicji języków regularnych ma swoje zalety.

Motywacją do rozważania języków regularnych jest możliwość opisywania nieskończonych zbiorów słów za w skończony sposób. Jest to przydatne np. przy pisaniu parserów (dokładniej, lekserów lub tokenizerów). Przykładowo, w formularzu na stronie możemy wymagać, by dane pole było wypełnione adresem e-mail, składającym się z ciągu znaków postaci

$$\mathit{Letter}(\mathit{Letter} + \mathit{Digit})^* \text{ @ } \mathit{Letter}(\mathit{Letter} + \mathit{Digit} + \text{ “.”})^*$$

gdzie *Letter* to zbiór liter w alfabecie łacińskim oraz *Digit* to zbiór cyfr, oraz * oznacza wielokrotną iterację, a $+$ oznacza alternatywę. Powyższy napis jest prostym przykładem *wyrażenia regularnego*, które zdefiniujemy poniżej. W rzeczywistości, poprawne adresy e-mail mogą mieć dużo bardziej skomplikowaną postać i są opisywane bardzo skomplikowanymi wyrażeniami regularnymi[1]. Co więcej, składnia wyrażeń regularnych używana w programach komputerowych jest o wiele bardziej rozbudowana niż ta podstawowa, którą wprowadzimy poniżej, choć obie składnie są w stanie opisać te same języki.

Za pomocą wyrażeń regularnych możemy więc w zwięzły sposób opisywać wymaganą przez nas składnię i co więcej, te opisy reprezentować w programach komputerowych (parserach) które mają sprawdzać zgodność wprowadzanego tekstu z naszą składnią. Jak zobaczymy później, używając *automatów* można łatwo napisać program który sprawdza, czy dany tekst jest zgodny ze składnią opisaną przez wyrażenie regularne.

[1] Wyrażenie regularne opisujące poprawne adresy mejlowe według oficjalnego standardu RFC 5322 to
\A(?:[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])? | \[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3} (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]: (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f] | \\[\x01-\x09\x0b\x0c\x0e-\x7f])+) \])\z

2.1 *Wyrażenia regularne*

Definiujemy następujące podstawowe operacje na językach:

Suma języków K i L, oznaczana $K + L$ i zdefiniowana jako teoriomnogościowa suma $K \cup L$.

Konkatenacja języków K i L, oznaczana $K \cdot L$ lub po prostu KL i zdefiniowana jako $\{v \cdot w \mid v \in K, w \in L\}$.

Gwiazdka Kleene'go języka L, oznaczana L^* i zdefiniowana jako $\bigcup_{n \geqslant 0} L^n$, gdzie $L^n = \underbrace{L \cdot L \cdots L \cdot L}_{n}$ dla $n \geqslant 1$ oraz $L^0 = \{\varepsilon\}$.

Inaczej mówiąc, słowo w należy do języka L^* wtedy, i tylko wtedy, gdy można je podzielić na pewną liczbę części $w_1, \ldots, w_n$, z których każda należy do języka L, jak zilustrowano poniżej:

$$\underbrace{w}_{\in L^*} \quad = \quad \underbrace{w_1}_{\in L} \cdot \underbrace{w_2}_{\in L} \cdots \underbrace{w_n}_{\in L}.$$

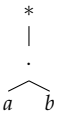
Przykładowo, jeżeli $L = \{a^*b\}$, to $abaabaaabaaaab \in L^*$. Zauważmy, że słowo puste ε zawsze należy do języka L^*.

Korzystając z powyższych trzech operacji, języka pustego $\emptyset$ i języków singletonowych postaci $\{a\}$, gdzie a jest literą alfabetu A, możemy konstruować bardziej skomplikowane języki. Nieco bardziej precyzyjnie[2], *wyrażenie regularne* jest to wyrażenie zbudowane rekurencyjnie, za pomocą operacji $K + L$, $K \cdot L$ oraz L^*, oraz z poszczególnych liter alfabetu (traktowanych jako języków singletonowych) i języka pustego $\emptyset$ (dodatkowo dopuszczamy nawiasy). Wyrażeniem regularnym jest więc przykładowo $(\{a\} \cdot \{b\})^* + (\{b\} \cdot \{a\})^*$, przy czym klamry wokół języków singletonowych są zawsze pomijane, więc to wyrażenie zapisujemy jako $(a \cdot b)^* + (b \cdot a)^*$ lub po prostu $(ab)^* + (ba)^*$.

W naturalny sposób, z każdym wyrażeniem regularnym $\mathcal{E}$ związany jest pewien język, oznaczany $L(\mathcal{E})$. Formalnie, definicja przebiega przez indukcję po budowie wyrażenia[3]. Poniżej jest kilka wyrażeń regularnych oraz związanych z nimi języków. Często pomijamy nawiasy, oraz symbol $\cdot$ w konkatenacji, gdy nie prowadzi to do nieporozumień. Pomijamy również klamry $\{\}$ przy definiowaniu języków singletonowych.

- $(ab)^*$ – zbiór wszystkich słów postaci $abab\ldots ab$, tj. $L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\}$.
- $\emptyset^*$ – język składający się tylko z ε, tj. $L(\emptyset^*) = \{\varepsilon\}$.
- $((a + b) \cdot (a + b))^*$ – zbiór słów parzystej długości nad alfabetem $\{a, b\}$.

[2] Formalnie, wyrażenia regularne należałoby przedstawiać jako *drzewa*, takie jak to:



To drzewo w tekście reprezentujemy jako $(a \cdot b)^*$. Tak więc nawiasy są tylko elementem składniowym służącym tekstowej reprezentacji drzew. Więcej o drzewach będzie w Rozdziale 2.6 oraz przy okazji omawiania gramatyk bezkontekstowych, w Rozdziale 3.

[3] Pełna definicja indukcyjna jest taka:

$$\begin{aligned} L(\mathcal{E} \cdot \mathcal{F}) &= L(\mathcal{E}) \cdot L(\mathcal{F}), \\ L(\mathcal{E}^*) &= L(\mathcal{E})^*, \\ L(\mathcal{E} + \mathcal{F}) &= L(\mathcal{E}) \cup L(\mathcal{F}), \\ L(a) &= \{a\}, \\ L(\emptyset) &= \emptyset. \end{aligned}$$

- $0 + 1 \cdot (0 + 1)^* \cdot 0$ – zbiór binarnych kodów liczb parzystych.

- $(0 + 1)^* \cdot 0 + ((0 + 1)^* \cdot 1)^* + \varepsilon$ – zbiór wszystkich słów nad alfabetem $\{0, 1\}$, zapisany w skomplikowany sposób.

- $(a + b)^* babaa\,(a + b)^*$ – zbiór wszystkich słów nad alfabetem $\{a, b\}$ zawierających infiks *babaa*.

Ćwiczenie. Napisać wyrażenie regularne opisujące zbiór binarnych kodów liczb podzielnych przez 3.

Mając dane wyrażenie regularne $\mathcal{E}$ oraz słowo w, chcielibyśmy za pomocą algorytmu stwierdzić, czy $w \in L(\mathcal{E})$. Nie jest oczywiste jak to zrobić efektywnie. Do tego nam posłużą automaty, omówione w następnym wykładzie.

2.2 *Automaty*

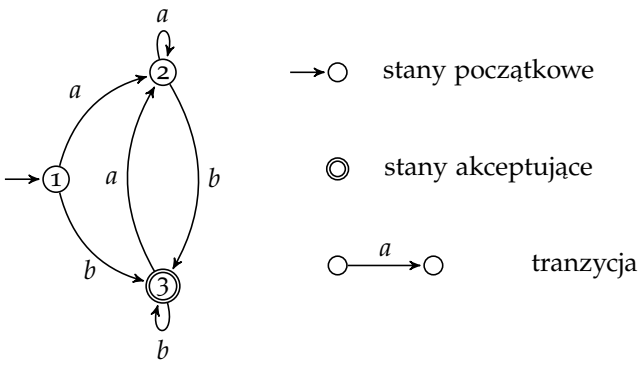
WYKŁAD 2

Wyrażenia regularne są wygodnym narzędziem do reprezentowania niektórych nieskończonych języków w skończony sposób. Jednak praktyczne manipulacje na takich wyrażeniach są dosyć uciążliwe. Przykładowo, mając dane wyrażenie regularne $\mathcal{E}$ oraz złowo w, chcielibyśmy mieć algorytm stwierdzający, czy $w \in L(\mathcal{E})$. Nie widać od razu, jak to efektywnie zrobić[4]. Jak zobaczymy, korzystając z *automatów*, można to sprawdzić w czasie liniowym ze względu na długość słowa w. Będziemy rozważać dwa rodzaje automatów: deterministyczne oraz ogólniejsze, niedeterministyczne. Każdy wariant ma pewne swoje przewagi nad drugim, umożliwiając efektywne wykonywanie różnych operacji. Zobaczymy, że wszystkie modele – wyrażenia regularne, automaty deterministyczne i niedeterministyczne – są sobie równoważne, tj. definiują te same języki – *języki regularne*.

[4] Definicja wyrażeń regularnych nasuwa na myśl następujący algorytm: jeżeli $\mathcal{E}$ jest postaci $\mathcal{E}_1 \cdot \mathcal{E}_2$, to sprawdzamy wszystkie możliwe podziały słowa w na dwie części, $w = w_1 \cdot w_2$, i rekurencyjnie sprawdzamy, czy $w_1 \in L(\mathcal{E}_1)$ oraz $w_2 \in L(\mathcal{E}_2)$; jeżeli $\mathcal{E}$ jest postaci $\mathcal{F}^*$, to sprawdzamy wszystkie możliwe podziały słowa w na dowolną liczbę części, $w = w_1 \cdots w_n$, i dla każdego w_i sprawdzamy, czy $w_i \in L(\mathcal{F})$. To daje algorytm działający w czasie wykładniczym względem długości słowa. Można ten czas zbić do wielomianowego, używając podejścia dynamicznego:: tablicujemy dla każdego infiksu u słowa w informację, czy $u \in L(\mathcal{F})$.

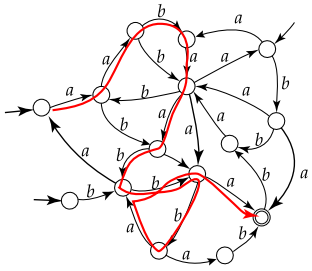
2.2.1 *Automaty niedeterministyczne*

Automat niedeterministyczny (zwany też NFA, od angielskiego *nondeterministic finite automaton*) jest to model prostego urządzenia o skończonej liczbie stanów, reagującego na "bodźce", będące literami alfabetu A. Będąc w ustalonym stanie, w reakcji na dany bodziec, automat dokonuje tranzycji w inny stan. Wyróżnione są ponadto dwa zbiory stanów (niekoniecznie rozłączne): stany początkowe oraz stany akceptujące. Automat często przedstawiamy za pomocą diagramu, jak poniżej.



Przedstawiony automat ma alfabet $A = \{a, b\}$, zbiór stanów $Q = \{1, 2, 3\}$, zbiór stanów początkowych $I = \{1\}$, zbiór stanów akceptujących $F = \{3\}$, oraz tranzycje

$$1 \xrightarrow{a} 2, \qquad 1 \xrightarrow{b} 3,$$
$$2 \xrightarrow{a} 2, \qquad 2 \xrightarrow{b} 3,$$
$$3 \xrightarrow{a} 2, \qquad 3 \xrightarrow{b} 3.$$



Bieg akceptujący automatu po słowie *aabaabbbaba*.

Formalnie, automat składa się z:

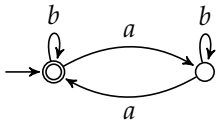
- alfabetu A,
- skończonego zbioru Q, nazywanego zbiorem *stanów*,
- zbioru $I \subseteq Q$ stanów *początkowych*,
- zbioru $F \subseteq Q$ stanów *akceptujących*,
- relacji $\delta \subseteq Q \times A \times Q$, zwanej *relacją przejścia*.

Trójkę (p, a, q) należącą do relacji przejścia δ nazywamy *tranzycją* automatu. Taka tranzycja odpowiada krawędzi w automacie, i jest oznaczana $p \xrightarrow{a} q$. *Bieg automatu* to ścieżka w diagramie automatu, t.j. ciąg tranzycji

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} q_n.$$

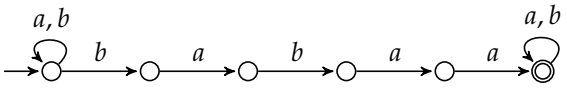
Mówimy, że bieg jak powyżej jest biegiem po słowie $w = a_1 a_2 \dots a_n$, ze stanu q_0 do stanu q_n. Jeżeli istnieje bieg ze stanu p do stanu q po słowie w, to piszemy $p \xrightarrow{w} q$. Bieg jest *akceptujący* jeżeli zaczyna się w stanie początkowym, a kończy się w stanie akceptującym. Automat niedeterministyczny $\mathcal{A}$ *akceptuje* słowo $w \in A^*$ jeżeli istnieje jego bieg akceptujący po tym słowie. Inaczej mówiąc, istnieje ścieżka ze stanu początkowego do stanu akceptującego, której kolejne etykiety tworzą słowo w. Przez $L(\mathcal{A}) \subseteq A^*$ oznaczamy zbiór wszystkich słów które są akceptowane przez automat $\mathcal{A}$. Mówimy też, że $\mathcal{A}$ *rozpoznaje* język $L(\mathcal{A})$.

Przykład 1. Poniższy automat $\mathcal{B}$ akceptuje słowa nad alfabetem $\{a, b\}$, które mają parzystą liczbę liter a.



⌟

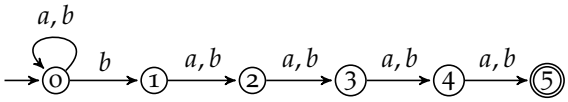
Przykład 2. Poniżej, narysowany jest automat niedeterministyczny $\mathcal{C}$ rozpoznający język tych słów nad alfabetem $\{a, b\}$, które zawierają słowo *babaa*. Innymi słowy, $L(\mathcal{C}) = \{a, b\}^*\{babaa\}\{a, b\}^*$.



Dla danego słowa w, automat może mieć kilka biegów zaczynających się w stanie początkowym. Przykładowo, są dwa takie biegi po słowie *babaa*. Słowo jest akceptowane przez automat jeżeli *któryś* z biegów po tym słowie kończy w stanie akceptującym.

Jasne jest, że jeżeli ρ jest biegiem akceptującym automatu $\mathcal{A}$ po słowie w, to w musi zawierać infiks *babaa*. Z drugiej strony, jeżeli słowo w zawiera infiks *babaa*, to istnieje bieg akceptujący automatu $\mathcal{A}$ po słowie w (choć istnieją też biegi nieakceptujące). ⌟

Przykład 3. Rozważmy następujący automat niedeterministyczny $\mathcal{K}$ nad alfabetem $A = \{a, b\}$.



Poniżej podamy intuicyjny argument, że ten automat rozpoznaje język $L = A^*bA^4$, składający się z słów w których piąta litera od końca to litera b. Ten argument odpowiada częstemu sposobowi *myślenia* o automatach niedeterministycznych; nie jest to formalny dowód[5].

Dla danego słowa wejściowego w, przy każdej wczytanej literze b, automat $\mathcal{K}$ ma możliwość pozostania w stanie 0, lub przejścia do stanu 1. Jeśli automat "zdecyduje się[6]" przejść do stanu 1 w i-tym kroku, to później nie ma już wyjścia – musi przy każdej kolejnej wczytanej literze zwiększyć numer stanu o jeden. Jeżeli wczytane zostaną więcej niż cztery litery, to automat "psuje się", tzn. nie może dokonać żadnego przejścia. Jeżeli wczytane zostaną mniej niż cztery litery, to automat nie osiągnie stanu akceptującego. Po wczytaniu całego słowa wejściowego, automat znajdzie się stanie akceptującym wtedy, i tylko wtedy, gdy piąta litera od końca to litera b.

Choć takie nieformalne rozumowanie może być pomocne przy konstrukcji automatu, jak widać, formalny argument jest zarówno bardziej precyzyjny, jak i zwięzły od nieformalnego opisu. Dlatego w zadaniach ograniczamy się zazwyczaj do podawania formalnych dowodów lub ich szkiców. ⌟

Przykład 4. Niech $\mathcal{A}_1, \mathcal{A}_2$ będą automatami niedeterministycznymi nad alfabetem A. Wtedy suma rozłączna automatów $\mathcal{A}_1$ i $\mathcal{A}_2$ jest takim automatem niedeterministycznym $\mathcal{B}$, że $L(\mathcal{B}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$. ⌟

[5] Tu jest szkic formalnego dowodu. Pokazujemy obie inkluzje równości $L(\mathcal{K}) = L$. Dla inkluzji z prawej w lewo, rozważamy słowo $a_1a_2 \dots a_n \in A^*$ takie, że $a_{n-4} = b$ i podajemy jawnie bieg akceptujący automatu $\mathcal{B}$ po tym słowie:

$$0 \xrightarrow{a_1} 0 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-5}} 0 \xrightarrow{a_{n-4}} 1 \xrightarrow{a_{n-3}} 2 \xrightarrow{a_{n-2}} 3 \xrightarrow{a_{n-1}} 4 \xrightarrow{a_n} 5.$$

Dla inkluzji z lewej w prawo, dowodzimy, że każdy bieg akceptujący jest powyższej postaci, oraz zauważamy, że jedyna tranzycja ze stanu 0 do stanu 1 w automacie jest po literze b, więc $a_{n-4} = b$.

[6] W sytuacji, gdy automat wczytał literę b i przeszedł do stanu 1, mówimy czasem, że automat "zgadł", że dana litera jest piątą literą od końca. Automat niedeterministyczny ma możliwość takiego "zgadywania", jednak musi potem sprawdzić, że "dobrze zgadł" – nasz automat akceptuje tylko wtedy, jeśli rzeczywiście przeszedł do stanu 1 w momencie napotkania piątej litery od końca.

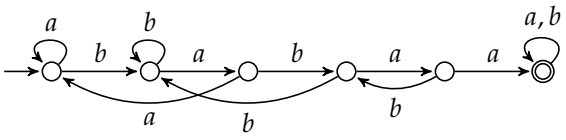
2.2.2 *Automaty deterministyczne*

Automat deterministyczny (DFA, od angielskiego *deterministic finite automaton*) to szczególny rodzaj automatów niedeterministycznych, które mają tylko jeden stan początkowy, i dla każdego stanu p i litery a jest dokładnie jeden taki stan q, że $p \xrightarrow{a} q$. Stan q jak powyżej oznaczamy czasem $\delta(p,a)$. Tak więc, δ definiuje funkcję $\delta\colon Q \times A \to Q$.

Zauważmy, że dla każdego słowa $w \in A^*$, automat deterministyczny ma dokładnie jeden bieg po tym słowie, zaczynający się w danym stanie p. Przez $p \cdot w$ oznaczamy stan, w którym ten jedyny bieg kończy[7].

[7] W szczególności, $p \cdot \varepsilon = p$ oraz $p \cdot (w\,a) = \delta((p \cdot w), a)$, dla $w \in A^*, a \in A$.
Dowodząc formalnie, że $L(\mathcal{C}')$ to język $L = L((a+b)^*babaa(a+b)^*)$, pokazalibyśmy następujący niezmiennik. Numerując stany automatu od lewej do prawej liczbami $0, 1, \ldots, 5$, dla każdego słowa $w \in \{a,b\}^* - L$, zachodzi własność: stan $0 \cdot w$ ma numer będący długością najdłuższego sufiksu słowa w, który jest prefiksem słowa *babaa*. Dowód przebiega przez indukcję po $|w|$.

Przykład 5. Poniższej narysowany jest deterministyczny automat automat $\mathcal{C}'$ akceptujący te same słowa, co automat z przykładu 2, tj. zbiór słów nad alfabetem $\{a,b\}$, które zawierają podsłowo *babaa*.



⌟

Przykład 6. Opiszemy teraz automat deterministyczny $\mathcal{D}$, który akceptuje dokładnie te słowa w nad alfabetem $\{0,1\}$, które opisują w zapisie binarnym liczbę podzielną przez 3. Jak zobaczymy, w tym przypadku, zamiast rysować automat, lepiej podać jego formalną definicję. Stany automatu $\mathcal{D}$ to $\{q_0, q_1, q_2\}$, reprezentujące reszty z dzielenia przez 3. Funkcja przejścia δ jest zdefiniowana następująco:

$$\delta(q_r, i) = q_k \qquad \textit{dla } r \in \{0,1,2\}, i \in \{0,1\} \textit{ oraz } k = (2 \cdot r + i) \mod 3.$$

Stan q_0 jest początkowy i akceptujący.

Pokażemy, że dla każdego słowa $w \in \{0,1\}^*$, zachodzi następujący niezmiennik, dla $w \in A^*$:

$$q_0 \cdot w = q_k \qquad \textit{gdzie } k = ([w]_2 \mod 3). \tag{2.1}$$

Dowodzimy równości (2.1) przez indukcję po długości słowa w. Baza indukcji jest trywialna, bo wtedy $w = \varepsilon$ i $q_0 \cdot w = q_0$.

W kroku indukcyjnym, przypuśćmy, że teza zachodzi dla słowa w i rozważmy słowo postaci $w\,i$, gdzie $i \in \{0,1\}$. Na mocy założenia indukcyjnego, $q_0 \cdot w = ([w]_2 \mod 3)$. Wówczas

$$\begin{aligned} q_0 \cdot (w\,i) = (q_0 \cdot w) \cdot i = \delta([w]_2 \mod 3, i) = q_k, \\ \text{gdzie } k = (2 \cdot ([w]_2 \mod 3) + i) \mod 3 = \\ (2 \cdot [w]_2 + i) \mod 3 = [w\,i]_2 \mod 3. \end{aligned}$$

To dowodzi tezy indukcyjnej, kończąc dowód równości (2.1).

Ponieważ stanem akceptującym jest stan q_0, z niezmiennika wynika, że automat $\mathcal{D}$ akceptuje słowo w wtedy, i tylko wtedy, gdy $[w]_2 \bmod 3 = 0$. ⌟

2.3 *Języki regularne*

Ustalmy alfabet A (jak zawsze, skończony). Udowodnimy następujące twierdzenie.

Twierdzenie 1. *Następujące warunki są równoważne dla języka $L \subseteq A^*$:*

1. *Język L jest generowany przez pewne wyrażenie regularne, tj. istnieje takie wyrażenie regularne $\mathcal{E}$, że $L = L(\mathcal{E})$.*
2. *Język L jest rozpoznawany przez pewien automat niedeterministyczny, tj. istnieje taki automat niedeterministyczny $\mathcal{A}$, że $L = L(\mathcal{A})$,*
3. *Język L jest rozpoznawany przez pewien automat deterministyczny, tj. istnieje taki automat deterministyczny $\mathcal{A}$, że $L = L(\mathcal{A})$.*

Język $L \subseteq A^*$ spełniający jeden (a więc każdy) z warunków w powyższym twierdzeniu nazywamy *językiem regularnym*.

Powiemy, że dwa automaty, bądź automat i wyrażenie regularne są sobie *równoważne*, jeżeli definiują one ten sam język.

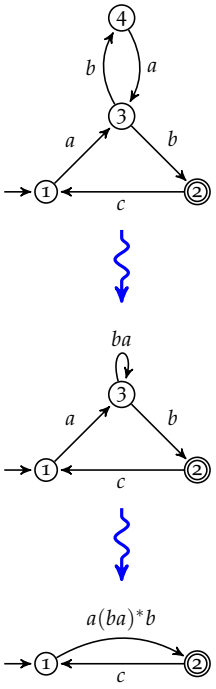
W Lemacie 3 pokażemy równoważność 2↔3, poprzez pokazanie że dla każdego automatu niedeterministycznego istnieje równoważny mu automat deterministyczny. W Lemacie 2 pokażemy implikację 1→2 – że dla każdego wyrażenia regularnego istnieje równoważny mu automat niedeterministyczny. W Lemacie 1 pokażemy odwrotną implikację 2→1 – że dla każdego automatu niedeterministycznego istnieje równoważne mu wyrażenie regularne.

2.3.1 *Wyrażenie z automatu*

W tej sekcji udowodnimy implikację $2 \to 1$ w twierdzeniu 1, tj. że dla każdego automatu niedeterministycznego istnieje równoważne mu wyrażenie regularne. Zanim pokażemy formalny dowód, opiszemy konstrukcję na prostym przykładzie.

Przykład 7. Rozważmy automat deterministyczny narysowany na marginesie. Dokonujemy serii przekształceń, w każdym kroku usuwając stan o największym numerze (ta kolejność jest przypadkowa).

W pierwszym kroku usuwamy stan 4. Ponieważ stan ten był połączony jedynie ze stanem 3 tranzycjami $3 \xrightarrow{b} 4$ oraz $4 \xrightarrow{a} 3$, kompensujemy utratę stanu 4 przez dodanie tranzycji $3 \xrightarrow{ba} 3$. Otrzymany



obiekt nie jest już automatem niedeterministycznym, bo tam etykiety mogą być tylko pojedynczymi literami. Taki automat, w którym tranzycje mogą mieć etykiety będące słowami, lub nawet wyrażeniami regularnymi, nazwiemy *automatem uogólnionym*. Kluczowa własność zachowana przy powyższym przekształceniu jest taka, że nowy automat "odczytuje" takie same słowa wzdłuż biegów akceptujących, jak stary automat – każdy bieg akceptujacy przechodządzy przez stan 4 musiał mieć postać $1 \xrightarrow{\varepsilon} \cdots 3 \xrightarrow{b} 4 \xrightarrow{a} 3 \xrightarrow{\varepsilon} 2$, a to odpowiada biegowi $1 \xrightarrow{\varepsilon} \cdots 3 \xrightarrow{ba} 3 \xrightarrow{\varepsilon} 2$ w nowym automacie w tym sensie, że słowa odczytane z etykiet tych biegów są identyczne.

W drugim kroku, usuwamy stan 3. Jakie biegi zostały utracone po usunięciu tego stanu? Stan 3 był połączony z innymi stanami tranzycjami $1 \xrightarrow{a} 3$ oraz $3 \xrightarrow{b} 2$. Tak więc, każdy bieg przechodzący przez stan 3 przychodził ze stanu 1 i wchodził do stanu 2, a w międzyczasie odczytywał słowo postaci $a(ba)^*b$. Tak więc, w nowym automacie uogólnionym skompensujemy utratę stanu 3 jeśli stworzymy tranzycję ze stanu 1 do stanu 2 o etykiecie $a(ba)^*b$.

Z otrzymanego automatu o dwóch stanach łatwo odczytujemy wyrażenie regularne opisujące język słów akceptowanych:

$$(a(ba)^*bc)^*a(ba)^*b.$$

Konstrukcja w ogólnym przypadku jest bardzo podobna do powyższej. Ogólnie, gdy usuwamy stan q z tranzycją $q \xrightarrow{\mathcal{K}} q$, to musimy rozważyć wszystkie takie pary tranzycji $p \xrightarrow{\mathcal{E}} q$ oraz $q \xrightarrow{\mathcal{F}} r$ i dla każdej takiej pary stworzyć nową tranzycję $p \xrightarrow{\mathcal{E}\cdot\mathcal{K}^*\mathcal{F}} r$. Po drugie, by uniknąć komplikacji w sytuacji, gdy chcemy usunąć stan początkowy lub akceptujący, łatwiej założyć bez utraty ogólności, że jest tylko jeden stan początkowy i tylko jeden stan akceptujący, i że tych nigdy nie usuwamy. Dokładny dowód podany jest poniżej. ⌟

Uogólnione automaty niedeterministyczny Rozważamy *uogólnione automaty niedeterministyczne*, w których przejścia mogą być etykietowane dowolnymi wyrażeniami regularnymi nad alfabetem A, tzn. są postaci $p \xrightarrow{\mathcal{E}} q$, gdzie $\mathcal{E}$ to dowolne wyrażenie nad alfabetem A. *Bieg* takiego automatu to ciąg tranzycji

$$q_0 \xrightarrow{\mathcal{E}_1} q_1 \xrightarrow{\mathcal{E}_2} q_2 \xrightarrow{\mathcal{E}_3} \cdots \xrightarrow{\mathcal{E}_n} q_n.$$

Formalnie, relacja przejścia uogólnionego automatu niedeterministycznego to dowolna skończona relacja $\delta \subseteq Q \times R_A \times Q$, gdzie R_A to zbiór wyrażeń regularnych nad alfabetem A.

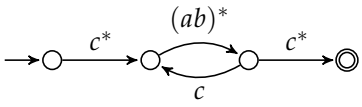
Powiemy, że jest to biego po słowie $w \in A^*$ jeżeli $w \in L(\mathcal{E}_1 \cdot \mathcal{E}_2 \cdots \mathcal{E}_n)$. Zauważmy, że jeden bieg może być po wielu różnych słowach. Podobnie jak poprzednio definiujemy język automatu uogólnionego.

Zauważmy, że dwie tranzycje $p \xrightarrow{\mathcal{E}} q$ i $p \xrightarrow{\mathcal{F}} q$ możemy (lecz nie musimy) zastąpić pojedynczą $p \xrightarrow{\mathcal{E}+\mathcal{F}} q$. Ponadto, możemy zakładać,

że etykiety tranzycji są różne od wyrażenia $\emptyset$ opisującego zbiór pusty (takie tranzycje możemy bez szkody usunąć).

Zwykły automat niedeterministyczny jest szczególnym przypadkiem uogólnionego automatu, w którym w tranzycjach pojawiają się jedynie wyrażenia postaci a (gdzie $a \in A$) lub $\emptyset$ (lub ε dla automatu z ε-przejściami).

Przykład 8. Poniższy uogólniony automat rozpoznaje język $c^*((ab)^*c)^*c^*$ (równoważnie, $(c + ab)^*$).



⌟

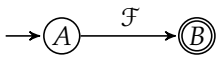
Następujący lemat w szczególności dowodzi implikacji $2 \to 1$ w twierdzeniu 1:

Lemat 1. *Niech $\mathcal{A}$ będzie automatem uogolnionym. Wówczas istnieje równoważne mu wyrażenie regularne $\mathcal{E}$.*

Dowód. Bez straty ogólności możemy założyć, że rozważany uogólniony automat niedeterministyczny $\mathcal{A}$ ma dokładnie jeden stan początkowy A i dokładnie jeden stan akceptujący B, oraz że do stanu A nie wchodzi żadna tranzycja, a ze stanu B nie wychodzi żadna tranzycja[8], oraz że $A \neq B$.

[8] Tworzymy nowy stan początkowy A i dodajemy tranzycje z etykietą ε z A do każdego stanu $q \in I$, gdzie I to zbiór stanów początkowych oryginalnego automatu. Podobnie, tworzymy nowy stan akceptujący B i dodajemy tranzycje z etykietą ε z każdego $q \in F$ do B, gdzie F to zbiór stanów akceptujących oryginalnego automatu.

Dowód lematu teraz przebiega przez indukcję po liczbie $n \geqslant 2$ stanów automatu $\mathcal{A}$, spełniającego powyższe dodatkowe założenia. Krok bazowy to gdy $n = 2$ oraz automat $\mathcal{A}$ ma jedynie stany A i B. W tym przypadku, automat ma następującą prostą postać:



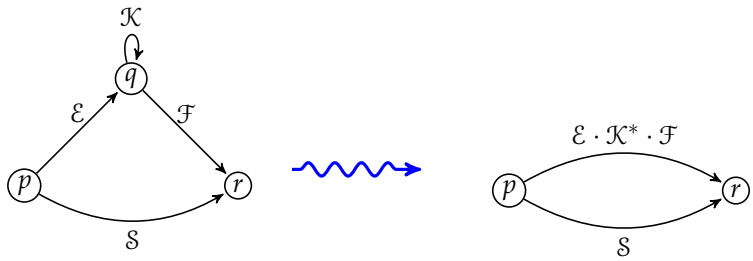
gdzie $\mathcal{F}$ jest pewnym wyrażeniem regularnym. Jasne jest, że język definiowany przez powyższy automat to dokładnie $L(\mathcal{F})$. Tak więc, $\mathcal{F}$ jest poszukiwanym wyrażeniem regularnym opisującym język definiowany przez oryginalny automat $\mathcal{A}$.

W kroku indukcyjnym, przypuśćmy, że $\mathcal{A}$ ma $n \geqslant 3$ stanów. Stworzymy automat uogólniony $\mathcal{B}$ o $n - 1$ stanach, który jest równoważny automatowi $\mathcal{A}$, więc teza wyniknie natychmiast z założenia indukcyjnego.

Automat $\mathcal{B}$ jest otrzymany z automatu $\mathcal{A}$ w wyniku następującej operacji. Wybieramy dowolny stan $q \in Q$, który nie jest stanem początkowym A ani stanem akceptującym B. Automat $\mathcal{B}$ otrzymujemy z automatu $\mathcal{A}$ poprzez usunięcie stanu q, i modyfikując tranzycje następująco. Przypuśćmy, że w automacie $\mathcal{A}$ jest dokładniej jedna tranzycja z q do q, postaci $q \xrightarrow{\mathcal{K}} q$ (jeśli nie ma żadnej, to bierzemy $\mathcal{K} = \emptyset$). Dla każdej pary tranzycji $p \xrightarrow{\mathcal{E}} q$ oraz $q \xrightarrow{\mathcal{F}} r$, gdzie p, r są różne od q, w automacie $\mathcal{B}$ tworzymy dodatkową tranzycję $p \xrightarrow{\mathcal{E}\cdot\mathcal{K}^*\cdot\mathcal{F}} r$. Przypomnijmy, że $\emptyset^* = \{\varepsilon\}$, więc jeśli w automacie

$\mathcal{A}$ nie ma żadnej tranzycji $q \xrightarrow{\mathcal{K}} q$ (lub równoważnie, jest tranzycja $q \xrightarrow{\emptyset} q$), to nowa etykieta to $\mathcal{E} \cdot \mathcal{F}$.

Bardziej obrazowo, dokonujemy następującej transformacji dla każdej pary tranzycji wchodzących i wychodzących z q:



Możemy dodatkowo zastąpić dwie równoległe tranzycje z p do r pojednynczą, by zredukować liczbę tranzycji w automacie.

Twierdzimy, że wynikowy automat $\mathcal{B}$ jest równoważny automatowi $\mathcal{A}$, tzn. $L(\mathcal{A}) = L(\mathcal{B})$. Dowód tego faktu polega na rozważeniu dowolnego biegu w automacie $\mathcal{A}$, i pocięciu go na kawałki, względem wystąpień stanu q.

Pokażemy inkluzję $L(\mathcal{A}) \subseteq L(\mathcal{B})$. Rozważmy dowolny bieg akceptujący ρ automatu $\mathcal{A}$. Możemy ten bieg przedstawić jako konkatenację biegów

$$\rho_0, \sigma_0, \rho_1, \sigma_1, \ldots, \sigma_{k-1}, \rho_k,$$

gdzie biegi $\rho_0, \ldots, \rho_k$ nie mają wystąpień stanu q, a każdy bieg σ_i jest postaci $p \xrightarrow{\mathcal{E}} q \xrightarrow{\mathcal{K}} q \xrightarrow{\mathcal{K}} q \cdots q \xrightarrow{\mathcal{K}} q \xrightarrow{\mathcal{F}} r$, gdzie p, r są różne od q, oraz $p \xrightarrow{\mathcal{E}} q$ i $q \xrightarrow{\mathcal{F}} r$ są pewnymi tranzycjami automatu $\mathcal{A}$. Bieg σ_i zastępujemy tranzycją $\sigma_i' = p \xrightarrow{\mathcal{E} \cdot \mathcal{K}^* \cdot \mathcal{E}} r$ automatu $\mathcal{B}$. Niech ρ' będzie konkatenacją biegów

$$\rho_0, \sigma_0', \rho_1, \sigma_1', \ldots, \sigma_{k-1}', \rho_k.$$

Wtedy ρ' jest biegiem akceptującym automatu $\mathcal{B}$. Ponadto, jeżeli bieg ρ jest po słowie w, to bieg ρ' też jest po słowie w. To dowodzi inkluzji $L(\mathcal{A}) \subseteq L(\mathcal{B})$. Dowód drugiej inkluzji jest podobny. ∎

2.3.2 *Automat z wyrażenia*

WYKŁAD 3

W tej sekcji pokażemy, jak z danego wyrażenia regularnego skonstruować równoważny mu automat niedeterministycznych. W tym celu, wygodnie będzie wpierw rozszerzyć nieco definicję automatów, wyposażając je w tzw. *ε-przejścia*. Są to tranzycja skierowane postaci $p \xrightarrow{\varepsilon} q$, którą automat może wykonać bez wczytywania litery słowa wejściowego. Tak więc, automat *niedeterministyczny z ε-przejściami* $\mathcal{A}$ ma relację przejścia $\delta \subseteq Q \times (A \cup \{\varepsilon\}) \times Q$. W takim automacie, *bieg*

to ciąg tranzycji

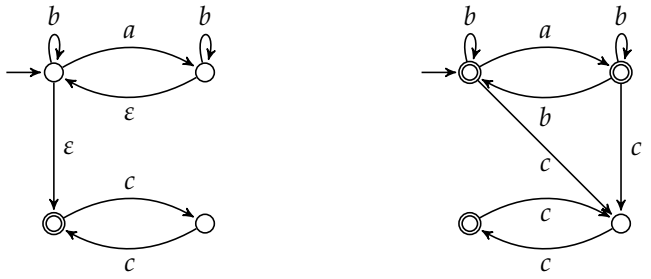
$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} q_n,$$

w którym $a_i \in A \cup \{\varepsilon\}$. Jest to bieg po słowie $a_1 \cdot a_2 \cdots a_n \in A^*$, w którym symbole $a_i = \varepsilon$ są pomijane. Jak poprzednio, bieg akceptujący to bieg, który zaczyna się w stanie początkowym i kończy w stanie akceptującym, automat $\mathcal{A}$ akceptuje słowo w jeśli ma po nim bieg akceptujący, oraz $L(\mathcal{A})$ to zbiór wszystkich takich słów w.

Nietrudno pokazać, że każdy automat niedeterministyczny z ε-przejściami $\mathcal{A}$ można zastąpić równoważnym mu automatem $\mathcal{B}$ niedeterministycznym bez ε-przejść o tym samym zbiorze stanów[9]. Proces ten nazywamy *eliminacją ε-przejść*.

[9] Nowy automat ma ten sam zbiór stanów Q co $\mathcal{A}$, tranzycje $p \xrightarrow{a} q$, dla wszystkich $p, q \in Q$ oraz $a \in A$ takich, że $\mathcal{A}$ ma bieg ze stanu p do stanu q którego etykiety to ciąg symboli ε zakończony pojedynczym symbolem a. Stany początkowe to stany początkowe automatu $\mathcal{A}$. Stany akceptujące to te stany p, że automat $\mathcal{A}$ ma bieg ze stanu p do pewnego swojego stanu akceptujęcego, którego etykiety to same ε.

Przykład 9. Automat niedeterministyczny z ε-przejściami narysowany poniżej z lewej akceptuje słowo *bacc*. Po prawej równoważny automat niedeterministyczny powstały przez eliminację ε-przejść.



⌟

By pokazać implikację $1 \to 2$ w twierdzeniu 1, wystarczy pokazać następujący lemat.

Lemat 2. *Dla każdego wyrażenia regularnego $\mathcal{E}$ istnieje równoważny mu automat niedeterministyczny z ε-przejściami $\mathcal{A}$.*

Dowód. Dowód postępuje przez indukcję po budowie wyrażenia $\mathcal{E}$. Krok bazowy, gdy wyrażenie $\mathcal{E}$ jest literą $a \in A$ lub językiem pustym $\emptyset$, jest trywialny. W kroku indukcyjnym wystarczy zastosować poniższą obserwację. Niech $K, L \subseteq A^*$ będą rozpoznawane przez automaty $\mathcal{A}$ i $\mathcal{B}$, odpowiednio. Wówczas:

- Istnieje automat $\mathcal{A} \cup \mathcal{B}$ rozpoznający język $K \cup L$. Ten automat to po prostu suma rozłączna automatów $\mathcal{A}$ i $\mathcal{B}$.
- Istnieje automat $\mathcal{A} \cdot \mathcal{B}$ rozpoznający język $K \cdot L$. Ten automat to suma rozłączna automatów $\mathcal{A}$ i $\mathcal{B}$, z tym że dla $p \in F_{\mathcal{A}}$ i $q \in I_{\mathcal{B}}$, dodajemy ε-przejście $p \xrightarrow{\varepsilon} q$, oraz p przestaje być akceptujący, a q przestaje być początkowy.
- Istnieje automat $\mathcal{A}^*$ rozpoznający język K^*. Ten automat to automat $\mathcal{A}$, z dodanym nowym stanem q_0 który jest nowym stanem

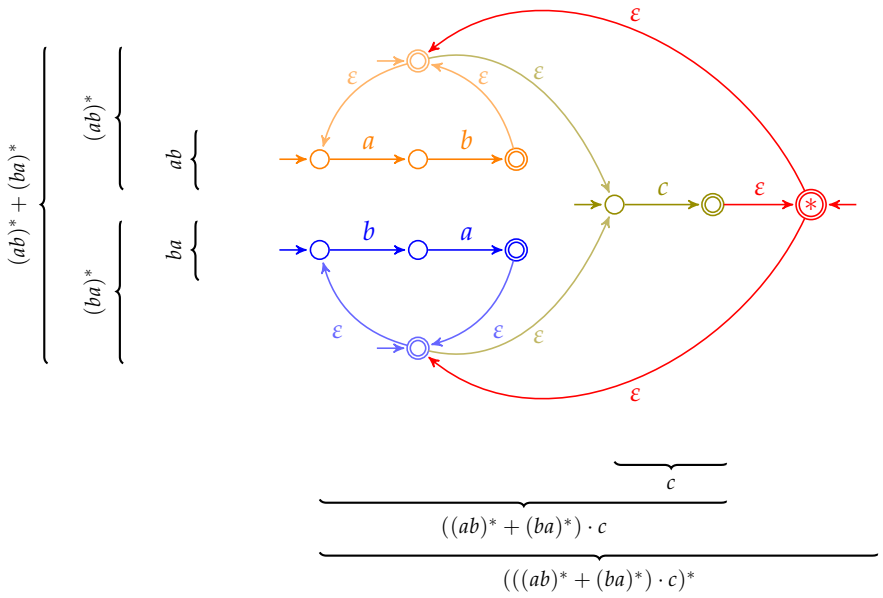
początkowym i akceptującym, oraz z dodanymi ε-przejściami z każdego stanu akceptującego stanu $\mathcal{A}$ do q_0 i z q_0 do każdego stanu początkowego automatu $\mathcal{A}$.

Zauważmy, że to daje algorytm który konstruuje automat niedeterministyczny z ε-przejściami z danego wyrażenia regularnego w czasie liniowym. ■

Przykład 10. Rozważmy następujące wyrażenie regularne:

$$(((ab)^* + (ba)^*) \cdot c)^*.$$

Kolory służą zaznaczaniu podwyrażeń, np. najbardziej zagnieżdżone są wyrażenia ab oraz ba. Dla powyższego wyrażenia, konstruujemy niedeterministyczny automat z ε-przejściami. Konstrukcja przebiega fazami, które są zaznaczone odpowiednimi kolorami. Zaczynamy od konstrukcji automatów dla wyrażeń ab oraz ba, a potem budujemy coraz to większe automaty. Ostateczny automat ma tylko jeden stan akceptujący i początkowy (oznaczony $*$).



Z otrzymanego automatu możemy dalej usunąć ε-przejścia, otrzymując automat niedeterministyczny. ⌟

2.3.3 *Determinizacja*

Oczywiście, każdy automat deterministyczny jest równoważny pewnemu automatowi niedeterministycznemu. Na odwrót też:

Lemat 3. *Dla każdego automatu niedeterministycznego $\mathcal{A}$ istnieje równoważny mu automat deterministyczny $\mathcal{B}$.*

Dowód. Automat $\mathcal{B}$ ma:

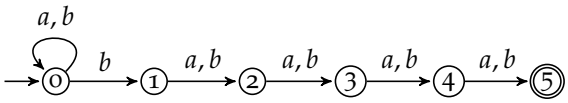
- zbiór stanów $P(Q)$, tj. stanami automatu $\mathcal{B}$ są wszystkie *zbiory* $S \subseteq Q$ stanów automatu $\mathcal{A}$,
- stan początkowy $q_0 = I$,
- zbiór stanów akceptujących $\{S \mid S \subseteq Q, S \cap F \neq \emptyset\}$, tj. stan $S \subseteq Q$ automatu $\mathcal{B}$ jest akceptujący wtedy, i tylko wtedy, gdy należy do niego stan akceptujący automatu $\mathcal{A}$,
- funkcję przejścia określoną wzorem:

$$\delta(S, a) = \{q \in Q \mid p \in S, p \xrightarrow{a} q \text{ w automacie } \mathcal{A}\},$$

dla $S \subseteq Q$ oraz $a \in A$.

Prosta indukcja po długości słowa $w \in A^*$ pokazuje, że w automacie $\mathcal{B}$ stan $q_0 \cdot w$ jest równy zbiorowi tych stanów q, które można osiągnąć w automacie $\mathcal{A}$ zaczynając w stanie początkowym i wczytując słowo w. W szczególności, stan $q_0 \cdot w$ jest akceptujący wtedy, i tylko wtedy, gdy automat $\mathcal{A}$ ma bieg akceptujący po słowie w. ∎

Przykład 11. Rozważmy ponownie automat $\mathcal{K}$ z przykładu 3:



Niech $\mathcal{B}$ będzie automatem skonstruowanym jak w dowodzie Lematu 3. Przykładowymi stanami tego automatu są zbiory

$$\{0,1,2\}, \{0,2,3\}, \{0,1,2,3\}.$$

Mamy też tranzycje:

$$\{0,1,2\} \xrightarrow{a} \{0,2,3\}$$
$$\{0,1,2\} \xrightarrow{b} \{0,1,2,3\}.$$

Automat $\mathcal{B}$ ma 2^6 stanów, ale wiele z nich jest nieosiągalnych: osiągalne są dokładnie te stany, do których należy stan 0. Tak więc, osiągalnych jest 2^5 stanów. Odpowiadają one podzbiorom zbioru $\{1,2,3,4,5\}$, a stanem początkowym jest $\emptyset$. Można pokazać, że po wczytaniu słowa w, automat $P(\mathcal{K})$ znajdzie się w stanie

$$X = \{i \in \{1,2,3,4,5\} \mid i\text{-ta litera od końca w słowie } w \text{ to } b\}.$$

Ćwiczenie. Pokazać, że nie istnieje automat deterministyczny rozpoznający język A^*bA^4 o mniej niż 2^5 stanach.

⌟

Automat $\mathcal{B}$ skonstruowany jak w dowodzie Lematu 3, z usuniętymi stanami nieosiągalnymi nazywany jest *automatem potęgowym*, i oznaczany $P(\mathcal{A})$. Obliczenie automatu $P(\mathcal{A})$ z automatu $\mathcal{A}$ nazywa się *determinizacją*. Zamiast wpierw obliczać automat $\mathcal{B}$, a potem zbiór stanów osiągalnych, łatwiej może być od razu obliczyć zbiór stanów osiągalnych[10].

[10] Zaczynamy od stanu początkowego I, a potem iteracyjnie, dla każdego obliczonego wcześniej stanu S, dorzucamy stany $\delta(S, a)$, dla $a \in A$.

Podsumowując, pokazaliśmy równoważność automatów deterministycznych, niedeterministycznych, uogólnionych niedeterministycznych, oraz wyrażeń regularnych. W Rozdziałach 2.7 oraz 2.8 (pominiętych na wykładzie), pokażemy równoważność z dwoma innymi formalizmami: logiką oraz półgrupami.

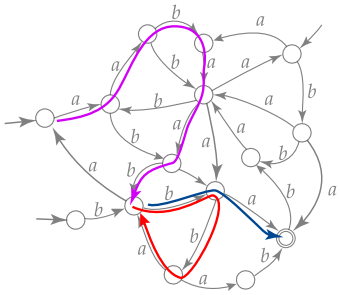
2.4 *Własności języków regularnych i algorytmy*

W większości, treść tego rozdziału jest omawiana podczas ćwiczeń.

W tym rozdziale opisane są różne narzędzia do manipulacji na automatach.

2.4.1 *Lemat o pompowaniu*

Pokażemy teraz wygodne narzędzie do pokazywania, że niektóre języki nie są regularne. Metoda opiera się na następującej obserwacji. Przypuśćmy, że język $L \subseteq A^*$ jest regularny. Wtedy istnieje automat niedeterministyczny $\mathcal{A}$ rozpoznający L; niech N oznacza jego liczbę stanów. Jeżeli $w \in L$ ma długość przynajmniej N, to w biegu akceptującym ρ musi się pojawić stan q, który występuje przynajmniej dwa razy (ponadto, taki stan q musi się pojawić już pośród pierwszych N stanów w biegu ρ). Możemy więc zdekomponować bieg ρ jako $\rho_1 \cdot \rho_2 \cdot \rho_3$, gdzie bieg ρ_2 zaczyna się i kończy w stanie q. Tej dekompozycji odpowiada dekompozycja słowa $w = w_1 \cdot w_2 \cdot w_3$, gdzie w_i to słowo utworzone z etykiet w biegu ρ_i. A wtedy, wszystkie następujące biegi też są akceptującymi biegami automatu $\mathcal{A}$:



W dostatecznie długim biegu akceptującym ρ automatu pewien stan q musi się powtórzyć. Wtedy bieg ρ można zdekomponować jako $\rho_1 \cdot \rho_2 \cdot \rho_3$, gdzie ρ_2 zaczyna się i kończy w stanie q. A wtedy biegi $\rho_1 \cdot \rho_2{}^k \cdot \rho_3$, dla $k \geqslant 0$, też są biegami akceptującymi.

$$
\begin{array}{rll}
\rho_1 \cdot \rho_3 & \text{po słowie} & w_1 \cdot w_3 \\
\rho_1 \cdot \rho_2 \cdot \rho_3 & \text{po słowie} & w_1 \cdot w_2 \cdot w_3 = w \\
\rho_1 \cdot \rho_2 \cdot \rho_2 \cdot \rho_3 & \text{po słowie} & w_1 \cdot w_2^2 \cdot w_3 \\
\rho_1 \cdot \rho_2 \cdot \rho_2 \cdot \rho_2 \cdot \rho_3 & \text{po słowie} & w_1 \cdot w_2^3 \cdot w_3 \\
 & \cdots\cdots\cdots & \\
\rho_1 \cdot \rho_2^n \cdot \rho_3 & \text{po słowie} & w_1 \cdot w_2^n \cdot w_3, \text{ gdzie } n \geqslant 0
\end{array}
$$

Udowodniliśmy zatem następujący lemat, zwany *lematem o pompowaniu* dla języków regularnych.

Lemat 4. *Przypuśćmy, że język $L \subseteq A^*$ jest regularny. Wówczas istnieje taka stała $N \in \mathbb{N}$, że dla każdego słowa $w \in L$ długości co najmniej N istnieje dekompozycja $w = w_1 \cdot w_2 \cdot w_3$ o następujących własnościach:*

- *słowo w_2 jest niepuste,*
- *$|w_1 \cdot w_2| \leqslant N$,*
- *dla dowolnej liczby naturalnej $k \geqslant 0$ słowo $w_1 \cdot w_2^k \cdot w_3$ należy do języka L.*

Przykład 12. Pokażemy, że język $L = \{a^n b^n \mid n \geqslant 0\}$ nie jest regularny. Przypuśćmy przeciwnie. Wówczas istnieje stała $N \in \mathbb{N}$ o której mowa w lemacie o pompowaniu. Rozważmy słowo $w = a^N b^N$. To słowo należy do języka L więc, na mocy lematu o pompowaniu, istnieje dekompozycja $w = w_1 \cdot w_2 \cdot w_3$ taka, że $w_2 \neq \varepsilon$ oraz $w_1 \cdot w_2^k \cdot w_3 \in L$ dla $k \geqslant 0$, oraz $|w_1 \cdot w_2| \leqslant N$. Wynika stąd, że słowo w_2 zawiera wyłącznie litery a oraz jest niepuste. Słowo $w_1 \cdot w_2^2 \cdot w_3$ zawiera więcej liter a niż liter b, więc nie należy do języka L, co jest sprzecznością.

Otrzymana sprzeczność pokazuje, że język L nie jest regularny. ⌟

2.4.2 *Operacje na językach regularnych*

Widzieliśmy już, że języki regularne są zamknięte na operacje używane w wyrażeniach regularnych: sumę, konkatenację i gwiazdkę Kleene'go. Zobaczymy teraz inne operacje, które zachowują języki regularne.

Przecięcie. Niech K i L będą dwoma językami regularnymi, rozpoznawanymi przez automaty niedeterministyczne $\mathcal{A}$ i $\mathcal{B}$, odpowiednio. Konstruujemy *automat produktowy* $\mathcal{A} \times \mathcal{B}$ który rozpoznaje język $K \cap L$. Idea jest taka, że ten automat jednocześnie i niezależnie symuluje działanie automatów $\mathcal{A}$ i $\mathcal{B}$ na słowie wejściowym. Ma on:

- Zbiór stanów $Q_{\mathcal{A}} \times Q_{\mathcal{B}}$,
- Zbiór stanów początkowych $I_{\mathcal{A}} \times I_{\mathcal{B}}$,
- Zbiór stanów akceptujących $F_{\mathcal{A}} \times F_{\mathcal{B}}$,
- Tranzycje $(p, q) \xrightarrow{a} (r, s)$, gdzie $p \xrightarrow{a} r$ jest tranzycją automatu $\mathcal{A}$ oraz $q \xrightarrow{a} s$ jest tranzycją automatu $\mathcal{B}$.

Dla słowa $w \in A^*$ oraz biegów π i ρ automatów $\mathcal{A}$ i $\mathcal{B}$ po słowie w, możemy skonstruować bieg $\pi \times \rho$ automatu $\mathcal{A} \times \mathcal{B}$ po słowie w, w którym i-ty stan jest parą składającą się z i-tych stanów biegów π oraz ρ. Co więcej, łatwo widać, że każdy bieg automatu $\mathcal{A} \times$

$\mathcal{B}$ po słowie w jest postaci $\pi \times \rho$. Stąd łatwo wynika, że automat $\mathcal{A} \times \mathcal{B}$ rozpoznaje język $K \cap L$. Zauważmy, że jeżeli automaty $\mathcal{A}$ oraz $\mathcal{B}$ są deterministyczne, to wynikowy automat $\mathcal{A} \times \mathcal{B}$ też jest deterministyczny.

Ponadto, gdyby zmienić zbiór stanów akceptujących na $(F_{\mathcal{A}} \times Q_{\mathcal{B}}) \cup (Q_{\mathcal{A}} \times F_{\mathcal{B}})$, to otrzymalibyśmy automat rozpoznający język $K \cup L$. Ta konstrukcja ma tę przewagę nad wcześniejszą konstrukcją (biorącą sumę rozłączną automatów $\mathcal{A}$ i $\mathcal{B}$), że produkuje automat deterministyczny, gdy automaty $\mathcal{A}$ i $\mathcal{B}$ są deterministyczne.

Przykład 13. Pokażemy teraz, że język $L = \{w \in \{a,b\}^* \mid \#_a(w) = \#_b(w)\}$ nie jest regularny. Jedna metoda to powtórzyć argument z przykładu 12. Druga metoda to zredukować do poprzedniego przykładu, korzystając z własności języków regularnych:

Przypuśćmy, że język L jest regularny. Ponieważ języki regularne są zamknięte na przecięcia, to wtedy język $L \cap a^*b^* = \{a^n b^n \mid n \geqslant 0\}$ też by był regularny, a nie jest. ⌟

Dopełnienie. Pokażemy teraz, że języki regularne są zamknięte na dopełnienie: jeżeli język $L \subseteq A^*$ jest regularny, to język $A^* - L$ też jest regularny.

Przypuśćmy, że L jest rozpoznawany przez automat *deterministyczny* $\mathcal{A}$ (ważne też, że $\mathcal{A}$ jest zupełny, tj. dla każdego stanu p i litery a stan $\delta(p,a)$ jest określony). Zamieńmy w automacie $\mathcal{A}$ stany akceptujące na nieakceptujące, i odwrotnie. Widać, że wynikowy automat akceptuje język $A^* - L$.

Ćwiczenie. Pokazać, że to może nie zadziałać, gdyby zacząć od automatu niedeterministycznego, bądź niezupełnego.

Uogólnione wyrażenia regularne. W *uogólnionych wyrażeniach regularnych*, oprócz sumy, konkatenacji i gwiazdki Kleene'ge, dopuszczamy przecięcia $K \cap L$ i dopełnienia L^c (gdzie ustalony jest alfabet A i $L^c = A^* - L$). Na mocy powyższych faktów, wynikowe wyrażenia wciąż definiują jedynie języki regularne (choć być może dużo zwięźlej). Podobnie, można w wyrażeniach regularnych dopuścić operacje $K^+ = K \cdot K^*$ ("przynajmniej jedno wystąpienie"), $K? = \varepsilon + K$ ("co najwyżej jedno wystąpienie"), $K^n = \underbrace{K \cdots K}_{n}$ ("dokładnie n wystąpień"), $K^{\leqslant n} = \bigcup_{i=0}^{n} K^i$ ("co najwyżej n wystąpień"). Te operacje nie zwiększają siły wyrazu wyrażeń regularnych: jeżeli K jest językiem regularnym, to regularne też są jezyki $K^{\leqslant n}, K^n, K^+, K?, K^{\geqslant n} = K^* \cdot K^n$.

Ćwiczenie. Pokazać to.

UNIXowe wyrażenia regularne. Wyrażenia regularne są podstawowym narzędziem w systemie UNIX[11]. Ich składnia jest nieco inna niż ta używana przez nas, co ułatwia ich pisanie na klawiaturze komputera. Poniżej są podane podstawowe elementy składni.

[11] Kenneth Thompson, twórca systemu UNIX, wcześniej rozwijał teorię języków regularnych. Wynalazł np. bardziej wydajny od poprzednich algorytm przekształcania wyrażenia regularnego w automat. Zbudował wokół wyrażeń regularnych wiele narzędzi UNIXowych nich m. in. narzędzie *grep* (*Global search for Regular Expression and Print matching lines*)

`.` *(kropka)*	dowolny pojedynczy znak.
`[agd]`	jeden ze znaków na liście pomiędzy kwadratowymi nawiasami, tj. a, g, d.
`[^agd]`	pojedynczy znak różny od znaków na liście, w tym przypadku, rożny od a, g, d.
`[c-f]`	zakres znaków, tutaj, $\{c, d, e, f\}$.
`E|F`	suma wyrażeń, tj. $E + F$.
`E?`	co najwyżej jedno wystąpienie E.
`E*`	dowolna liczba wystąpień E.
`E+`	co najmniej jedno wystąpienie E.
`E{n}`	dokładnie n wystąpień E.
`E{n,m}`	przynajmniej n i co najwyżej m wystąpień E.

Przykład 14. Wyrażenie `[a-z0-9._%+-]+@[a-z0-9.-]+[a-z]{2,4}` to uproszczone wyrażenie opisujące adresy e-mailowe (pełne wyrażenie jest dużo bardziej skomplikowane, zob. przypis 1 na stronie 11). ⌟

Odwrócenie języka. Dla słowa $w \in A^*$, niech w^R oznacza jego odbicie lustrzane, a dla języka $L \subseteq A^*$, niech $L = \{w^R \mid w \in L\}$. Jeżeli $L \subseteq A^*$ jest językiem regularnym, to język L^R też jest regularny. Można to zobaczyć na dwa sposoby. Jeżeli $\mathcal{E}$ jest wyrażeniem regularnym definiującym język L, to można skonstruować[12] wyrażenie $\mathcal{E}^R$ otrzymane przez "odwrócenie" wyrażenia $\mathcal{E}$, definiujące język L^R.

Drugi sposób, bardziej pouczający, to przez automaty. Jeżeli $\mathcal{A}$ jest niedeterministycznym automatem rozpoznającym język L, to konstruujemy automat $\mathcal{A}^R$ o tych samych stanach co $\mathcal{A}$, ale w którym tranzycje mają odwrócone kierunki, i stany akceptujące są zamienione z początkowymi. Widać, że wynikowy automat $\mathcal{A}^R$ akceptuje dokładnie język L^R. Nawet jeśli automat $\mathcal{A}$ jest deterministyczny, to wynikowy automat $\mathcal{A}^R$ zazwyczaj nie jest.

[12] formalnie, definicja wyrażenia $\mathcal{E}^R$ przebiega przez indukcję po budowie wyrażenia:

$$\begin{aligned} \emptyset^R &= \emptyset, \\ a^R &= a, \\ (\mathcal{E} + \mathcal{F})^R &= \mathcal{E}^R + \mathcal{F}^R, \\ (\mathcal{E} \cdot \mathcal{F})^R &= \mathcal{F}^R \cdot \mathcal{E}^R, \\ (\mathcal{E}^*)^R &= (\mathcal{E}^R)^*. \end{aligned}$$

Podstawienia. Niech A oraz B będą dwoma alfabetami, i niech $h\colon A \to P(B^*)$ będzie funkcją, która literom alfabetu a przypisuje języki nad alfabetem B. Taka funkcja wyznacza funkcję $\hat{h}\colon P(A^*) \to P(B^*)$ przekształcającą języki w języki, zdefiniowaną wzorem:

$$\hat{h}(L) = \bigcup_{a_1 \cdots a_n \in L} h(a_1) \cdots h(a_n),$$

gdzie po prawej jest konkatenacja języków. Przykładowo, jeśli $h(a) = 0^*$ oraz $h(b) = 1^*$ to $\hat{h}((ab)^*) = (0^*1^*)^*$.

Twierdzimy, że jeśli język L oraz języki $h(a)$, dla $a \in A^*$, są regularne, to język $\hat{h}(L)$ też jest regularny. Istotnie: łatwo widać, że jeżeli wyrażenie regularne $\mathcal{E}$ definiuje język L, to język $\hat{h}(L)$ jest definiowany przez wyrażenie regularne $\mathcal{F}$, w którym zamiast litery a wstawiamy wyrażenie regularne $\mathcal{E}_a$ definiujące język $h(a)$.

Przykład 15. Pokażemy teraz, że język $L_{\geqslant} = \{a^n b^m \mid n \geqslant m \geqslant 0\}$ nie jest regularny. Można to bezpośrednio pokazać z lematu o pompowaniu, powtarzając argument z przykładu 12, w którym pokazaliśmy nieregularność języka $L = \{a^n b^n \mid n \geqslant 0\}$. Pokażemy inny sposób.

Rozważmy język $L_{\leqslant}$ zdefiniowany analogicznie do $L_{\geqslant}$. Zachodzi równość $L_{\leqslant} = \hat{h}(L_{\geqslant}^{\mathsf{R}})$, gdzie h jest podstawieniem $a \mapsto b$ oraz $b \mapsto a$, oraz równość $L = L_{\geqslant} \cap L_{\leqslant}$. A zatem, język L można otrzymać z języka $L_{\geqslant}$ dokonując operacji zachowujących regularność. Więc gdyby $L_{\geqslant}$ był regularny, to L też by był, a nie jest. ⌟

Przeciwobrazy homomorficzne. Niech $h\colon A^* \to B^*$ będzie homomorfizmem. Jeżeli język $K \subseteq B^*$ jest regularny, to język $h^{-1}(K)$ też jest regularny. Żeby to pokazać, weźmy automat niedeterministyczny $\mathcal{B}$ rozpoznający język $K \subseteq B^*$ z którego konstruujemy automat $\mathcal{A}$ nad alfabetem A o tych samych stanach co $\mathcal{B}$, następująco. Dla każdej litery $a \in A$ oraz pary stanów p, q takiej, że $p \xrightarrow{h(a)} q$ w automacie $\mathcal{B}$, tworzymy tranzycję $p \xrightarrow{a} q$ w automacie $\mathcal{A}$. Stany akceptujące i początkowe zostawiamy bez zmian. Łatwo widać, że dla $w \in A^*$ zachodzi równoważność $h(w) \in L(\mathcal{B}) \iff w \in L(\mathcal{A})$.

Ilorazy prawo- i lewostronne. Dla dwóch języków $K, L \subseteq A^*$, przez $K^{-1} \cdot L$ oznaczamy język $\{w \mid \exists v \in K.\, vw \in L\}$. Tak więc, język $K^{-1} \cdot L$ składa się ze wszystkich słów w otrzymanych następująco: dla każdego słowa $u \in L$ oraz dla każdego jego prefiksu $v \in K$, słowo w jest otrzymane przez usunięcie prefiksu v ze słowa u. Pokażemy, że jeśli K jest dowolnym językiem a L jest językiem regularnym, to język $K^{-1} \cdot L$ też jest regularny. Weźmy automat niedeterministyczny $\mathcal{B}$ rozpoznający język L, o stanach Q i stanach początkowych I. Zdefiniujmy

Na przykład, jeśli $K = L(a^*)$ oraz $L = L(a^*b^*)$, to $K^{-1} \cdot L = L$.

$$I' = \{q \in Q \mid p \xrightarrow{w} q \text{ dla pewnych } p \in I, w \in K\}.$$

Tak więc, I' jest zbiorem stanów osiągalnych z jakiegoś stanu początkowego, po wczytaniu słowa z języka K. Nietrudno pokazać[13], że automat $\mathcal{B}'$ otrzymany z $\mathcal{B}$ przez uznanie stanów I' za początkowe, rozpoznaje język $K^{-1} \cdot L$.

Symetrycznie definiujemy język $K \cdot L^{-1} = \{w \mid \exists v \in L. wv \in K\}$ i pokazujemy, że jeśli K jest regularny, to $K \cdot L^{-1}$ też.

[13] Formalnie, pokazujemy dla każdego $u \in A^*$ oraz $q \in Q$ równoważność następujących warunków:

1. $q_0 \xrightarrow{u} p$ dla pewnego $q_0 \in I'$,
2. $q_0 \xrightarrow{vu} p$ dla pewnego $q_0 \in I$ oraz $v \in K$.

Dowód przebiega przez indukcję po długości słowa u. Przypadek bazowy, $u = \varepsilon$, wynika z definicji I'. W kroku indukcyjnym korzystamy z obserwacji, że $q_0 \xrightarrow{wa} q$ wtedy i tylko wtedy, gdy istnieje taki stan $p \in Q$, że $q_0 \xrightarrow{w} p$ oraz $p \xrightarrow{a} q$. Z równoważności warunków 1 i 2 wynika łatwo, że $L(\mathcal{B}') = K^{-1} \cdot L(\mathcal{B})$.

2.4.3 *Algorytmy*

Automaty niedeterministyczne i deterministyczne można reprezentować w programach jako krotki stanów, relacji, itp. Rozważymy trzy problemy obliczeniowe. Pierwszy z nich to problem *parsowania*: czy dane słowo jest akceptowane przez dany automat?

Problem: NFA-ACCEPTS
Dane: Automat niedeterministyczny $\mathcal{A}$ oraz słowo $w \in A^*$
Rozstrzygnąć: Czy $w \in L(\mathcal{A})$?

Zauważmy, że gdy automat $\mathcal{A}$ jest deterministyczny, to problem jest trywialny[14]: wystarczy symulować działanie automatu $\mathcal{A}$ na słowie w i sprawdzić, czy ostatni stan jest akceptujący. To właśnie robi pseudokod na marginesie.

Dla automatów niedeterministycznych możemy postąpić podobnie, lecz zamiast jednego aktualnego stanu, przechowywać *zbiór* wszystkich stanów[15] które da się osiągnąć ze stanów początkowych, po przeczytaniu danego prefiksu słowa wejściowego. To daje algorytm działający w czasie $\mathcal{O}(n \cdot k + i)$, dla słowa długości n oraz automatu $\mathcal{A}$ o k tranzycjach oraz i stanach początkowych, oraz używającym pamięci $\mathcal{O}(\log n \cdot k + i)$.

[14] **function** DFA-ACCEPTS(DFA $\mathcal{A}$, word w)
    $q \leftarrow \mathcal{A}.q_0$
    for a **in** w **do**
        $q \leftarrow \mathcal{A}.\delta(q, a)$
    return $(q \in \mathcal{A}.F)$

[15] **function** NFA-ACCEPTS(NFA $\mathcal{A}$, word w)
    $R \leftarrow \mathcal{A}.I$
    for a **in** w **do**
        $R \leftarrow \bigcup_{q \in R} \mathcal{A}.\delta(q, a)$
    return $(R \cap \mathcal{A}.F \neq \emptyset)$

Drugi rozważany problem obliczeniowy, to problem *pustości automatu*: czy istnieje jakiekolwiek słowo akceptowane przez dany automat?

Problem: NFA-EMPTINESS
Dane: automat niedeterministyczny $\mathcal{A}$
Rozstrzygnąć: czy $L(\mathcal{A}) \neq \emptyset$?

Zauważmy, że automat $\mathcal{A}$ akceptuje pewne słowo wtedy, i tylko wtedy, gdy[16] w grafie G skierowanym o wierzchołkach Q i krawędziach $\{(p, q) \mid \exists a \in A.(p, a, q) \in \delta\}$ istnieje ścieżka skierowana prowadząca z wierzchołka w zbiorze I do wierzchołka w zbiorze F. A zatem, by stwierdzić, czy $L(\mathcal{A}) \neq \emptyset$ wystarczy wykonać algorytm przeszukiwania grafu skierowanego G, na przykład używając przeszukiwania w głąb (ang. *depth-first search*, DFS).

[16] **function** NFA-EMPTINESS(NFA $\mathcal{A}$)
    $R \leftarrow \mathcal{A}.I$
    while R *changes* **do**
        $R \leftarrow R \cup \bigcup_{q \in R, a \in A} \mathcal{A}.\delta(q, a)$
    return $(R \cap \mathcal{A}.F \neq \emptyset)$

Trzeci rozważany problem obliczeniowy, to problem *inkluzji automatów*: czy język rozpoznawany przez jeden automat jest zawarty w języku rozpoznawanym przez drugi automat?

Problem: NFA-INCLUSION
Dane: automaty niedeterministyczne $\mathcal{A}_1, \mathcal{A}_2$ nad alfabetem A
Rozstrzygnąć: czy $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$?

Ten problem uogólnia problem pustości (wystarczy za $\mathcal{A}_2$ wziąć automat bez stanów), a także problem pełności – czy dany automat

rozpoznaje wszystkie słowa? – gdyż za $\mathcal{A}_1$ można wziąć automat akceptujący wszystkie słowa.

By rozstrzygnąć problem inkluzji, zauważmy, że $L_1 \subseteq L_2$ wtedy, i tylko wtedy, gdy $L_1 \cap (A^* - L_2) = \emptyset$. Automat $\mathcal{B}$ rozpoznający język $L(\mathcal{A}_1) \cap (A^* - L(\mathcal{A}_2))$ możemy obliczyć w dwóch krokach, korzystając z własności języków regularnych omówionych w sekcji *operacje na językach regularnych*. [17] Mając automat $\mathcal{B}$, sprawdzamy jego pustość, używając algorytmu NFA-EMPTINESS.

[17] Wpierw z automatu $\mathcal{A}_2$ obliczamy automat $\mathcal{A}_2'$ rozpoznający język $A^* - L(\mathcal{A}_2)$ (dopełnienie), a potem, mając $\mathcal{A}_1$ oraz $\mathcal{A}_2'$ obliczamy automat $\mathcal{B}$ rozpoznający język $L(\mathcal{A}_1) \cap (A^* - L(\mathcal{A}_2))$ (przecięcie).

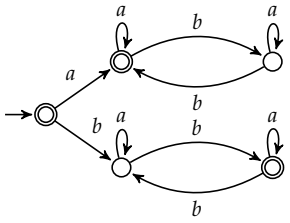
Przykład 16. Praktyczny przykład zastosowania powyższych obserwacji jest następujący: mając dane dwa wyrażenia regularne $\mathcal{E}_1, \mathcal{E}_2$, możemy rozstrzygnąć, czy definiują one ten sam język, tj. czy $L(\mathcal{E}_1) = L(\mathcal{E}_2)$. W tym celu, obliczamy automaty $\mathcal{A}_1, \mathcal{A}_2$ równoważne odpowiednim wyrażeniom, za pomocą algorytmu który jest zaszyty w dowodzie Lematu 2. Dla wyrażeń rozmiaru n, ten algorytm produkuje automat niedeterministyczny o $\mathcal{O}(n)$ stanach w czasie $\mathcal{O}(n^2)$. Następnie, sprawdzamy inkluzje języków $L(A_1), L(A_2)$ w obie strony.

Podobnie, możemy rozstrzygnąć, czy $L(\mathcal{E}_1) \subseteq L(\mathcal{E}_2)$, bądź czy $L(\mathcal{E}_1) \cap L(\mathcal{E}_2)$ jest niepuste. ⌟

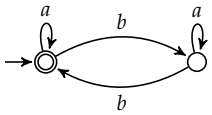
WYKŁAD 4

2.5 *Minimalizacja*

Poniższy automat



rozpoznaje ten sam język, co automat



Każdy język regularny L ma wiele automatów (deterministycznych bądź nie), które go rozpoznają. Jak pokażemy poniżej, jest jednak pewien "kanoniczny" automat *deterministyczny* rozpoznający język L, zwany automatem *syntaktycznym* języka L. Jest on nie tylko najmniejszy pod względem liczby stanów ze wszystkich automatów deterministycznych rozpoznających L, ale też może być otrzymany z dowolnego automatu $\mathcal{A}$ rozpoznającego L poprzez "sklejenie" niektórych stanów.

W reszcie Rozdziału 2.5 będziemy rozważać wyłącznie automaty deterministyczne nad alfabetem A. W takim automacie, dla każdego stanu p oraz słowa w istnieje jedyny taki stan q, że $p \xrightarrow{w} q$. Powyższy stan oznaczamy przez $p \cdot w$. Będziemy rozważali jedynie automaty, w których każdy stan q jest *osiągalny*, tj. taki, że istnieje słowo $w \in A^*$ t.że $q = q_0 \cdot w$, gdzie q_0 to stan początkowy.

Pokażemy, że dla każdego języka regularnego L istnieje automat $\mathcal{A}_L$ (nazywany automatem syntaktycznym języka L), który można otrzymać z dowolnego automatu $\mathcal{A}$ rozpoznającego L poprzez "sklejenie" pewnych stanów ze sobą. Automat $\mathcal{A}_L$ zdefiniujemy abtrakcyjnie, wyłącznie na podstawie języka L. Wpierw zdefiniujemy jego zbiór stanów.

Kongruencja syntaktyczna Niech $L \subseteq A^*$ będzie dowolnym językiem, regularnym bądź nie. Zdefiniujmy relację równoważności $\sim_L$ na słowach w A^* następująco:

$$u \sim_L v \quad wtw. \quad \forall_{w \in A^*}(uw \in L \iff vw \in L).$$

Innymi słowy, $u \not\sim_L v$ wtedy i tylko wtedy, gdy istnieje pewne słowo $w \in A^*$ które "rozróżnia" słowa u i v tak, że dopisanie w do jednego ze słów u, v daje w wyniku słowo należące do L, a dopisanie do drugiego daje słowo nienależące do L.

Przykład 17. Rozważmy język $L = (ab)^*a$. Znajdziemy klasy abstrakcji relacji $\sim_L$. Rozważmy przykładowo słowo ab.

Pokażemy wpierw, że $ab \not\sim aba$. To dlatego, że

$$ab \cdot a \in L \qquad ale \qquad aba \cdot a \notin L.$$

A więc dopisanie słowa a "rozróżnia" słowa ab oraz aba (względem języka L).

Z drugiej strony, pokażemy, że $ab \sim abab$. Rozważmy dowolne takie słowo $u \in \{a, b\}^*$, że $ab \cdot u \in L$. Wtedy u musi być postaci $(ab)^*a$. A zatem, $abab \cdot u$ też należy do języka L. Tak więc,

$$ab \cdot u \in L \implies abab \cdot u \in L.$$

W drugą stronę podobnie:

$$abab \cdot u \in L \implies ab \cdot u \in L.$$

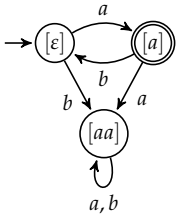
A zatem, $ab \sim abab$. Używając tego samego argumentu, $\varepsilon \sim ab \sim abab \sim ababab \sim \ldots$. Tak więc, słowa postaci $(ab)^*$ są zawarte w jednej klasie abstrakcji relacji $\sim$. Podobnie, $a \sim aba \sim ababa \sim \ldots$, tj. słowa postaci $(ab)^*a$ są zawarte w jednej klasie abstrakcji relacji $\sim$.

Wreszcie, wszystkie słowa, które nie są postaci $(ab)^*$ ani $(ab)^*a$ są parami $\sim$-równoważne (bo dopisanie czegokolwiek do takiego słowa zawsze produkuje słowo nienależące do L), i nie są równoważne ze słowami postaci $(ab)^*$ ani $(ab)^*a$.

Podsumowując, relacja syntaktyczna języka L ma trzy klasy abstrakcji:

$$\begin{aligned}
[\varepsilon]_L &= (ab)^*, \\
[a]_L &= (ab)^*a, \\
[aa]_L &= (ab)^*aa(a+b)^* + (ab)^*b(a+b)^*.
\end{aligned}$$

Na podstawie tych trzech klas abstrakcji możemy łatwo skonstruować automat rozpoznający język L, który jest narysowany na marginesie. ⌟



Automat syntaktyczny Jeżeli relacja $\sim_L$ ma skończenie wiele klas abstrakcji, to możemy skonstruować pewien automat rozpoznający język L, którego stanami są klasy abstrakcji relacji $\sim_L$ (jeżeli $\sim_L$ ma nieskończenie wiele klas abstrakcji to konstrukcja też ma sens, lecz otrzymamy nieskończony automat). Stanem początkowym jest stan $[\varepsilon]_{\sim_L}$, stanami akceptującymi są stany $[w]_{\sim_L}$, a tranzycje są postaci:

$$[w]_{\sim_L} \xrightarrow{a} [wa]_{\sim_L} \qquad \text{dla } w \in A^* \text{ oraz } a \in A.$$

Powyższa definicja definiuje automat, który nazywamy *automatem syntaktycznym* języka L, i oznaczamy $\mathcal{A}_L$.

Lemat 5. *Jeżeli $\sim_L$ ma skończenie wiele klas abstrakcji to automat syntaktyczny jest automatem deterministycznym rozpoznającym język L*

Dowód. Żeby pokazać, że $\mathcal{A}_L$ jest deterministyczny, wystarczy pokazać, że jeśli $u \sim_L v$ to $ua \sim_L va$, dla dowolnych $u, v \in A^*$ oraz $a \in A$. Istotnie, jeśli to pokażemy, to z tego wyniknie, że jeśli $p \xrightarrow{a} q$ oraz $p \xrightarrow{a} q'$ to $q = q'$, czyli automat $\mathcal{A}_L$ jest deterministyczny (jasne jest, że ma jeden stan początkowy, oraz że z każdego stanu wychodzi przynajmniej jedna tranzycja o każdej etykiecie).

Przypuśćmy zatem, że $u_{\sim_L} v$. Rozważmy dowolne słowo $w \in A^*$. Jeżeli $(ua)w \in L$ to również $u(aw) \in L$, więc z równoważności $u_{\sim_L} v$ wynika, że $v(aw) \in L$, co implikuje $(va)w \in L$. Symetrycznie, jeśli $(va)w \in L$ to $(ua)w \in L$. Tak więc, $(ua)w \in L$ wtw. $(va)w \in L$, dla dowolnego $w \in A^*$, dowodząc $ua \sim_L va$.

Pozostaje pokazać, że automat $\mathcal{A}_L$ rozpoznaje język L. Pokażemy, że dla dowolnego słowa $w \in A^*$ zachodzi następujący niezmiennik:

$$[\varepsilon]_{\sim_L} \cdot w = [w]_{\sim_L}$$

(przypomnijmy, że $p \cdot w$ to stan osiągnięty ze stanu p po wczytaniu słowa w). Dowód niezmiennika przebiega przez indukcję po $|w|$. Baza indukcyjna jest natychmiastowa, a w kroku indukcyjnym obserwujemy, że $[w]_{\sim_L} \cdot a = [w \cdot a]_{\sim_L}$, na mocy definicji automatu $\mathcal{A}_L$.

Zaobserwujmy jeszcze, że stan $[w]_{\sim_L}$ jest akceptujący wtedy, i tylko wtedy, gdy $w \in L$. Wynika to z definicji stanów akceptujących automatu $\mathcal{A}_L$ oraz z faktu, że jeśli $u \sim_L v$ to albo obydwa słowa u, v są w języku L, albo obydwa są poza językiem L.

Podsumowując, dla dowolnego słowa $w \in A^*$, autoamt $\mathcal{A}_L$ znajdzie się w stanie $[w]_{\sim_L}$ który jest akceptujący wtedy, i tylko wtedy, gdy $w \in L$. A zatem, automat ten rozpoznaje język L. ∎

Ilorazy automatów Sprecyzujemy teraz definicję "sklejania". Powiemy, że automat $\mathcal{B}$ jest *ilorazem* automatu $\mathcal{A}$ jeśli istnieje taka funkcja

$$f \colon Q_{\mathcal{A}} \xrightarrow{\text{na}} Q_{\mathcal{B}},$$

że:

- jeśli q jest stanem początkowym w $\mathcal{A}$, to $f(q)$ jest stanem początkowym w $\mathcal{B}$,
- jeśli q jest stanem akceptującym w $\mathcal{A}$ to $f(q)$ jest stanem początkowym w $\mathcal{B}$,
- jeśli q nie jest stanem akceptującym w $\mathcal{A}$ to $f(q)$ nie jest stanem początkowym w $\mathcal{B}$,
- jeśli $p \xrightarrow{a} q$ jest tranzycją w $\mathcal{A}$, to $f(p) \xrightarrow{a} f(q)$ jest tranzycją w $\mathcal{B}$.

Możemy myśleć, że stany p i q zostały ze sobą "sklejone" jeśli $f(p) = f(q)$.

Lemat 6. *Jeżeli automat deterministyczny $\mathcal{A}$ rozpoznaje język L to relacja $\sim_L$ ma skończenie wiele klas abstrakcji, oraz automat syntaktyczny $\mathcal{A}_L$ jest ilorazem automatu $\mathcal{A}$.*

Dowód. Niech $\mathcal{A}$ będzie automatem deterministycznym takim, że $L(\mathcal{A}) = L$. Skonstruujemy funkcję $f\colon Q_{\mathcal{A}} \xrightarrow{\text{na}} A^*/\!\sim_L$ następująco (tu $A^*/\!\sim_L$ oznacza zbiór klas abstrakcji relacji $\sim_L$).

Napiszmy $f(q) = [w]_{\sim_L}$ jeżeli $q_0 \cdot w = q$, gdzie q_0 to stan początkowy automatu $\mathcal{A}$. Pokażemy, że to poprawnie definiuje pewną funkcję. Po pierwsze, ponieważ każdy stan w automacie $\mathcal{A}$ jest osiągalny, to dla dowolnego $q \in Q_{\mathcal{A}}$ istnieje pewne takie słowo w, że $q_0 \cdot w = q$. Po drugie, jeśli $q_0 \cdot u = q_0 \cdot v$ to $u \sim_L v$. Istotnie – dla dowolnego słowa $w \in A^*$, jeśli $u \cdot w \in L$ to stan $q_0 \cdot (u \cdot w)$ jest stanem akceptującym w $\mathcal{A}$. Jest to ten sam stan, co $(q_0 \cdot u) \cdot w = (q_0 \cdot v) \cdot w = q_0 \cdot (v \cdot w)$, a wówczas $v \cdot w \in L$. Odwrotnie, jeśli $v \cdot w \in L$, to $u \cdot w \in L$. A więc $u \sim_L v$.

Stąd wynika, że funkcja f jest poprawnie zdefiniowana. Jest ona surjekcją, gdyż dla $w \in A^*$ mamy $[w]_{\sim_L} = f(q_0 \cdot w)$. Skoro mamy surjekcję ze zbioru skończonego $Q_{\mathcal{A}}$ na zbiór $A^*/\!\sim_L$, to ten drugi zbiór też musi być skończony.

Pokażemy teraz, że automat $\mathcal{A}_L$ jest ilorazem automatu $\mathcal{A}$. W tym celu, zauważmy, że funkcja f zdefiniowana powyżej spełnia wszystkie warunki w definicji ilorazu. Przykładowo, jeśli $p \xrightarrow{a} q$ oraz $p = q_0 \cdot w$, to $q_0 \cdot (wa) = q$ oraz $[w]_{\sim_L} \xrightarrow{a} [wa]_{\sim_L}$.

To kończy dowód lematu. ∎

Z powyższego lematu wynikają następujące wnioski.

Twierdzenie 2 (Myhill-Nerode)**.** *Niech $L \subseteq A^*$ będzie dowolnym językiem. Język $L \subseteq A^*$ jest regularny wtedy, i tylko wtedy, gdy jego relacja syntaktyczna $\sim_L$ ma skończenie wiele klas abstrakcji.*

Dowód. Przypuśmy, że język L jest regularny. Na mocy Twierdzenia 1, istnieje automat deterministyczny $\mathcal{A}$ rozpoznający język L. Na mocy Lematu 6, relacja $\sim_L$ ma skończenie wiele klas abstrakcji.

Odwrotnie, jeśli relacja $\sim_L$ ma skończenie wiele klas abstrakcji, to język L jest rozpoznawany przez automat $\mathcal{A}_L$, na mocy Lematu 5. ∎

Poniższy wniosek tłumaczy, dlaczego automat $\mathcal{A}_L$ jest nazywany *automatem minimalnym* języka L: jest to automat o najmniejszej liczbie stanów, spośród wszystkich automatów deterministycznych rozpoznających język L. Ten automat jest jedyny, z dokładnością do izomorfizmu. Dwa automaty $\mathcal{A}$ i $\mathcal{B}$ są izomorficzne, jeżeli różnią się jedynie nazwami stanów. Formalnie: istnieje taka bijekcja $f\colon Q_\mathcal{A} \to Q_\mathcal{B}$, która przekształca stany początkowe $\mathcal{A}$ na stany początkowe $\mathcal{B}$, stany akceptujące $\mathcal{A}$ na stany akceptujące $\mathcal{B}$, oraz taka, że $p \xrightarrow{a} q$ w automacie $\mathcal{A}$ wtedy, i tylko wtedy, gdy $f(p) \xrightarrow{a} f(q)$ w automacie $\mathcal{B}$. Nietrudno zobaczyć, że jeśli $\mathcal{B}$ jest ilorazem $\mathcal{A}$ oraz automaty $\mathcal{B}$ i $\mathcal{A}$ mają tyle samo stanów, to są one izomorficzne.

Wniosek 1. *Jeśli język L jest regularny, to każdy automat go rozpoznający ma przynajmniej tyle stanów, co liczba klas abstrakcji relacji $\sim_L$. Co więcej, każdy automat o dokładnie tej liczbie klas abstrakcji jest izomorficzny z automatem syntaktycznym $\mathcal{A}_L$.*

Dowód. Niech $\mathcal{A}$ rozpoznaje L. Na mocy Lematu 6, automat $\mathcal{A}_L$ jest ilorazem $\mathcal{A}$. A zatem, istnieje surjekcja $f\colon Q_\mathcal{A} \xrightarrow{\text{na}} A^*/\!\sim_L$ na zbiór stanów automatu $\mathcal{A}_L$. Jeżeli $\mathcal{A}$ ma dokładnie $A^*/\!\sim_L$ stanów, to ta surjekcja musi być bijekcją. A zatem, automat $\mathcal{A}_L$ jest wtedy izomorficzny z $\mathcal{A}$. ∎

2.5.1 *Algorytm rafinacji podziałów*

Po angielsku: *partition refinement algorithm.*

Opiszemy teraz algorytm który dostawszy dowolny automat deterministyczny $\mathcal{A}$ rozpoznający język L, oblicza automat syntaktyczny $\mathcal{A}_L$ języka L.

Ustalmy automat $\mathcal{A}$ rozpoznający język L. Zauważmy, że w dowodzie Lematu 6 pokazaliśmy, że automat $\mathcal{A}_L$ jest ilorazem automatu $\mathcal{A}$, oraz skonstruowaliśmy jawnie funkcję $f\colon Q_\mathcal{A} \to A^*/\!\sim_L$ mapującą stany automatu $\mathcal{A}$ w stany automatu $\mathcal{A}_L$. Można myśleć, że automat $\mathcal{A}_L$ otrzymujemy przez "sklejenie" ze sobą pewnych stanów w automacie , gdzie dwa stany p i q są ze sobą sklejone wtedy, i tylko wtedy, gdy $f(p) = f(q)$. Dla $p, q \in Q_\mathcal{A}$, napiszemy $p \sim q$ wtedy, i tylko wtedy, gdy $f(p) = f(q)$. Ponieważ funkcja f jest surjekcją, to zbiór stanów automatu $\mathcal{A}_L$ jest w bijekcji ze zbiorem $Q/\!\sim$ klas abstrakcji relacji $\sim$.

A zatem, w pierwszym kroku, zbadamy relację $\sim$.

Lemat 7. *Następujące warunki są równoważne $p, q \in Q_\mathcal{A}$:*

- $p \sim q$,
- $p \cdot w \in F_{\mathcal{A}} \iff q \cdot w \in F_{\mathcal{A}}$, *dla dowolnego słowa* $w \in A^*$.

Ponadto, istnieje algorytm, który dostawszy automat $\mathcal{A}$*, oblicza relację* $\sim \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{A}}$ *w czasie* $\mathcal{O}(n^2 \cdot s)$*, gdzie* n *to liczba stanow automatu* $\mathcal{A}$ *oraz* s *to rozmiar alfabetu.*

Dowód. Niech $u, v \in A^*$ będą takie, że $q_0 \cdot u = p$ i $q_0 \cdot v = q$. Z definicji funkcji f mamy, że $p \sim q$ wtedy, i tylko wtedy, gdy $u_{\sim_L} v$. Z kolei, z definicji relacji $\sim_L$, jest to równoważne temu, że dla dowolnego słowa $w \in A^*$, zachodzi $u \cdot w \in L \iff v \cdot w \in L$. Zauważmy, że ponieważ automat $\mathcal{A}$ akceptuje język L, to $u \cdot w \in L$ wtedy, i tylko wtedy, gdy $q_0 \cdot (u \cdot w) \in F_{\mathcal{Q}}$. Podobnie, $v \cdot w \in L$ wtedy, i tylko wtedy, gdy $q_0 \cdot (v \cdot w) \in F_{\mathcal{Q}}$. To dowodzi pierwszej częsci lematu lematu.

Żeby opisać algorytm, opiszmy jeszcze w inny sposób relację $\sim$. Mianowicie, rozważmy graf G o wierzchołkach $Q_{\mathcal{A}} \times Q_{\mathcal{A}}$ i krawędziach $(p, q) \xrightarrow{a} (p \cdot a, q \cdot a)$. Wówczas $p \not\sim q$ wtedy, i tylko wtedy, gdy w grafie G istnieje ścieżka z wierzchołka (p, q) do wierzchołka postaci (p', q'), gdzie dokładnie jeden ze stanów p', q' należy do $F_{\mathcal{A}}$. A zatem, zbiór $\not\sim \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{A}}$ możemy obliczyć za pomocą przeszukiwania w szerz w grafie G z odwróconymi krawędziami, rozpoczynając przeszukiwanie od wszystkich wierzchołków postaci (p', q') jak powyżej. Algorytm obliczający relację $\sim$ wpierw konstruuje graf G (w czasie $\mathcal{O}(n^2 \cdot s)$) a później wykonuje przeszukiwanie wgłąb (w czasie liniowym względem rozmiaru grafu). ∎

Mając obliczoną relację $\sim \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{A}}$, obliczamy automat ilorazowy $\mathcal{A}/\sim$ zdefiniowany następująco:

- stanami są klasy abstrakcji relacji $\sim$,
- stanem początkowym jest klasa abstrakcji $[q_0]_\sim$ stanu początkowego $q_0 \in Q_{\mathcal{A}}$,
- stanami akceptującymi są klasy abstrakcji $[q]_\sim$ stanów akceptujących $q \in F_{\mathcal{A}}$,
- tranzycje są postaci $[p]_\sim \xrightarrow{a} [q]_\sim$, gdzie $p \xrightarrow{a} q$ jest tranzycją automatu $\mathcal{A}$.

Nietrudno pokazać, że automat $\mathcal{A}/\sim$ jest izomorficzny z automatem $\mathcal{A}_L$, gdzie stanowi $[q_0 \cdot w]_\sim$ automatu $\mathcal{A}/\sim$ odpowiada stan $[w]_{\sim_L}$ automatu $\mathcal{A}_L$.

Algorytm wpierw oblicza relację $\sim$, korzystając z Lematu 7, a następnie oblicza automat $\mathcal{A}/\sim$, korzystając wprost z jego definicji. Wynikowy automat jest izomorficzny z automatem syntaktycznym.

Podsumowując: algorytm rafinacji podziałów dostaje automat rozpoznający język regularny L o n stanach i s literach w alfabecie,

i oblicza automat syntaktyczny tego języka w czasie $\mathcal{O}(n^2 \cdot s)$. Najszybszy (według pesymistycznego czasu działania) znany algorytm obliczający automat syntaktyczny to algorytm Hopcrofta, który działa w czasie $\mathcal{O}(n \log n \cdot s)$ i jest modyfikacją algorytmu rafinacji.

Pominięte na wykładzie

2.5.2 *Algorytm Brzozowskiego*

Jako ciekawostkę, podamy tu wyjątkowo prosty algorytm minimalizacji automatów, który ma tę zaletę, że działa również gdy wejściowy automat jest niedeterministyczny (oczywiście wyjściowy automat syntaktyczny jest deterministyczny). Wadą zaś jest to, że w pesymistycznym przypadku, czas działania jest wykładniczy względem liczby stanów automatu, nawet dla automatów deterministycznych.

Algorytm korzysta z dwóch operacji: $\mathcal{A} \mapsto P(\mathcal{A})$, która konstruuje automat potęgowy (bez stanów nieosiągalnych), i operacji $\mathcal{A} \mapsto \mathsf{R}(\mathcal{A})$, która zamienia kierunki tranzycji, zamienia stany początkowe na akceptujące i odwrotnie, i wyrzuca stany nieosiągalne. Oto cały algorytm:

return $P(\mathsf{R}(P(\mathsf{R}(\mathcal{A}))))$.

Jego poprawność zostawiamy jako ćwiczenie[18].

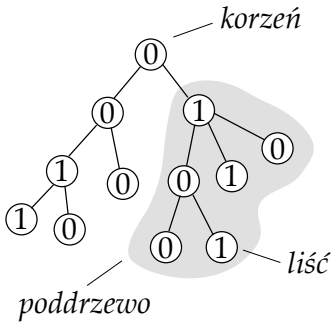
[18] *Wskazówka.* Pokazać, że automat $\mathsf{R}(P(\mathcal{A}))$ jest automatem syntaktycznym dla języka $L(\mathcal{A})^{\mathsf{R}}$, poprzez zbadanie kongruencji wyznaczonej przez ten automat.

2.6 *Regularne języki drzew**

*Pominięte na wykładzie

Do tej pory rozważaliśmy języki słów, ale można też rozważać języki innych obiektów, np. drzew, grafów, itd. Zobaczymy zaraz, że w przypadku drzew, spora część teorii języków regularnych się przenosi.

Rozważamy drzewa *ukorzenione* (tj. z wyróżnionym wierzchołkiem nazywanym korzeniem), *etykietowane* ustalonym alfabetem A (każdy wierzchołek ma etykietę pochodzącą z alfabetu A), *uporządkowane* (tj. dzieci każdego wierzchołka są uporządkowane od lewej do prawej). *Ranga* wierzchołka w drzewie to liczba jego dzieci. Liść to wierzchołek rangi 0.



Automat (niedeterministyczny) na drzewach ma następujące składniki[19]:

- alfabet A, będący skończonym zbiorem,
- zbiór stanów Q,
- zbiór stanów początkowych $I \subseteq Q$,
- relację przejścia[20] $\delta \subseteq Q^* \times A \times Q$ składającą się ze skończonej liczby krotek $(q_1 \dots q_r, a, q)$. Krotkę $(q_1 \dots q_r, a, q) \in \delta$ nazywamy *tranzycją* automatu i zapisujemy tak: $q_1 \cdots q_r \xrightarrow{a} q$ i rysujemy tak:

Biegiem automatu $\mathcal{A}$ po drzewie t jest przyporządkowanie $\rho\colon t \to Q$ wierzchołkom drzewa t stanów z Q (gdzie stan w wierzchołku v rysujemy tuż nad wierzchołkiem v) w sposób zgodny z relacją przejścia δ, tzn. tak, że jeśli wierzchołek v ma etykietę a i jego dzieci mają stany $q_1, \dots q_r$, to v ma stan q taki, że $q_1 \cdots q_r \xrightarrow{a} q$. W szczególności, jeżeli wierzchołek v jest liściem, to dostaje taki stan q, że $\varepsilon \xrightarrow{a} q$. Bieg $\rho\colon t \to Q$ jest *akceptujący* jeżeli w korzeniu jest stan początkowy. Automat $\mathcal{A}$ akceptuje drzewo t jeżeli ma bieg akceptujący po tym drzewie. Język automatu $\mathcal{A}$ to zbiór $L(\mathcal{A})$ wszystkich drzew przez niego akceptowanych.

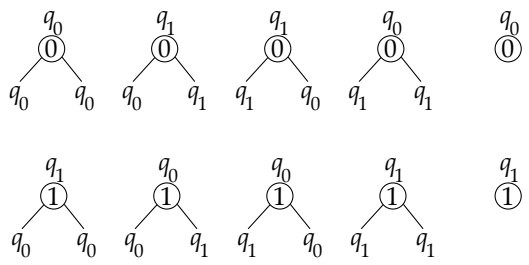
Przykład 18. Opiszemy automat $\mathcal{A}$ którego celem będzie akceptowanie drzew binarnych nad alfabetem $\{0,1\}$, w których każdy wierzchołek ma rangę 0 lub 2, oraz w których liczba wierzchołków o etykiecie 1 jest parzysta. Automat ma dwa stany: q_0 oraz q_1, przy czym stanem początkowym jest stan q_0. Stan q_0 będzie oznaczał, że poniżej jest parzyście wiele wierzchołków z etykietą 1, a stan q_1, że ich jest ich nieparzyście wiele. Tranzycje automatu narysowane są poniżej[21]:

Formalnie, drzewa się definiuje tak samo, jak się je implementuje w programach, tj. indukcyjnie: drzewo to para $t = (a, (t_1, \dots, t_n))$ składająca się z etykiety a oraz krotki $(t_1, \dots, t_n)$ drzew. Wtedy a jest *etykietą korzenia* drzewa t, a $t_1, t_2, \dots, t_n$ to poddrzewa zaczepione pod korzeniem drzewa t. Definiujemy zbiór poddrzew drzewa rekurencyjnie: $\text{Poddrzewa(t)} = \{t\} \cup \bigcup_{i=1}^{n} \text{Poddrzewa}(t_i)$. Wierzchołki drzewa t utożsamiamy z poddrzewami.

[19] wszystkie zbiory są skończone

[20] Według tej definicji, język automatu $L(\mathcal{A})$ składa się wyłącznie z drzew o ograniczonym stopniu. Gdybyśmy chcieli rozważać drzewa o nieograniczonym stopniu, należałoby rozważać relację przejścia która składa się z reguł postaci $\mathcal{E} \xrightarrow{a} q$, gdzie $\mathcal{E}$ jest wyrażeniem regularnym nad alfabetem Q.

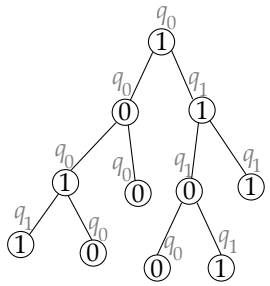
[21] Z postaci tranzycji wynika, że automat jedynie jest w stanie akceptować drzewa o maksymalnej randze 2.



Wzorem:

$$\delta = \{q_i q_j \xrightarrow{k} q_l \mid i,j,k \in \{0,1\}, l = i+j+k \mod 2\} \cup \{\varepsilon \xrightarrow{i} q_i \mid i \in \{0,1\}\}.$$

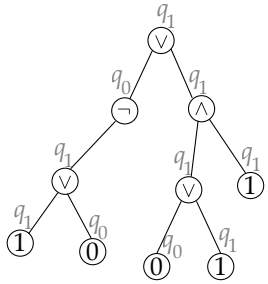
Poniżej narysowany jest bieg akceptujący automatu $\mathcal{A}$ po pewnym drzewie o 11 wierzchołkach.



Nietrudno pokazać[22], że $L(\mathcal{A})$ to zbiór tych drzew binarnych, które mają parzystą liczbę wierzchołków o etykiecie 1. ⌟

[22] Wystarczy pokazać następujący niezmiennik: *jeżeli* $\rho\colon t \to Q$ *jest biegiem automatu* $\mathcal{A}$ *po drzewie* t *oraz* v *jest wierzchołkiem drzewa* t*, to* $\rho(v) = q_0$ *wtedy, i tylko wtedy, gdy* v *ma parzyście wiele potomków (wraz z* v*) o etykiecie* 1.

Przykład 19. Alfabet B składa się z operacji boolowskich $B = \{\vee, \wedge, \neg, 0, 1\}$. Rozważamy język L tych drzew t nad alfabetem B, w których wierzchołki o etykietach $\vee$ lub $\wedge$ mają rangę 2, wierzchołki o etykietach $\neg$ mają rangę 1, a wierzchołki o etykietach $0, 1$ są liśćmi, oraz dodatkowo, drzewo t ewaluowane rekurencyjnie w naturalny sposób ma wartość boolowską 1. Formalnie, najłatwiej jest język L zdefiniować za pomocą automatu $\mathcal{B}$ o stanach q_0, q_1 i tranzycjach:



$$q_i q_j \xrightarrow{\vee} q_{i \vee j}$$
$$q_i q_j \xrightarrow{\wedge} q_{i \wedge j}$$
$$q_i \xrightarrow{\neg} q_{\neg i}$$
$$\varepsilon \xrightarrow{i} q_i,$$

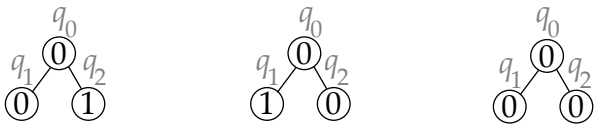
dla $i, j \in \{0,1\}$. Stanem początkowym jest stan q_1. Wówczas $L = L(\mathcal{B})$. ⌟

Automat *deterministyczny liście-korzeń* to taki, w którym relacja δ jest funkcją częściową $\delta\colon Q^* \times A \to Q$, a automat *deterministyczny korzeń-liście* to taki, w którym relacja δ jest[23] funkcją częściową $\delta\colon Q \times A \to Q^*$, oraz jest tylko jeden stan początkowy.

[23] tym razem interpretowana jako podzbiór $Q \times A \times Q^*$

Przykład 20. Automaty $\mathcal{A}$ i $\mathcal{B}$ z Przykładów 18 i 19 są deterministyczne liście-korzeń, ale nie są deterministyczny korzeń-liście: w automacie $\mathcal{A}$, jeżeli w wierzchołku v mamy etykietę 0 oraz stan q_0, to w dzieciach v możemy mieć stany q_0 i q_0 lub stany q_1 i q_1. Przykładowy automat deterministyczny korzeń-liście to automat sprawdzający, że wszystkie wierzchołki w drzewie mają tę samą etykietę [24].

Język L drzew binarnych nad alfabetem $\{0,1\}$ składający się z tych drzew, w których istnieje wierzchołek o etykiecie 1 można rozpoznać automatem deterministycznym liście-korzeń (w stanie zapamiętujemy, czy poniżej jest wierzchołek z etykietą 1), ale nie automatem deterministycznym korzeń-liście. Przypuśćmy, że taki automat istnieje. Wówczas akceptuje on pierwsze dwa drzewa z lewej poniżej. Z determinizmu korzeń-liście wynika[25], że automat ten akceptuje też trzecie drzewo, co jest sprzecznością.



[24] Powiedzmy, że alfabet to $\{a,b\}$ i rozważamy drzewa binarne. Automat ma trzy stany: q_a, q_b oraz q_0, przy czym q_0 jest początkowy. Tranzycje (zapisane odwrotnie, "od góry do dołu") to:
$q_0 \xrightarrow{a} q_a q_a \quad q_a \xrightarrow{a} q_a q_a \quad q_a \xrightarrow{a} \varepsilon$
$q_0 \xrightarrow{b} q_b q_b \quad q_b \xrightarrow{b} q_b q_b \quad q_b \xrightarrow{b} \varepsilon$.

[25] Przypuśćmy, że w automacie jest tranzycja $q_0 \xrightarrow{0} q_1 q_2$ (czytana "od góry do dołu"). Skoro lewe drzewo jest akceptowane, to musi też być tranzycja $q_1 \xrightarrow{0} \varepsilon$. Skoro prawe drzewo jest akceptowane, to musi też być tranzycja $q_2 \xrightarrow{0} \varepsilon$. A wtedy drzewo po prawej ma bieg akceptujący.

⌟

Konstrukcja potęgowa działa dla automatów liście-korzeń, dając następujący wynik:

Twierdzenie 3. *Dla każdego automatu na drzewach $\mathcal{A}$ istnieje taki automat deterministyczny liście-korzeń $\mathcal{B}$, że $L(\mathcal{A}) = L(\mathcal{B})$.*

Język drzew postaci $L(\mathcal{A})$, dla pewnego automatu na drzewach $\mathcal{A}$, nazywamy *regularnym językiem drzew.*

Przykład 21. Ustalmy regularny język słów $R \subseteq A^*$ i liczbę naturalną $d \geqslant 2$. Dla drzewa t nad alfabetem A, niech $\mathrm{Plon}(t)$ oznacza słowo utworzone z etykiet liści, czytanych od lewej do prawej[26]. Niech L będzie językiem tych drzew t nad alfabetem B, w których każdy wierzchołek ma rangę co najwyżej d, oraz $\mathrm{Plon}(t) \in R$. Pokażemy, że L jest regularnym językiem drzew.

Skonstruujemy taki automat niedeterministyczny liście-korzeń $\mathcal{A}$, że $L(\mathcal{A}) = L$. Niech $\mathcal{R}$ będzie automatem niedeterministycznym na słowach rozpoznającym język $R \subseteq A^*$, i niech Q będzie jego zbiorem stanów. Zbiór stanów automatu $\mathcal{A}$ to $Q \times Q$. Definiujemy tranzycje automatu $\mathcal{A}$ następująco:

$$\delta = \{(q_0,q_1)(q_1,q_2)\cdots(q_{i-1},q_i) \xrightarrow{a} (q_0,q_i) \mid i \leqslant d, a \in A\} \cup \{\varepsilon \xrightarrow{a} (p,q) \mid a \in A, p \xrightarrow{a} q \text{ w automacie } \mathcal{R}\}.$$

Łatwo widać, że zachodzi następujący niezmiennik dla drzew t nad alfabetem A, w których każdy wierzchołek ma rangę co najwyżej d:

Szkic dowodu Twierdzenia 3. Jako zbiór stanów automatu $\mathcal{B}$ bierzemy zbiór $P(Q)$, gdzie Q to stany automatu $\mathcal{A}$. Relacja przejścia składa się z tranzycji postaci $Q_1 \cdots Q_n \xrightarrow{a} Q_0$ gdzie $Q_0, \ldots, Q_n \subseteq Q, a \in A$ oraz $\emptyset \neq Q_0 = \{q \in Q \mid q_1 \cdots q_n \xrightarrow{a} q \text{ dla pewnych } q_1 \in Q_1, \ldots, q_n \in Q_n\}$.

[26] Formalnie, jeśli t jest drzewem $(a,(t_1,\ldots,t_n))$ i $n \geqslant 1$, to $\mathrm{Plon}(t) = \mathrm{Plon}(t_1)\cdots\mathrm{Plon}(t_n)$, a jeśli $n = 0$, to $\mathrm{Plon}(t) = a$.

Niezmiennik dowodzimy przez indukcję po rozmiarze drzewa t.

Dla dowolnej pary $(p,q) \in Q \times Q$ istnieje bieg ρ po drzewie t spełniający $\rho(\text{korzeń}) = (p,q)$ wtedy, i tylko wtedy, gdy automat $\mathcal{R}$ ma bieg ze stanu p do stanu q po słowie $\text{Plon}(t)$.

Jako zbiór stanów akceptujących automatu $\mathcal{A}$ bierzemy zbiór $I_{\mathcal{R}} \times F_{\mathcal{R}}$. Z niezmiennika wynika, że $L(\mathcal{A}) = L$. ⌟

Języki drzew mają wiele dobrych własności, z których kilka jest ujętych w poniższym lemacie. Dowód przebiega prawie identycznie, jak w przypadku słów[27].

[27] W przypadku sumy można wziąć sumę rozłączną automatów, w przypadku przecięcia, powtarzamy konstrukcję produktową, a w przypadku różnicy języków, determinizujemy automat korzystając z Twierdzenia 3

Lemat 8. *Regularne języki drzew są zamknięte na przecięcia, sumy, różnice.*

Podobnie jak w przypadku języków słów, regularne języki drzew L można scharakteryzować jako te, dla których pewna relacja syntaktyczna[28] $\sim_L$ ma skończenie wiele klas abstrakcji.

[28] Zdefiniowana tak, że dwa drzewa s, t są równoważne wtedy, i tylko wtedy, gdy w dowolnym drzewie u zastąpienie dowolnego poddrzewa s poddrzewem t nie zmienia przynależności do języka L.

Poza słowami i drzewami, teoria języków regularnych może być z powodzeniem rozwijana dla słów nieskończonych, drzew nieskończonych, niektórych rodzajów grafów przypominających drzewa[29] i różnych innych obiektów.

[29] tzw. grafy o *ograniczonej szerokości drzewiastej*

2.7 *Automaty a logika**

*Pominięte na wykładzie

W tym rozdziale, rozważymy jeszcze jeden formalizm definiujący języki regularne, mianowicie logikę.

Ustalmy alfabet A. Będziemy używać formuł logicznych do opisywania własności słów. Przykładowo, słowo w będzie spełniać formułę

$$\forall x.\forall y.(a(x) \wedge succ(x,y)) \to b(y)$$

jeżeli w słowie w po każdej literze a następuje litera b. Tutaj, zmienne x, y przebiegają po *pozycjach* słowa w, tj. po zbiorze $\{1, 2, \ldots, |w|\}$. Jeżeli x jest pozycją oraz a jest literą alfabetu, to piszemy, że słowo w spełnia $a(x)$, jeżeli x-ta litera słowa w to litera a. Jeżeli x, y są pozycjami, to piszemy $succ(x,y)$ jeżeli pozycja y jest następnikiem pozycji x, tj. $y = x + 1$. Pozycje możemy też porównywać używając relacji $\leqslant, <$ oraz $=$ tj. możemy napisać np. $x \leqslant y$. Wówczas $succ(x,y)$ jest równoważne formule

$$x \leqslant y \wedge \neg(y \leqslant x) \wedge \forall z.(x \leqslant z \wedge z \leqslant y) \to (x = z \vee y = z).$$

Powyższe formuły są formułami *logiki pierwszego rzędu*, co oznacza, że pojawiające się zmienne przebiegają po elementach struktury logicznej. W tym przypadku, struktura ma elementy $\{1, \ldots, |w|\}$ oraz relacje unarne $a(x)$ dla $a \in A$ i relacje binarną $x \leqslant y$ (relacje $<, =$ oraz $succ$ można wyrazić za pomocą $\leqslant$). W formułach logiki pierwszego rzędu, poza wyżej wymienionymi relacjami struktury,

możemy używać zwykłych operacji logicznych, $\wedge, \vee, \neg, \leftarrow, \rightarrow, \leftrightarrow$ oraz kwantyfikatorów $\exists$ oraz $\forall$, choć do wyrażenia tych operacji wystarczy mniejszy zestaw, np. $\exists$ oraz $\wedge$ i $\neg$.

Rozważamy też potężniejszą logikę, zwaną *logiką monadyczną drugiego rzędu*. W tej logice można także kwantyfikować po *zbiorach* pozycji, oznaczanych dużymi zmiennymi X, Y, etc. oraz pisać $x \in X$, jeżeli x jest pozycją i X jest zbiorem pozycji. Przykładowo, formuła

$$\forall X. \forall x. (x \in X) \rightarrow a(x)$$

jest równoważna formule $\forall x. a(x)$.

Zdanie to formuła, w której każda zmienna jest związana kwantyfikatorem. Będziemy małymi literami oznaczać zmienne pierwszego rzędu, tj. takie, które przebiegają po pozycjach słowa – a wielkimi literami – zmienne drugiego rzędu, tj. takie, które przebiegają po zbiorach pozycji słowa. Jeżeli φ jest zdaniem, to piszemy $w \models \varphi$ jeżeli zdanie φ zachodzi w formule w. Przez $L(\varphi)$ oznaczamy zbiór tych słów w, że $w \models \varphi$.

Przykład 22. Jeżeli $\varphi = \forall x. a(x)$, to $L(\varphi)$ to język a^*. Język $(aa)^*$ można opisać formułą logiki monadycznej drugiego rzędu, ale nie formułą logiki pierwszego rzędu. ⌟

Lemat 9. *Niech $\mathcal{A}$ będzie automatem niedeterministycznym nad alfabetem A. Wtedy istnieje taka formuła φ logiki monadycznej drugiego rzędu, że $L(\varphi) = L(\mathcal{A})$.*

Dowód. Bez straty ogólności, możemy założyć, że automat $\mathcal{A}$ ma dokładnie jeden stan początkowy, nazwijmy go q_0. Niech Q będzie zbiorem stanów automatu $\mathcal{A}$.

Bieg automat $\mathcal{A}$ po słowie w to ciąg $q_0, q_1, q_2, \ldots, q_n$, gdzie $q_i \in Q, q_n \in F$ oraz $q_{i-1} \xrightarrow{a}_i q_i$, dla każdego $1 \leqslant i \leqslant |w|$.

Dla każdego stanu $q \in Q$ tworzymy zmienną drugiego rzędu X_q. Idea jest taka, że zmienna X_q to jest zbiór tych pozycji i słowa w, że $q_i = q$ (czyli, intuicyjnie, q_i to stan automatu po wczytaniu i-tej litery słowa). Istnienie biegu akceptującego po słowie w jest równoważne następującej własności:

> Istnieją zbiory $X_q \subseteq \{1, \ldots, |w|\}$, dla $q \in Q$, o następujących własnościach.
>
> - Dla każdej pozycji i istnieje dokładnie jeden taki stan $q \in Q$, że $i \in X_q$.
> - Jeżeli j jest ostatnią pozycją, to $j \in X_q$ dla pewnego $q \in F$.
> - Jeżeli i jest pozycją oraz j jej następnikiem w słowie w, w słowie w na pozycji i jest litera a, $i \in X_p$ i $j \in X_q$, to $p \xrightarrow{a} q$.
> - Jeżeli w słowie w na pierwszej pozycji jest litera a oraz $1 \in X_q$, to $q_0 \xrightarrow{a} q$.

Ćwiczenie. Obie części tego stwierdzenia pozostawiam jako ćwiczenie. Pierwszą część można pokazać łatwo wypisując formułę, która "zgaduje" zbiór pozycji o parzystych numerach, wynika ona też z Lematu 9 poniżej. Druga część jest trudniejsza.

Teraz, powyższą własność zapisujemy jako formułę logiki monadycznej. Formuła zaczyna się od ciągu kwantyfikatorów postaci $\exists X_q$, dla $q \in Q$ (obojętnie w jakiej kolejności). Potem następuje koniunkcja ($\wedge$) czterech formuł, odpowiadającym powyższym punktom. Przykładowo, pierwszy punkt zapisujemy formułą:

$$\forall x. \left(\bigvee_{q \in Q} x \in X_q \right) \wedge \neg \left(\bigwedge_{p,q \in Q, p \neq q} (x \in X_p \wedge x \in X_q) \right).$$

gdzie $\bigvee_{u \in U} \varphi_u$ to skrót notacyjny na $\varphi_{u_1} \vee \varphi_{u_2} \vee \ldots \vee \varphi_{u_s}$, jeżeli $U = \{u_1, \ldots, u_s\}$, i podobnie dla $\bigwedge_{u \in U} \varphi_u$. Podobnie, drugi punkt zapisujemy formułą:

$$\forall x.(\forall y.y \leqslant x) \rightarrow \bigvee_{q \in F} x \in X_q.$$

W podobny sposób formalizujemy drugi i trzeci punkt. ∎

Z Lematu 9 wynika, że logika monadyczna drugiego rzędu jest przynajmniej tak wyrazista, jak automaty niedeterministyczne. Poniższe twierdzenie mówi, że jest też odwrotnie: automaty są przynajmniej tak wyraziste, jak ta logika, a więc, oba formalizmy są równoważne.

Twierdzenie 4. *Niech φ będzie zdaniem logiki monadycznej drugiego rzędu, jak opisane powyżej. Wtedy istnieje taki automat $\mathcal{A}_\varphi$ który akceptuje dokładnie te słowa w, że $w \models \varphi$.*

Zanim udowodnimy twierdzenie, zdefiniujemy powyższe pojęcia trochę precyzyjniej. W szczególności, zdefiniujemy teraz starannie, co to oznacza, że $w \models \varphi$ lub $\varphi \in L(\varphi)$. Poniższy opis można potraktować jako formalną definicję.

Rozróżniamy pomiędzy *zmienną* (czyli symbolem) a jej *wartością*. Wartość zmiennej jest ustalona dopiero, gdy ustalimy pewne *wartościowanie*. Wartościowanie zmiennej drugiego rzędu X w słowie w jest zbiór pozycji w słowie w, tj. podzbiór U zbioru $\{1, \ldots, |w|\}$. Wartościowaniem zmiennej pierwszego rzędu x w słowie w jest pojedynczy element zbioru $\{1, \ldots, |w|\}$.

Zbiory pozycji, lub pojedyncze pozycje w słowie, będziemy reprezentować jako słowa nad alfabetem $\{0, 1\}$. Na przykład zbiór pozycji $\{1, 3, 5\}$ w słowie długości 7 reprezentujemy za pomocą słowa 1010100, również długości 7. Trzecią pozycję w słowie długości 7 reprezentujemy za pomocą słowa 0010000.

Niech $\mathcal{X}$ będzie zbiorem zmiennych. *Wartościowaniem* $\mathcal{X}$ w słowie w jest funkcja, która każdej zmiennej $X \in \mathcal{X}$ drugiego rzędu przyporządkowuje zbiór pozycji $F(X) \subseteq \{1, \ldots, |w|\}$ w słowie w, i każdej zmiennej $x \in \mathcal{X}$ pierwszego rzędu przyporządkowuje pozycję

$F(x) \in \{1, \ldots, |w|\}$ w słowie w. Przykładowo, jeśli $\mathcal{X} = \{X, Y, x\}$ składa się z dwóch zmiennych X, Y drugiego rzędu i jednej zmiennej x pierwszego rzędu, to wartościowaniem zbioru zmiennych $\mathcal{X}$ w słowie *aaabacabab* jest na przykład funkcja F taka, że

$$\begin{aligned} F(x) &= 7, \\ F(X) &= \{3, 5, 7\}, \\ F(Y) &= \{4, 7\}. \end{aligned}$$

Powyższe wartościowanie będziemy reprezentować jako słowo oznaczane $w \otimes F$, następująco. W powyższym przykładzie, słowo $w \otimes F$ jest słowem opisanym poniższą macierzą (bez pierwszej, wyróżnionej kolumny):

x	0	0	0	0	0	0	1	0	0
X	0	0	1	0	1	0	1	0	0
Y	0	0	0	1	0	0	1	0	0
w	a	a	b	a	c	a	b	a	b

Literami tego słowa są całe *kolumny* powyższej macierzy (poza pierwszą). Przykładowo, trzecią literą powyższego słowa jest kolumna $(0, 1, 0, b)$, którą zapiszemy jako parę $(b, [x : 0, X : 1, Y : 0])$ oznaczającą, że na danej pozycji i jest litera b, oraz że $F(x) \neq i, i \in F(X)$ oraz $i \notin F(Y)$.

Formalnie, niech $\{0,1\}^{\mathcal{X}}$ oznacza alfabet, którego elementy to są wektory zerojedynkowe indeksowane elementami zbioru $\mathcal{X}$. Niech $A_{\mathcal{X}}$ oznacza alfabet $A \times \{0,1\}^{\mathcal{X}}$. Literami alfabetu $A_{\mathcal{X}}$ są więc pary (a, f), gdzie $a \in A$ oraz $f \in \{0,1\}^{\mathcal{X}}$. Niech F będzie wartościowaniem zmiennych w $\mathcal{X}$ w słowie w, jak opisano powyżej. Słowo w wraz z wartościowaniem F opisujemy za pomocą słowa nad alfabetem $A_{\mathcal{X}}$ oznaczanego $w \otimes F$, którego i-tą literą jest para (a, f), gdzie a jest i-tą literą słowa w oraz $f \colon \mathcal{X} \to \{0,1\}$ jest taką funkcją, że dla każdej zmiennej pierwszego rzędu $x \in \mathcal{X}$, $f(x) = 1$ wtedy, i tylko wtedy, gdy $x = F(x)$, oraz dla każdej zmiennej drugiego rzędu $X \in \mathcal{X}$, $f(X) = 1$ wtedy, i tylko wtedy, gdy $x \in F(X)$.

Rozważmy teraz formułę φ logiki monadycznej drugiego rzędu, która ma zbiór zmiennych wolnych $\mathcal{X}$, pierwszego lub drugiego rzędu. Zdefiniujmy język $L(\varphi)$ nad alfabetem $A_{\mathcal{X}}$ następująco. Definicja przebiega przez indukcję po budowie formuły φ. Przez *słowo postaci* $w \otimes F$ rozumiemy takie słowa nad alfabetem $A_{\mathcal{X}}$, które poprawnie opisują wartościowania zbioru zmiennych $\mathcal{X}$ w słowie w, tj. zmiennym pierwszego rzędu wartościowanie F przypisuje dokładnie jedną pozycję w słowie.

1. Jeżeli φ jest formułą $x \leqslant y$ (gdzie x i y to zmienne pierwszego rzędu należące do $\mathcal{X}$), to $L(\varphi)$ to zbiór słów postaci $w \otimes F$, gdzie $F(x) \leqslant F(y)$.

2. Jeżeli φ jest formułą $a(x)$, to $L(\varphi)$ to zbiór tych słów postaci $w \otimes F$, że w słowie w na pozycji $F(x)$ jest litera a.

3. Jeżeli φ jest formułą $x \in X$, to $L(\varphi)$ to zbiór słów postaci $w \otimes F$, gdzie $F(x) \in F(X)$.

4. Jeżeli φ jest formułą $\neg\psi$, to $L(\varphi)$ to język słów postaci $w \otimes F$ które nie należą do języka $L(\psi)$.

5. Jeżeli φ jest formułą $\varphi_1 \vee \varphi_2$, to $L(\varphi)$ to język $L(\varphi_1) \cup L(\varphi_2)$.

6. Jeżeli φ jest formułą $\exists x.\psi$, to $L(\varphi)$ to zbiór słów postaci $w \otimes F$, gdzie F jest wartościowaniem zmiennych $\mathcal{X}$, dla pewnego wartościowania $\hat{F}$ zmiennych $\mathcal{X} \cup \{x\}$ rozszerzającego F, zachodzi $w \otimes \hat{F} \in L(\psi)$.

7. Jeżeli φ jest formułą $\exists X.\psi$, to $L(\varphi)$ to zbiór słów $w \otimes F$ nad alfabetem $A_{\mathcal{X}}$ dla których istnieje takie wartościowanie $\hat{F}$ zmiennych $\mathcal{X} \cup \{X\}$, że $w \otimes \hat{F} \in L(\psi)$.

Wreszcie, możemy podać formalną definicję napisu $w \models \varphi$, dla zdania φ:

$$w \models \varphi \iff w \in L(\varphi).$$

Podamy teraz dowód Twierdzenia 4, który jest teraz automatyczny.

Dowód. Pokazujemy ogólniejszą własność:

> *dla każdej formuły φ ze zmiennymi wolnymi $\mathcal{X}$, istnieje taki automat $\mathcal{A}_\varphi$ nad alfabetem $A_{\mathcal{X}}$, że $L(\varphi) = L(\mathcal{A}_\varphi)$.*

Dowód przebiega przez indukcję po rozmiarze formuły φ. W każdym z powyżej opisanych przypadków, konstruujemy automat niedeterministyczny definiujący język $L(\varphi)$. Przykładowo, jeśli φ jest formułą $x \leqslant y$, to nasz automat ma trzy stany: q_0, q_x, q_y, gdzie q_0 jest początkowy i q_y jest akceptujący, i ma następujące tranzycje:

$q_0 \xrightarrow{(a,f)} q_x$ jeżeli litera (a, f) spełnia warunek $f(x) = 1$,
$q_x \xrightarrow{(a,f)} q_y$ jeżeli litera (a, f) spełnia warunek $f(y) = 1$,
$q_0 \xrightarrow{(a,f)} q_y$ jeżeli litera (a, f) spełnia warunek $f(x) = f(y) = 1$.

W przypadku, gdy φ jest formułą $\exists X.\psi$, to automat $\mathcal{A}_\varphi$ konstruujemy z automatu $\mathcal{A}_\psi$, poprzez zastąpienie każdej tranzycji $p \xrightarrow{(a,f)} q$ (gdzie $f\colon \mathcal{X} \cup \{X\} \to \{0,1\}$) tranzycją $p \xrightarrow{(a,f')} q$, gdzie $f' = f|_{\mathcal{X}}$ jest obcięciem f do zbioru $\mathcal{X}$. I tak dalej, dla każdej z powyższych siedmiu reguł. ∎

Zwróćmy uwagę, że w punkcie szóstym i siódmym (kwantyfikacja egzystencjalna), w istotny sposób korzystamy z tego, że możemy rozważać automaty niedeterministyczne. Istotnie, niedeterminizm jest natury egzystencjalnej: automat akceptuje słowo, gdy *istnieje* pewien bieg akceptujący. Z kolei w punkcie czwartym (negacja) wygodnie jest założyć, że mamy automat deterministyczny $\mathcal{A}$ dla języka $L(\psi)$, bo wtedy automat dla języka $L(\neg\psi)$ możemy łatwo obliczyć, zamieniając w automacie $\mathcal{A}$ stany akceptujące na nieakceptujące, i odwrotnie. Tak więc, za każdym razem, gdy po kwantyfikatorze egzystencjalnym następuje negacja, to musimy dokonać determinizacji automatu niedeterministycznego.

Podsumowując, pokazaliśmy, że języki definiowane przez formuły monadycznej logiki drugiego rzędu to dokładnie języki regularne.

Twierdzenie 4 pozwala na stosowanie metod logicznych do badania języków regularnych, i odwrotnie – do stosowania automatów do dowodzenia wyników logicznych.

Tej tematyce poświęcony jest wykład monograficzny *automaty a logika*, który od czasu do czasu się odbywa.

2.8 *Automaty a półgrupy**

*Pominięte na wykładzie

W tym rozdziale, pokażemy jeszcze jeden formalizm definiujący języki regularne. Tutaj, podejście będzie algebraiczne.

Przez *półgrupę* rozumiemy zbiór S wyposażony w operację mnożenia, oznaczaną $\cdot$, która jest łączna:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c).$$

Monoid to półgrupa S która posiada *element neutralny*, tj. taki element $e \in S$, że $e \cdot s = s \cdot e = s$ dla $s \in S$.

Łatwo sprawdzić, że może być tylko jeden element neutralny w półgrupie.

Przykład 23. Przykładowe półgrupy to:

- Każda grupa, np $\mathbb{Z}_3$ z dodawaniem modulo 3.
- Zbiór liczb naturalnych z operacją dodawania.
- Zbiór słów nad alfabetem A z operacją konkatenacji (gdy A ma jeden element, to jest to zasadniczo to samo, co w poprzednim punkcie).
- Zbiór liczb naturalnych z operacją $a, b \mapsto \max(a, b)$.
- Zbiór $\{0, 1\}$ z operacją $\wedge$.
- Jeżeli Q jest zbiorem, to zbiór Q^Q wszystkich funkcji z Q do Q, wraz z operacją składania funkcji jest półgrupą.

Wszystkie te półgrupy są monoidami. Przykładem półgrupy która nie jest monoidem jest półgrupa A^+ składająca się z niepustych słów nad alfabetem A, wraz z konkatenacją. ⌟

Jeżeli S, T są dwoma monoidami, to *homomorfizmem* z S do T jest funkcja $h\colon S \to T$ spełniająca:

$$h(s \cdot t) = h(s) \cdot h(t) \quad \textit{dla } s, t \in S,$$

oraz $h(e_S) = e_T$, gdzie e_S i e_T to elementy neutralne monoidów S i T, odpowiednio.

Przykład 24. Niech S będzie monoidem $\{a, b\}^*$ z operacją konkatenacji i niech T będzie monoidem $\{0, 1\}$ z operacją $\vee$. Definiujemy funkcję $h\colon S \to T$ tak, że $h(w) = 1$ wtedy, i tylko wtedy, gdy słowo w ma literę b. Wtedy h jest homomorfizmem, bo $u \cdot v$ ma literę b wtedy, i tylko wtedy, gdy u ją ma lub gdy v ją ma. Zauważmy, że zbiór $h^{-1}(0) \subseteq A^*$ to język a^*, a zbiór $h^{-1}(1) \subseteq A^*$ to język $(a+b)^*b(a+b)^*$. ⌟

Twierdzenie 5. *Niech $L \subseteq A^*$ będzie językiem. Następujące warunki są równoważne:*

- *L jest językiem regularnym,*
- *Istnieje homomorfizm $h\colon A^* \to S$ w pewien monoid skończony, oraz zbiór $K \subseteq S$, taki, że $L = h^{-1}(K)$.*

Dowód. 1→2. Przypuśćmy, że język L jest regularny. Niech $\mathcal{A}$ będzie takim deterministycznym automatem takim, że $L(\mathcal{A}) = L$. Niech Q będzie jego zbiorem stanów. Niech S będzie monoidem wszystkich funkcji ze zbioru Q w zbiór Q, wraz z operacją lewostronnego składania funkcji

$$f, g \mapsto f; g,$$

gdzie $(f;g)(x) = g(f(x))$. Zdefiniujmy funkcję $h\colon A^* \to S$ wzorem

$$h(w) = f, \quad \textit{gdzie } f\colon Q \to Q \textit{ jest funkcją spełniającą } f(q) = q \cdot w, \textit{ dla } q \in Q.$$

Nietrudno sprawdzić, że powyższa funkcja jest homomorfizmem. Definiujemy K jako zbiór tych funkcji $f\colon Q \to Q$, że $f(q_0) \in F$, gdzie q_0 to stan początkowy automatu $\mathcal{A}$ oraz F to zbiór stanów akceptujących. Z definicji wynika, że

$$h(w) \in K \iff q_0 \cdot w \in F \iff w \in L,$$

kończąc dowód implikacji 1→2.

2→1. Niech $h\colon A^* \to S$ będzie homomorfizmem w monoid skończony, oraz niech $K \subseteq S$ będzie takim zbiorem, ze $L = h^{-1}(K)$. Definiujemy automat $\mathcal{S}$, którego stany to elementy monoidu S, oraz w którym

$$\delta(s, a) = s \cdot h(a), \quad \textit{dla } s \in S, a \in A.$$

Niech q_0 będzie elementem neutralnym monoidu S. Wtedy, przez indukcję po długości słowa $w \in A^*$, pokazujemy, że

$$h(w) = q_0 \cdot w \quad \textit{w automacie } \mathcal{S}.$$

W szczególności, jeżeli zdefiniujemy zbiór stanów akceptujących automatu $\mathcal{S}$ jako K, to dostajemy, że

$$L(\mathcal{S}) = \{w \in A^* \mid q_0 \cdot w \in K\} = \{w \in A^* \mid h(w) \in K\} = h^{-1}(K) = L,$$

kończąc dowód implikacji 2→1. ∎

Twierdzenie 5 pozwala na stosowanie bogatych metod algebraicznych do badania języków regularnych. Przykładowy wynik łączący wiele z poruszonych zagadnień to następujące twierdzenie.

Tej tematyce poświęcony jest wykład monograficzny *automaty a półgrupy*, który od czasu do czasu się odbywa.

Twierdzenie 6 (Schützenberger)**.** *Niech $L \subseteq A^*$ będzie językiem regularnym. Następujące warunki są równoważne:*

1. *Język L jest* bezgwiazdkowy*, tj. da się go zdefiniować wyrażeniem regularnym używającym pojedynczych liter, zbioru pustego, sumy, konkatetenacji, dopełnienia, ale nie używającym gwiazdki Kleene'go.*
2. *Język L da się zdefiniować zdaniem logiki pierwszego rzędu, t.j. istnieje takie zdanie φ nie używające zmiennych drugiego rzędu, że $L = L(\varphi)$.*
3. *Istnieje homomorfizm $h\colon A^* \to S$ w pewien* aperiodyczny *monoid skończony, oraz zbiór $K \subseteq S$, taki, że $L = h^{-1}(K)$. Monoid jest aperiodyczny, jeżeli każda grupa w nim zawarta ma tylko jeden element.*
4. *Istnieje* bezlicznikowy *automat deterministyczny rozpoznający język L. Automat deterministyczny jest bezlicznikowy jeżeli dla każdego słowa $v \in A^*$ i stanu q, jeżeli $q \xrightarrow{v^n} q$ dla pewnego słowa v i liczby $n \geqslant 1$, to $q \xrightarrow{v} q$.*
5. *Automat syntaktyczny języka L jest bezlicznikowy.*

Przykładowo, język $L \subseteq \{a, b\}^*$ słów w spełniających zdanie "po każdym a pojawia się b" jest:

1. bezgwiazdkowy: $L = (\emptyset^c \cdot a)^c$, gdzie X^c oznacza dopełnienie $\{a, b\}^* - X$
2. definiowalny zdaniem pierwszego rzędu: $L = L(\varphi)$, gdzie $\varphi = \forall x.(a(x)) \to \exists y.(x \leqslant y) \wedge b(y)$,
3. równy $h^{-1}(K)$, gdzie $h\colon \{a, b\}^* \to S$ jest homomorfizmem w monoid aperiodyczny $S = \{s, t\}$ w którym $x \cdot y = y$ dla $x, y \in S$, oraz $h(a) = s$, $h(b) = t$ oraz $K = \{b\}$,
4. rozpoznawany przez deterministyczny automat bezlicznikowy, mianowicie automat syntaktyczny języka L.

3
Języki bezkontekstowe

WYKŁAD 5

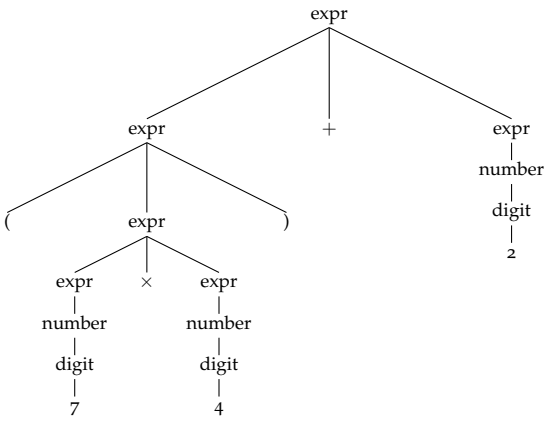
Języki regularne są bardzo przydatne w opisywaniu prostych składni, np. dozwolonych postaci nazw zmiennych w języku programowania czy nazw plików w systemie operacyjnym, czy dozwolonych postaci argumentów programów wywoływanych z linii komend. Jednak do opisu bardziej skomplikowanych składni, takich jak składnia całego języka programowania, czy choćby składnia poprawnie nawiasowanych wyrażeń arytmetycznych, języki regularne nie wystarczają, a do ich opisu stosowane są gramatyki bezkontekstowe. Przykładowo, następująca gramatyka opisuje poprawne wyrażenia arytmetyczne używające symboli $+, \times$ oraz nawiasów:

$$\begin{aligned} \text{expr} &\to (\,\text{expr}\,) \mid \text{expr} + \text{expr} \mid \text{expr} \times \text{expr} \mid \text{number} \\ \text{number} &\to \text{digit number} \mid \text{digit} \\ \text{digit} &\to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

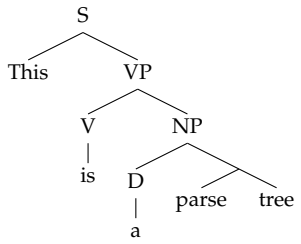
Są dwa spojrzenia na język opisywany przez gramatykę: jako słowa które można *wygenerować* z danego symbolu (np. z symbolu expr) poprzez *derywacje*, bądź słowa, które mają *drzewa parsowania*. Przykładowo, powyższa gramatyka generuje słowo $(7 \times 4) + 2$ za pomocą następującej derywacji:

$$\begin{aligned} \text{expr} \to \text{expr} + \text{expr} \to (\text{expr}) + \text{expr} \to (\text{expr} \times \text{expr}) + \text{expr} \to \\ (\text{number} \times \text{expr}) + \text{expr} \to (\text{number} \times \text{number}) + \text{expr} \to (\text{number} \times \text{number}) + \text{number} \to \\ (\text{digit} \times \text{digit}) + \text{digit} \to (\text{digit} \times 4) + \text{digit} \to (7 \times 4) + \text{digit} \to (7 \times 4) + 2. \end{aligned}$$

Drzewo parsowania jest wygodniejsze do przedstawienia graficznego, ale ma też wiele innych zalet nad derywacjami. Liście drzewa parsowania narysowanego w Ryc. 3.1, czytane od lewej do prawej, tworzą napis $(7 \times 4) + 2$. Poddrzewa o korzeniach z z etykieta *expr*



Rysunek 3.1: Drzewo parsowania wyrażenia $(7 \times 4) + 2$.



odpowiadają podwyrażeniom, mianowicie $7, 4, 2, 7 \times 4$ oraz (7×4). Poddrzewa o korzeniach z z etykieta *number* odpowiadają liczbom pojawiającym się w w wyrażeniu, tj. $7, 4, 2$, itd.

Gramatyki bezkontekstowe oraz drzewa parsowania stosowane są także do opisywania języków naturalnych, np. języka angielskiego.

Języki bezkontekstowe to są te języki, które dadzą się opisać za pomocą gramatyki bezkontekstowej. Nim poświęcony jest ten rozdział.

Generatory parserów. Parser to program który dostawszy słowo w, buduje jego drzewo parsowania zgodne z ustaloną gramatyką $\mathcal{G}$. W Rozdziale 3.4.3 zobaczymy algorytm CYK który jest to w stanie realizować dla dowolnej danej gramatyki $\mathcal{G}$. Tutaj wspomnimy tylko, że istnieją gotowe narzędzia, które są w stanie generować parser na podstawie danej gramatyki $\mathcal{G}$, o której wymaga się, by była w szczególnej postaci. Popularne narzędzia tego typu to Yacc[1], oraz jego warianty, np. Bison. Program Yacc nie tylko generuje parser, ale też kompilator, tj. wynikowy program od razu oblicza wartość wynikowego drzewo, według reguł opisanych w kodzie w języku C (zob. Ryc. 3.2).

[1] Yacc to skrót of od *Yet Another Compiler-Compiler*

```
expr:
    expr '+' expr   { $$ = $1 + $3; }
  | expr '*' expr   { $$ = $1 * $3; }
  | '(' expr ')'    { $$ = $2; }
  | number          { $$ = $1; };

number:
    DIGIT           { $$ = $1; }
  | number DIGIT    { $$ = 10 * $1 + $2; };
```

Rysunek 3.2: Kod gramatyki generującej wyrażenia arytmetyczne dla kompilatora Yacc, wraz z kodem kompilatora wykonującego obliczenie wartości wyrażenia. Przykładowo, w pierwszej regule, wartością wyrażenia postaci $e + f$ jest wartość wyrażenia e plus wartość wyrażenia f, czyli suma wartości w pierwszym i trzecim dziecku danego wierzchołka.

Nieterminal `DIGIT` jest definiowany w innej części kodu, opisującej *lekser* (analizator leksykalny, lub *tokenizer*), który wstępnie etykietuje wejście za pomocą "tokenów", określonych najczęściej za pomocą (UNIX-owych) wyrażeń regularnych. Przykładowo, każda litera pasująca do wyrażenia `[0-9]` dostaje token `DIGIT`.

3.1 Gramatyki bezkontekstowe

Podamy teraz formalną definicję gramatyki bezkontekstowej, drzew parsowania i języka generowanego przez gramatykę.

Gramatyka bezkontekstowa $\mathcal{G}$ ma następujące składniki[2]:

- Zbiór *terminali* A,
- Zbiór *nieterminali* N, rozłączny z A,
- Zbiór *produkcji* postaci $q \to w$, gdzie $q \in N$ oraz $w \in (A \cup N)^*$ jest słowem składającym się z terminali i/lub nieterminali,
- Symbolu *startowego* $S \in N$.

Dla nas wszystkie gramatyki są bezkontekstowe, więc dalej pomijamy ten przymiotnik.

[2] wszystkie zbiory są skończone

Drzewem parsowania takiej gramatyki jest drzewo etykietowane, w którym:

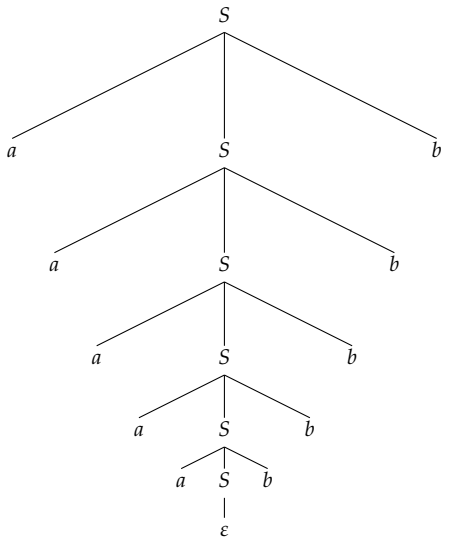
- etykieta korzenia to symbol startowy S,
- liście etykietowane są terminalami ze zbioru $A \cup \{\varepsilon\}$,
- wierzchołki wewnętrzne etykietowane są nieterminalami ze zbioru N,
- jeżeli wierzchołek wewnętrzny ma etykietę q, to albo ma dokładnie jedno dziecko o etykiecie ε, albo ma k dzieci (gdzie $k \geqslant 1$) o etykietach $s_1, \ldots, s_k \in (N \cup A)$ (czytając od lewej do prawej) oraz $q \to s_1 s_2 \ldots s_k$ jest produkcją gramatyki.

Plon drzewa to słowo utworzone z etykiet liści, czytanych od lewej do prawej. Drzewo parsowania słowa w (w gramatyce $\mathcal{G}$) to drzewo parsowania, którego plonem jest słowo w. *Językiem* gramatyki jest zbiór $L(\mathcal{G})$ składający się ze wszystkich słów w dla których istnieje przynajmniej jedno drzewo parsowania. *Język bezkontekstowy* to język postaci $L(\mathcal{G})$, dla pewnej gramatyki $\mathcal{G}$.

[3] $S \to A_1 \mid A_2 \mid \ldots \mid A_n$ to skrót notacyjny dla zbioru produkcji $\{S \to A_1, \ldots, S \to A_n\}$.

Przykład 25. Rozważmy gramatykę $\mathcal{G}$ o terminalach a oraz b i nieterminalu S, oraz produkcjach[3]

$$S \to aSb \mid \varepsilon.$$



Przykładowe drzewo parsowania tej gramatyki jest narysowane z prawej. Przez indukcję po rozmiarze drzewa parsowania widać, że dla każdej liczby $n \geqslant 1$ jest dokładnie jedno drzewo parsowania o wysokości n, i jego plonem jest słowo $a^{n-1}b^{n-1}$. Tak więc, $L(\mathcal{G}) = \{a^n b^n \mid n \geqslant 0\}$. Zauważmy, że nie jest to język regularny. ⌟

Przykład 26 (Palindromy). Rozważmy gramatykę $\mathcal{G}$ o terminalach a oraz b i nieterminalu S, oraz produkcjach

$$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon.$$

Przez indukcję po rozmiarze drzewa parsowania widać, że plonem każdego drzewa parsowania jest palindrom, tj. takie słowo $w \in \{a, b\}$, że $w = w^{\mathsf{R}}$. Tak więc, $L(\mathcal{G}) \subseteq \{w \in \{a, b\}^* \mid w = w^{\mathsf{R}}\}$. Odwrotna inkluzja również zachodzi, co pokazujemy konstruując drzewo parsowania dla każdego palindromu. Konkretnie, przez indukcję po długości n palindromu w, konstruujemy jego drzewo parsowania. Słowa ε, a, b oczywiście mają drzewa parsowania. W kroku indukcyjnym obserwujemy, że każdy palindrom w długości $n \geqslant 2$ ma postać ava bądź bvb, gdzie v jest palindromem długości $n - 2$. Z założenia indukcyjnego, słowo v ma pewne drzewo parsowania t, z którego łatwo produkujemy drzewo parsowania dla słów ava oraz bvb. To dowodzi pozostałej inkluzji. Tak więc, $L(\mathcal{G})$ to dokładnie język palindromów nad alfabetem $\{a, b\}$. ⌟

Przykład 27 (Wyrażenia nawiasowe). Rozważmy gramatykę $\mathcal{G}$ o terminalach (oraz) i nieterminalu S, oraz produkcjach

$$S \to (S) \mid SS \mid \varepsilon.$$

Pokażemy, że $L(\mathcal{G})$ to dokładnie zbiór poprawnych wyrażeń nawiasowych. Czym jest właściwie poprawne wyrażenie nawiasowe? Jest to takie wyrażenie w nad alfabetem składającym się z symboli (oraz), że występujące w nim symbole można "sparować" w niekrzyżujący się sposób. Dokładniej, jeśli słowo w ma długość n, to ma istnieć graf G którego wierzchołkami są pozycje[4] słowa w, mający następujące własności:

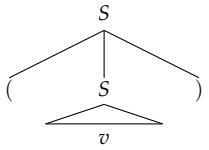
[4] formalnie, są to elementy zbioru $\{1, \ldots, |w|\}$

- graf G jest *skojarzeniem*, tj. każdy jego wierzchołek jest końcem dokładnie jednej krawędzi,
- jeżeli pozycje i i j są połączone krawędzią, gdzie $1 \leqslant i < j \leqslant n$, to $w[i] = $ '(' oraz $w[j] = $ ')'
- krawędzie się "nie krzyżują", tj. dla każdych krawędzi ij oraz kl grafu G, jeżeli $i < k < j$ to $i < l < j$.

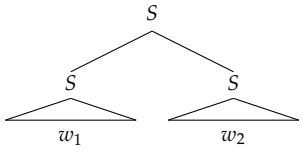
Wpierw pokażemy, że każde drzewo parsowania gramatyki $\mathcal{G}$ jest poprawnym wyrażeniem nawiasowym. W tym celu, dla drzewa parsowania t pokazujemy wprost jak zdefiniować odpowiednie skojarzenie na liściach drzewa t: dwa liście o etykietach (lub) łączymy krawędzią wtedy, i tylko wtedy, gdy mają wspólnego ojca. Z postaci gramatyki wynika, że każdy liść o etykiecie (lub) jest sparowany dokładnie z jednym innym liściem, o przeciwnej etykiecie. Z własności drzew wynika, że nie ma krzyżujących się krawędzi. A zatem, plon drzewa t jest poprawnym wyrażeniem nawiasowym.

Odwrotnie, rozważmy poprawne wyrażenie nawiasowe w. Przez indukcję po długości $|w|$ pokazujemy, że istnieje drzewo parsowania dla w.

Rozważmy skojarzenie G dla słowa w jak opisane powyżej. W tym skojarzeniu, pierwsza pozycja słowa w jest skojarzona z jakąś inną pozycją, powiedzmy i-tą. Jeżeli $i = |w|$, to w jest postaci (v), gdzie v jest mniejszym wyrażeniem nawiasowym (o skojarzeniu otrzymanym z G przez usunięcie dwóch wierzchołków). Na mocy założenia indukcyjnego, istnieje drzewo parsowania t dla wyrażenia v. Z drzewa t łatwo otrzymujemy drzewo parsowania dla słowa $w = (v)$, korzystając z produkcji $S \to (S)$.



Z kolei, jeżeli $i < |w|$, to pokażemy, że słowa $w_1 = w[1..i]$ oraz $w_2 = w[i+1..n]$ są poprawnymi wyrażeniami nawiasowymi. Zauważmy, że w skojarzeniu G nie mogą istnieć krawędzi o jednym końcu w $\{1,\ldots,i\}$ a drugim końcu w $\{i+1,\ldots,n\}$, bo taka krawędź by się krzyżowała z krawędzią $1i$. Tak więc, G indukuje dwa skojarzenia o wymaganych własnościach dla słów w_1 oraz w_2, więc są one poprawnymi wyrażeniami nawiasowymi, krótszymi od w. Na mocy założenia indukcyjnego, mają one drzewa parsowania t_1 oraz t_2, z których łatwo otrzymujemy drzewo parsowania dla słowa $w = w_1w_2$, korzystając z produkcji $S \to SS$. ⌟



Przykład 28 (Tyle samo *a* co *b*). Opiszemy teraz gramatykę $\mathcal{G}$ generującą język $L = \{w \in \{a,b\}^* \mid \#_a(w) = \#_b(w)\}$. Ma ona następujące produkcje:

$$S \to aSb \mid bSa \mid SS \mid \varepsilon.$$

Łatwo pokazać przez indukcję po rozmiarze drzewa parsowania, że jeśli $w \in L(\mathcal{G})$, to $\#_a(w) = \#_b(w)$, co dowodzi inkluzji $L(\mathcal{G}) \subseteq L$.

Odwrotnie, pokazujemy, że jeśli $w \in L$, to $w \in L(\mathcal{G})$, przez indukcję po długości słowa w. Ustalmy słowo w. Dla $i = 1,\ldots,n$, niech $k_i = \#_a(w[1..i]) - \#_b(w[1..i])$. Z definicji L, zachodzi $k_n = 0$. Kluczowa obserwacja jest taka, że ma miejsce jeden z następujących trzech przypadków:

- $k_i = 0$ dla pewnego $1 \leqslant i < n$. Wówczas, słowa $w_1 = w[1..i]$ oraz $w_2 = w[i+1..n]$ należą do L i są krótsze niż w, więc z założenia indukcyjnego, mają drzewa parsowania. Wtedy $w = w_1w_2$ też ma drzewo parsowania, korzystające w korzeniu z reguły $S \to SS$.

- $k_i > 0$ dla każdego $1 \leqslant i < n$. Wówczas, słowo w zaczyna się od litery a oraz kończy się literą b . A zatem, $w = avb$ i widać, że $v \in L$, więc z założenia indukcyjnego, istnieje drzewo parsowania dla v. Wtedy $w = avb$ też ma drzewo parsowania, korzystające w korzeniu z reguły $S \to aSb$.

Gdyby w się kończyło literą a, to na mocy $k_n = 0$ mielibyśmy $k_{n-1} = -1$, co przeczy założeniu $k_i > 0$.

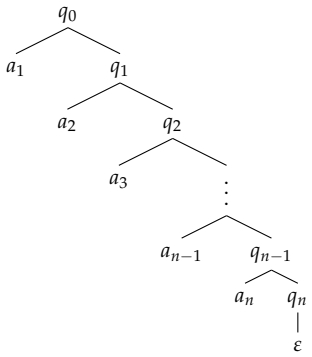
- $k_i < 0$ dla każdego $1 \leqslant i < n$. Ten przypadek jest analogiczny do poprzedniego, i korzysta z reguły $S \to bSa$.

W każdym przypadku, pokazaliśmy, że $w \in L(\mathcal{G})$. Istotne jest, że jeden z tych przypadków musi zachodzić, co wynika ze specyfiki ciągów $k_1, k_2, \ldots, k_n \in \mathbb{Z}$ o tej własności, że $|k_{i+1} - k_i| = 1$, dla każdego $1 \leqslant i < n$. To kończy dowód indukcyjny. ⌟

Nietrudno zobaczyć, że języki bezkontekstowe uogólniają języki regularne:

Lemat 10. *Każdy regularny język słów jest bezkontekstowy.*

Dowód. Niech $\mathcal{A}$ będzie automatem niedeterministycznym o stanach Q, jednym stanie początkowym $q_0 \in Q$. Tworzymy gramatykę o terminalach A, nieterminalach Q, symbolu startowym q_0, oraz produkcjach $p \to aq$, dla każdej tranzycji $p \xrightarrow{a} q$ automatu $\mathcal{A}$, oraz $q \to \varepsilon$ dla każdego stanu akceptującego q. Sprawdzimy teraz, że ta gramatyka rozpoznaje dokładnie język $L(\mathcal{A})$. Zauważmy, że każde drzewo parsowania gramatyki $\mathcal{G}$ ma postać jak pokazano na marginesie, gdzie $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \ldots q_{n-1} \xrightarrow{a_n} q_n$ jest biegiem akceptującym automatu $\mathcal{A}$ po słowie $a_1 \ldots a_n$. Odwrotnie, każdy bieg automatu $\mathcal{A}$ po słowie $a_1 \ldots a_n$ wyznacza drzewo parsowania gramatyki $\mathcal{G}$ dla tego słowa. ■



3.1.1 Lemat o pompowaniu

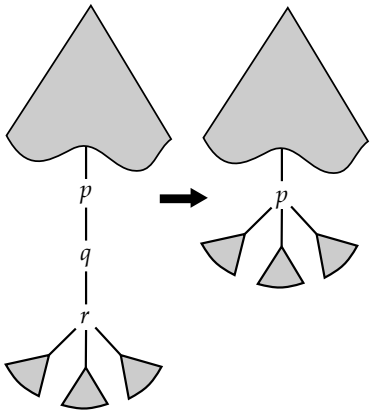
Udowodnimy *lemat o pompowaniu dla języków bezkontekstowych*, który często umożliwia wykazanie, że dany język $L \subseteq A^*$ *nie jest* bezkontekstowy.

Wpierw pokażemy, że każdą gramatykę można przekształcić do pewnej postaci normalnej, o następującej własności $(*)$:

> Dla każdego drzewa parsowania, każdy wierzchołek ma rodzeństwo lub jest liściem o etykiecie różnej od ε.

W szczególności, w takiej gramatyce nie występują nieterminale q takie, że $q \to_{\mathcal{G}} \varepsilon$.

Produkcje postaci $q \to p$, gdzie q, p są nieterminalami lub $p = \varepsilon$, nazywamy *nieefektywnymi*. Zauważmy, że jeśli wszystkie produkcje w gramatyce są efektywne, to spełnia ona warunek $(*)$ opisany powyżej.



Eliminacja nieefektywnych produkcji.

Lemat 11. *Niech $L \subseteq A^*$ będzie językiem bezkontekstowym. Istnieje gramatyka bez nieefektywnych produkcji która generuje język $L - \{\varepsilon\}$.*

Dowód. Ustalmy dowolną gramatykę $\mathcal{G}$ generującą język L. Załóżmy, że $L \neq \emptyset$; w przeciwnym przypadku, teza zachodzi.

Bez straty ogólności, dla każdego nieterminala q, istnieje drzewo parsowania którego korzeń ma etykietę q o niepustym plonie (tj. którego pewien liść ma etykietę różną od ε). Istotnie – jeżeli pewien

nieterminal q nie spełnia tej własności, to możemy się go pozbyć, następująco. Przypuśćmy wpierw, że nie istnieje żadne drzewo parsowania o etykiecie q w korzeniu. Wtedy możemy z gramatyki $\mathcal{G}$ usunąć nieterminal q oraz wszystkie produkcje, w których pojawia się nieterminal q, nie zmieniając języka generowanego przez tę gramatykę. Teraz przypuśćmy, że w każde drzewie parsowania o etykiecie q w korzeniu ma pusty plon. Wtedy nieterminal q również usuwamy, oraz usuwamy go ze wszystkich produkcji, zastępując go ε. W ten sposób otrzymujemy równoważną gramatykę o mniejszej liczbie nieterminali. Powtarzając proces do skutku zapewnimy, że zachodzi opisana własność.

Skonstruujmy teraz gramatykę $\mathcal{H}$, następująco. Składa się ona ze wszystkich efektywnych produkcji postaci $q_0 \to w$, gdzie $q_0 \in N$ oraz $w \in (N \cup A)^*$, dla których w gramatyce $\mathcal{G}$ istnieje ciąg produkcji

$$q_0 \to q_1 \to \cdots \to q_n \to w,$$

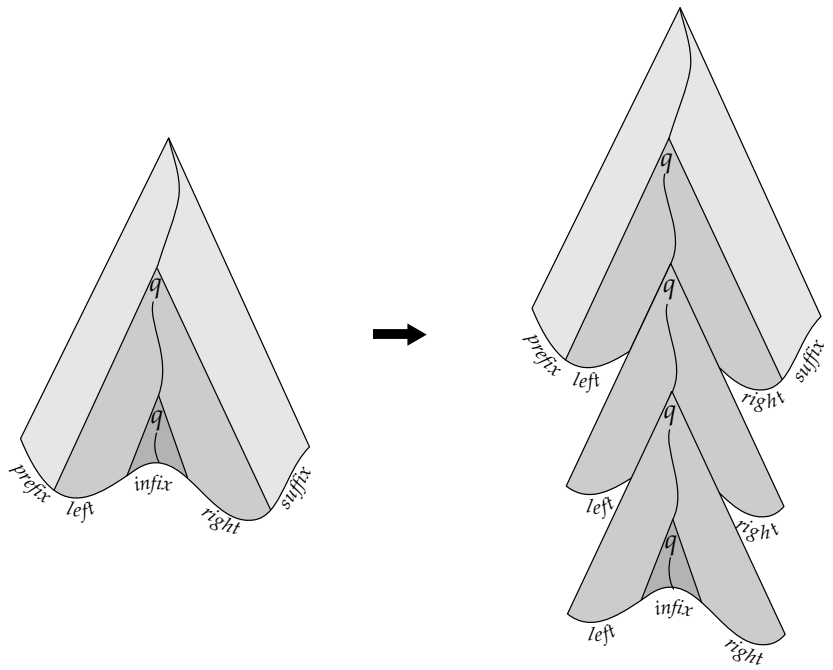
gdzie $q_1, \ldots, q_n$ są nieterminalami. Z definicji, wszystkie produkcje gramatyki $\mathcal{H}$ są efektywne. Łatwo widać, że gramatyka $\mathcal{H}$ produkuje język $L - \{\varepsilon\}$: drzewa parsowania dla gramatyki $\mathcal{G}$ zamieniamy na drzewa parsowania dla gramatyki $\mathcal{H}$, jak na rysunku obok. Dokonując odwrotnej transformacji, drzewa parsowania dla gramatyki $\mathcal{H}$ zamieniamy na drzewa parsowania dla gramatyki $\mathcal{G}$. ∎

Przed sformułowaniem lematu o pompowaniu, zobaczmy jak wygląda struktura bardzo długich słów w języku bezkontekstowym $L \subseteq A^*$. Niech $\mathcal{G}$ będzie gramatyką generującą język $L - \{\varepsilon\}$, nie mająca nieefektywnych produkcji.

Niech d będzie maksymalną liczbą nieterminali występujących po prawej stronie produkcji w gramatyce $\mathcal{G}$. Zauważmy, że w każdym drzewie parsowania t, każdy wierzchołek ma co najwyżej d dzieci. Przez *głębokość* drzewa rozumiemy liczbę wierzchołków wewnętrznych na najdłuższej ścieżce prowadzącej z korzenia do liścia. Skorzystamy z następującej prostej obserwacji:

> *Niech t będzie drzewem o głębokości co najwyżej K, w którym każdy wierzchołek wewnętrzny ma co najwyżej d dzieci. Wtedy t ma co najwyżej d^K liści.*

Niech K będzie liczbą nieterminali w gramatyce $\mathcal{G}$. Przypuśćmy, że słowo $w \in L$ ma więcej niż d^K liter. Ponieważ $w \in L$, istnieje drzewo parsowania t słowa w. Wtedy drzewo t ma $|w| > d^K$ liści, więc na mocy powyższej obserwacji, istnieje pewna ścieżka π prowadząca od korzenia do liścia w drzewie t, która ma więcej niż K wierzchołków wewnętrznych. Przypomnijmy, że K to liczba nieterminali oraz każdy wierzchołek wewnętrzny na ścieżce π jest etykietowany jakimś nieterminalem. A zatem, z zasady szufladkowej (zob. Ryc. 3.3), na



Rysunek 3.3: Pompowanie – konstrukcja drzewa parsowania t_3 dla słowa $w_3 = prefix \cdot left^3 \cdot infix \cdot right^3 \cdot suffix$

ścieżce π pewien nieterminal q pojawia się przynajmniej w dwóch wierzchołkach. Niech x będzie takim wierzchołkiem na ścieżce π z etykietą q, który jest najbliższy liścia, i niech y będzie drugim z kolei najbliższym liścia wierzchołkiem na ścieżce π z etykietą q.

Słowo w możemy sfaktoryzować jako

$$w = prefix \cdot left \cdot infix \cdot right \cdot suffix,$$

gdzie *infix* jest plonem poddrzewa zaczepionego w x i $left \cdot infix \cdot right$ jest plonem poddrzewa zaczepionego w y. Zauważmy, że przynajmniej jedno ze słów *left*, *right* jest niepuste, bo na mocy efektywności, wierzchołek y ma przynajmniej jedno dziecko y' nie leżące na ścieżce π, i dowolny liść poniżej y' jest literą słowa *left* lub *right*. Ponadto, możemy założyć[5], że słowo $left \cdot infix \cdot right$ ma długość co najwyżej d^{K+1}. Konstruujemy drzewa $t_0, t_1, t_2, \ldots$ jak na rysunku. Łatwo widać, że są to poprawne drzewa parsowania dla gramatyki $\mathcal{G}$. Ponadto, drzewo t_k jest drzewem parsowania słowa

$$w_k = prefix \cdot left^k \cdot infix \cdot right^k \cdot suffix.$$

A zatem, słowa $w_0, w_1, w_2, \ldots$ należą do języka L. Udowodniliśmy zatem lemat o pompowaniu (w dowodzie, bierzemy $N = d^{K+1}$):

Lemat 12 (Lemat o pompowaniu dla języków bezkontekstowych)**.** *Przypuśćmy, że język $L \subseteq A^*$ jest bezkontekstowy. Wtedy istnieje taka stała*

[5] Niech y będzie takim wierzchołkiem w drzewie t, że spełnione są dwa warunki: (1) y ma tę samą etykietę, co pewien jego potomek, oraz (2) poddrzewo t' drzewa t zakorzenione w y ma najmniejszą liczbę liści. Gdyby t' miało więcej niż d^{K+1} liści, to znaleźlibyśmy w nim ścieżkę, na której jakiś nieterminal pojawia się dwukrotnie, poza korzeniem y, przecząc założeniu (2).

$N \in \mathbb{N}$, że każde słowo $w \in L$ długości co najmniej N posiada faktoryzację

$$w = prefix \cdot left \cdot infix \cdot right \cdot suffix \tag{3.1}$$

następujących własnościach:

- *słowo $left \cdot right$ jest niepuste,*
- *słowo $left \cdot infix \cdot right$ ma długość co najwyżej N,*
- *dla każdej liczby $k \geqslant 0$, słowo $w_k = prefix \cdot left^k \cdot infix \cdot right^k \cdot suffix$ należy do języka L.*

Przykład 29. Pokażemy, że język $L = \{a^n b^n c^n \mid n \geqslant 0\} \subseteq \{a, b, c\}^*$ nie jest bezkontekstowy. Przypuśćmy przeciwnie, że jest. Wtedy istnieje stała $N \in \mathbb{N}$ o której mowa w Lemacie 12. Rozważmy słowo $w = a^N b^N c^N$ i jego faktoryzację postaci (3.1). Gdyby *left* zawierało dwie różne litery (tj. a i b, lub a i c, lub b i c), to słowo $left^2$ nie byłoby postaci $a^* b^* c^*$, więc słowo w_2 nie mogłoby należeć do języka L. A zatem, *left* używa tylko jednej litery, i podobnie, *right* używa tylko jednej litery. A zatem, jedna z liter a, b, c, nazwijmy ją x, nie pojawia się ani w słowie *left*, ani w słowie *right*. Ponieważ przynajmniej jedno ze słów *left*, *right* jest niepuste, więc jakaś litera y różna od x pojawia się w *left* lub *right*. Słowo w_2 ma więc więcej wystąpień litery y niż litery x, więc $w_2 \notin L$. To przeczy tezie lematu o pompowaniu. Tak więc, język L nie jest bezkontekstowy.

Zauważmy, że język L jest przecięciem dwóch języków bezkontekstowych: $\{a^n b^n c^m \mid n, m \geqslant 0\}$ oraz $\{a^n b^m c^m \mid n, m \geqslant 0\}$. Tak więc, języki bezkontekstowe nie są zamknięte na przecięcia, a w konsekwencji, nie są też zamknięte na dopełnienie. ⌟

Ćwiczenie. Pokazać, że dopełnienie języka L jest językiem bezkontekstowym.

Lemat 12 czasem się okazuje niewystarczający by wykazać, że dany język nie jest bezkontekstowy. Wówczas możemy próbować skorzystać z następującego uogólnienia, którego dowód jest prawie identyczny jak dowód Lematu 12, i pozostawiamy jako ćwiczenie.

Lemat 13 (Lemat Ogdena)**.** *Przypuśćmy, że język $L \subseteq A^*$ jest bezkontekstowy. Wtedy istnieje taka stała $N \in \mathbb{N}$, że każde słowo $w \in L$ w którym zaznaczono co najmniej N pozycji posiada faktoryzację*

$$w = prefix \cdot left \cdot infix \cdot right \cdot suffix$$

o następujących własnościach:

- *$left \cdot right$ zawiera co najmniej jedną zaznaczoną pozycję,*
- *$left \cdot infix \cdot right$ ma co najwyżej N zaznaczonych pozycji,*
- *dla każdej liczby $k \geqslant 0$, słowo $w_k = prefix \cdot left^k \cdot infix \cdot right^k \cdot suffix$ należy do języka L.*

Przykład 30. Rozważmy język

$$L = \{a^i b^j c^k \mid i \neq j, i \neq k, j \neq k\}.$$

Lemat 12 nie przydaje się tu, by wykazać, że język L nie jest bezkontekstowy[6]. Można to jednak pokazać[7] korzystając z Lematu Ogdena, rozważając słowo $a^{N!}b^{2N!}c^{3N!}$, oraz zaznaczając w nim wszystkie litery a. ⌟

[6] Rozważmy długie słowo $w \in L$. Dobieramy dekompozycję jak w (3.1) tak, by *left* $= \varepsilon$ i *right* było pojedynczą literą, o największej liczbie wystąpień w słowie w. Wtedy słowa $w_2, w_3, \ldots$ znów należą do języka L i nie mamy sprzeczności.

[7] Zobacz NR102

WYKŁAD 6

3.2 *Własności języków bezkontekstowych*

W tym rozdziale, zobaczymy że języki bezkontekstowe mają niektóre z dobrych własności języków regularnych (np. są zamknięte na sumę i gwiazdkę Kleene'go), lecz nie wszystkie (np. nie są zamknięte na dopełnienia). Żeby pokazać taki negatywny wynik, potrzebne jest narzędzie do pokazywania, że język L *nie* jest bezkontekstowy. Takim narzędziem jest odpowiedni lemat o pompowaniu, dla języków bezkontekstowych.

3.2.1 *Operacje na językach bezkontekstowych*

Pokażemy niektóre z wielu dobrych własności języków bezkontekstowych.

Suma, konkatenacja, gwiazdka Kleene'go, odwrócenie. Niech K i L będą językami bezkontekstowymi nad alfabetem A. Wtedy ich suma $K \cup L$ też językiem bezkontekstowym: wystarczy wziąć sumę rozłączną gramatyk $\mathcal{G}, \mathcal{H}$ dla języków K i L oraz dodać nowy symbol startowy S z dwiema produkcjami, $S \to S_{\mathcal{G}}$ i $S \to S_{\mathcal{H}}$. Jasne jest, że ta gramatyka generuje język $K \cup L$. Jeśli zamiast tych dwóch produkcji dodamy produkcję $S \to S_{\mathcal{G}} S_{\mathcal{H}}$, to otrzymamy gramatykę, która generuje język $K \cdot L$. Wreszcie, jeśli zamiast tej produkcji dodamy produkcje $S \to S S_{\mathcal{G}}$ i $S \to \varepsilon$, to otrzymamy gramatykę dla języka K^*.

Jeżeli w gramatyce $\mathcal{G}$ zamienimy każdą produkcję $q \to w$ na $q \to w^{\mathsf{R}}$ to otrzymamy gramatykę dla języka $L(\mathcal{G})^{\mathsf{R}}$.

Przykład 31. Język $L = \{a^i b^j \mid i, j \geqslant 0, i \neq j\}$ jest bezkontekstowy. Wystarczy pokazać, że języki $L_1 = \{a^i b^j \mid i > j\}$ oraz $L_2 = \{a^i b^j \mid i < j\}$ są bezkontekstowe, bo $L = L_1 \cup L_2$. Zauważmy, że $L_1 = a^+ \cdot K$, gdzie $K = \{a^i b^i \mid i \geqslant 0\}$ jest językiem bezkontekstowym z Przykładu 25, oraz b^+ jest oczywiście bezkontekstowy. Tak więc, L_1 jest bezkontekstowy, bo jest konkatenacją dwóch języków bezkontekstowych. Podobnie, język $L_2 = K \cdot b^+$ jest bezkontekstowy. A zatem, $L = L_1 \cup L_2$ jest bezkontekstowy, jako suma języków bezkontekstowych. ⌟

Podstawienia. Przypomnijmy, że jeżeli A i B są alfabetami oraz $h\colon A \to P(B^*)$ jest funkcją przypisującą literom alfabetu A języki nad alfabetem B, to w naturalny sposób definiujemy podstawienie (zob. strona 27 w Rozdziale 2.4.2), które jest funkcją $\hat{h}\colon P(A^*) \to P(B^*)$ przekształcającą języki na języki. Przykładowo, jeśli $L = \{a^n b^n \mid n \geqslant 0\}$ oraz h jest podstawieniem $a \mapsto c^*d, b \mapsto b$, to w wyniku podstawienia otrzymamy język $\hat{h}(L) = \bigcup_{n\geqslant 0}(c^*d)^n b^n$.

Twierdzimy, że jeżeli język L oraz języki $h(a)$, dla $a \in A$, są bezkontekstowe, to język $\hat{h}(L)$ też jest bezkontekstowy.

Istotnie: niech $\mathcal{G}$ będzie gramatyką dla języka L oraz niech $\mathcal{G}_a$ będzie gramatyką dla języka $h(a)$, dla każdej litery $a \in A$. Załóżmy przy tym, że wszystkie te gramatyki mają różne zbiory nieterminali, i niech S będzie symbolem startowym w $\mathcal{G}$ i niech S_a będzie symbolem startowym w $\mathcal{G}_a$. Rozważmy gramatykę $\mathcal{H}$ otrzymaną jako sumę wszystkich gramatyk [8] $\mathcal{G}$ oraz $\mathcal{G}_a$, dla $a \in A$. Dodatkowo, zmodyfikujmy gramatykę $\mathcal{H}$ przez zastąpienie każdego terminala $a \in A$ nieterminalem S_a. Otrzymana gramatyka generuje język $\hat{h}(L)$.

[8] To znaczy, nieterminalami gramatyki $\mathcal{H}$ są nieterminale wszystkich tych gramatyk, oraz produkcje to wszystkie produkcje tych gramatyk

Przykład 32. Alfabet B to $\{c, d, \$\}$. Język $\bigcup_{i\geqslant 0}((cd)^*\$)^i((cd)^*\$)^i \subseteq B^*$ jest bezkontekstowy, bo jest to $\hat{h}(L)$, gdzie $L = \{a^i b^i \mid i \geqslant 0\}$ oraz $h(a) = h(b) = (cd)^*\$$. ⌟

Przecięcia z językami regularnymi. Przecięcie dwóch języków bezkontekstowych $K, L \subseteq A^*$ może nie być językiem bezkontekstowym: język $\{a^n b^n c^n \mid n \geqslant 0\}$ jest przecięciem dwóch języków bezkontekstowych: $\{a^n b^n c^m \mid n, m \geqslant 0\}$ oraz $\{a^n b^m c^m \mid n, m \geqslant 0\}$, ale sam bezkontekstowy nie jest, jak zobaczymy w Rozdziale 3.1.1.

Stąd też wynika, że języki bezkontekstowe nie są zamknięte na dopełnienia, bo $K \cap L = (K^c \cup L^c)^c$, a języki bezkontekstowe są zamknięte na sumy.

Udowodnimy jednak, że języki bezkontekstowe są zamknięte na przecięcia z językami regularnymi:

Twierdzenie 7. *Jeśli język $L \subseteq A^*$ jest bezkontekstowy, a język $R \subseteq A^*$ jest regularny, to język $L \cap R$ jest bezkontekstowy.*

Dowód. Niech $\mathcal{A}$ będzie automatem niedeterministycznym rozpoznającym język R oraz niech $\mathcal{G}$ będzie gramatyką generującą język L. Bez utraty ogólności możemy założyć, że każda produkcja w gramatyce $\mathcal{G}$ ma postać $X \to X_1 \cdots X_k$ lub $X \to a$, gdzie $X, X_1, \ldots, X_k$ to nieterminale oraz $a \in A \cup \{\varepsilon\}$ jest literą lub słowem pustym.

Opiszemy gramatykę $\mathcal{G}'$ generującą język $L \cap R$. Jej nieterminale to trójki (p, X, q), gdzie X jest nieterminalem gramatyki $\mathcal{G}$ oraz p, q są stanami automatu $\mathcal{A}$. Gramatyka $\mathcal{G}'$ z nieterminala (p, X, q) będzie generować dokładnie takie słowa $w \in A^*$, że w można wygenerować z X w gramatyce $\mathcal{G}$ oraz automat $\mathcal{A}$ ma bieg ze stanu p do stanu q po słowie w.

Gramatyka $\mathcal{G}'$ ma produkcje

$$(p, X, q) \to (q_0, X_1, q_1)(q_1, X_2, q_2) \cdots (q_{k-1}, X_k, q_k)$$

dla każdej produkcji $X \to X_1 \dots X_k$ gramatyki $\mathcal{G}$ oraz stanów $q_0, \dots, q_k$ automatu $\mathcal{A}$ takich, że $q_0 = p$ oraz $q_k = q$. Dodatkowo, są produkcje

$$(p, X, q) \to a$$

dla każdego $a \in A \cup \{\varepsilon\}$ takiego, że $X \to a$ jest produkcją gramatyki $\mathcal{G}$ oraz automat $\mathcal{A}$ ma tranzycję ze stanu p do stanu q po literze a, lub $p = q$ jeśli $a = \varepsilon$.

Ustalmy stany p, q automatu $\mathcal{A}$ oraz nieterminal X gramatyki $\mathcal{G}$. Pokażemy równoważność dwóch warunków:

(1) Słowo w można wygenerować z nieterminala (p, X, q) w gramatyce $\mathcal{G}$, oraz

(2) w można wygenerować z nieterminala X w gramatyce $\mathcal{G}$ oraz automat $\mathcal{A}$ ma bieg ze stanu p do stanu q po słowie w.

Wpierw dowód implikacji z (1) do (2). Dowód przebiega przez indukcję po rozmiarze drzewa wyprowadzenia.

W bazie indukcji rozważamy takie drzewa, że każde dziecko korzenia jest liściem. Z postaci gramatyki $\mathcal{G}'$ wynika, że takie drzewo ma w korzeniu symbol (p, X, q) a w liściu taki symbol $a \in A \cup \{\varepsilon\}$, że $X \to a$ jest produkcją gramatyki $\mathcal{G}$ oraz automat $\mathcal{A}$ ma tranzycję ze stanu p do stanu q po literze a, lub $p = q$ jeśli $a = \varepsilon$. Wówczas $w = a$ oraz warunek (2) zachodzi.

W kroku indukcyjnym rozważamy drzewa wyprowadzenia które nie są powyższej postaci. Z postaci gramatyki $\mathcal{G}'$ wynika, że takie drzewo ma w korzeniu symbol (p, X, q), a drzewa $T_1, \dots, T_k$ zaczepione pod korzeniem (od lewej do prawej) mają korzenie $(q_0, X_1, q_1), \dots, (q_{k-1}, X_k, q_k)$, gdzie $X \to X_1 \dots X_k$ jest produkcją gramatyki $\mathcal{G}$ oraz $q_0, \dots, q_k$ są stanami automatu $\mathcal{A}$ takimi, że $q_0 = p$ oraz $q_k = q$. Niech $w_1, \dots, w_k$ będą plonami drzew $T_1, \dots, T_k$, odpowiednio. Wówczas, $w = w_1 \cdots w_k$. Warunek (2) wówczas wynika z założenia indukcyjnego zastosowanego do drzew $T_1, \dots, T_k$. To dowodzi impliacji z (1) do (2).

Dowód implikacji z (2) do (1) przebiega podobnie, i pozostawiamy go jako ćwiczenie.

Z równoważności powyższych warunków wynika, że słowo w należy do języka $L \cap R$ wtedy i tylko wtedy, gdy w można wygenerować z pewnego nieterminala (p, S, q) gramatyki $\mathcal{G}'$, gdzie S jest symbolem startowym gramatyki $\mathcal{G}$, p jest stanem początkowym, zaś q stanem akceptującym automatu $\mathcal{A}$.

Do gramatyki $\mathcal{G}'$ dodajmy więc nowy symbol startowy S' oraz produkcje $S' \to (p, S, q)$, gdzie S, p oraz q są jak powyżej. Wówczas gramatyka $\mathcal{G}'$ generuje język $L \cap R$. ∎

Inna metoda udowodnienia powyższego wyniku korzysta z automatów ze stosem, które wprowadzimy w Rozdziale 3.3.

Przykład 33. Czasami, żeby pokazać że jakiś język nie jest bezkontekstowy, warto połączyć lemat o pompowaniu z własnościami zamkniętości języków bezkontekstowych. Rozważmy na przykład język

$$L = \{a^i b^j c^k d^l \mid i, j, k, l \geqslant 0, i = 0 \vee j = k = l\}.$$

Pokażemy, że ten język nie jest bezkontekstowy.

Gdyby był, to język $L' = L \cap ab^*c^*d^*$ też byłby bezkontekstowy, jako przecięcie języka bezkontekstowego z regularnym. Z definicji L, mamy

$$L' = \{ab^n c^n d^n \mid n \geqslant 0\}.$$

Dokonajmy teraz podstawienia $a \mapsto \varepsilon, b \mapsto a, c \mapsto b, d \mapsto c$, otrzymując język

$$L'' = \{a^n b^n c^n \mid n \geqslant 0\}.$$

Skoro języki bezkontekstowe są zamknięte na podstawienia, otrzymany język L'' jest bezkontekstowy, co jest sprzeczne z Przykładem 29. Tak więc, język L nie jest bezkontekstowy. ⌟

3.3 *Automaty ze stosem*

Języki bezkontekstowe są rozpoznawane przez pewien rodzaj automatów, nazywanych automatami *ze stosem*. Model ten uogólnia niedeterministyczne automaty z ε-przejściami. W skrócie, w każdym kroku obliczenia, na podstawie aktualnego stanu, automat może wykonać następujące czynności: wczytać literę słowa wejściowego, wykonać operację stosową (push/pop), przejść do nowego stanu. Zaczniemy od prostego przykładu.

Przykład 34. Zdefiniujemy automat ze stosem rozpoznający język $\{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$. Automat ma dwa stany: q_0 (początkowy) i q_f (akceptujący). Na stosie automat będzie przechowywał trzy rodzaje symboli: #, oznaczający dno stosu, oraz dwa rodzaje "żetonów", $+1$ oraz -1. Niezmiennik będzie taki, że po wczytaniu słowa v, na stosie znajduje się ciąg postaci $\# + 1 + 1 \ldots + 1$ lub $\# - 1 - 1 \ldots - 1$, przy czym suma liczb na stosie jest równa różnicy $\#_a(v) - \#_b(v)$. W szczególności, przeczytawszy słowo w do końca, automat przejdzie do stanu akceptującego, jeśli na stosie będzie tylko #. Taki automat implementujemy za pomocą instrukcji wypisanych obok, i opisanych poniżej.

$$q_0 \xrightarrow{\mathbf{pop}(+1),a,\mathbf{push}(+1+1)} q_0$$
$$q_0 \xrightarrow{\mathbf{pop}(-1),a,\mathbf{push}(\varepsilon)} q_0$$
$$q_0 \xrightarrow{\mathbf{pop}(+1),b,\mathbf{push}(\varepsilon)} q_0$$
$$q_0 \xrightarrow{\mathbf{pop}(-1),b,\mathbf{push}(-1-1)} q_0$$
$$q_0 \xrightarrow{\mathbf{pop}(\#),a,\mathbf{push}(+1)} q_0$$
$$q_0 \xrightarrow{\mathbf{pop}(\#),b,\mathbf{push}(-1)} q_0$$
$$q_0 \xrightarrow{\mathbf{pop}(\#),\varepsilon,\mathbf{push}(\varepsilon)} q_f.$$

Widząc literę a, automat wrzuci na stos żeton $+1$, chyba że na stosie jest żeton -1, wtedy po prostu go zdejmie ze stosu. Widząc literę b, automat zdejmie ze stosu żeton $+1$, chyba, że takiego nie ma, wtedy na stos włożony będzie żeton -1. Formalnie, model zdefiniowany jest tak, że automat zawsze zdejmuje najwyższy żeton ze stosu, więc jeśli chcemy dołożyć żeton $+1$ do stosu na którym już jest $+1$, to zdejmujemy raz $+1$ oraz wkładamy z powrotem dwukrotnie $+1$. Tranzycje automatu są wypisane obok. Ostatnia tranzycja to ε-przejście, tj. nie wczytuje żadnej litery ze słowa wejściowego. Może być tylko wykonana, gdy stos jest pusty (tzn. widać symbol dna stosu #). Poniżej narysowany jest bieg automatu na słowie *aaabbb*. ⌟

(Rysunek.)

Formalna składnia automatu. Automat ze stosem $\mathcal{A}$ ma następujące składniki[9]:

[9] wszystkie zbiory są skończone

- Alfabet wejściowy A,
- Alfabet stosowy Γ zawierający symbol # dna stosu,
- Zbiór stanów Q,
- Zbiór stanów początkowych $I \subseteq Q$,
- Zbiór stanów akceptujących $F \subseteq Q$,
- Zbiór tranzycji δ, gdzie każda tranzycja jest postaci

$$p \xrightarrow{\mathbf{pop}(Z),a,\mathbf{push}(\gamma)} q,$$

dla pewnych $p, q \in Q, a \in A \cup \{\varepsilon\}, Z \in \Gamma, \gamma \in \Gamma^*$.

Formalnie, δ jest skończonym podzbiorem zbioru $Q \times \Gamma \times (A \cup \{\varepsilon\}) \times \Gamma^* \times Q$.

Rysunek 3.4: Konfiguracje automatu ze stosem.

Konfiguracja automatu. W każdym momencie obliczenia, automat jest w konfiguracji (zob. Ryc. 3.4) składającej się z:

- aktualnego stanu $q \in Q$.
- słowa wejściowego w,
- już przeczytanego prefiksu u słowa w,
- pozostałego sufiksu v słowa w, gdzie $u \cdot v = w$,
- aktualnej zawartości stosu, zapisanej jako słowo β nad alfabetem Γ (uznajemy, że stos rośnie w prawo),

Taką konfigurację zapisujemy jako $u[q : \beta]v$ (słowo w można wydedukować z $w = u \cdot v$) i przedstawiamy graficznie jak narysowano obok. Konfiguracja początkowa automatu dla słowa $w \in A^*$ to konfiguracja postaci $\varepsilon[q : \#]w$ dla pewnego stanu początkowego $q \in I$. Konfiguracja akceptująca to konfiguracja postaci $w[q : \gamma]\varepsilon$ dla pewnego $\gamma \in \Gamma^*$ i pewnego stanu akceptującego $q \in F$.

Zdefiniujemy teraz kiedy automat może dokonać przejścia z jednej konfiguracji c do innej konfiguracji d, po tranzycji $\tau \in \delta$. Rozważamy dwa przypadki, w zależności od tego, czy τ jest ε-przejściem, czy nie.

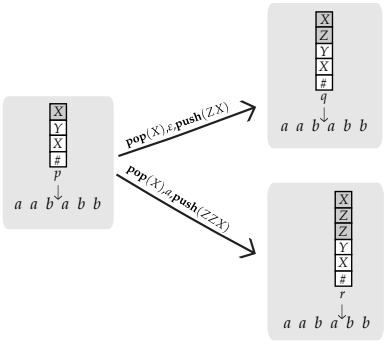
(ε-przejście) Jeżeli $\tau \in \delta$ jest tranzycją postaci $p \xrightarrow{\mathbf{pop}(Z),\varepsilon,\mathbf{push}(\gamma)} q$, to dla każdych $u, v \in A^*, \beta \in \Gamma^*$ piszemy

$$u\,[p : \beta Z]\,v \xrightarrow{\mathbf{pop}(Z),\varepsilon,\mathbf{push}(\gamma)} u\,[q : \beta\gamma]\,v.$$

(wczytanie litery) Jeżeli $\tau \in \delta$ jest tranzycją postaci $p \xrightarrow{\mathbf{pop}(Z),a,\mathbf{push}(\gamma)} q$, to dla każdych $u, v \in A^*, \beta \in \Gamma^*$ piszemy

$$u\,[p : \beta Z]\,av \xrightarrow{\mathbf{pop}(Z),a,\mathbf{push}(\gamma)} ua\,[q : \beta\gamma]\,v.$$

W obydwu przypadkach powyżej, oznaczając przez c konfigurację po lewej stronie strzałki, a przez d konfigurację po prawej strzałki, piszemy też $c \to d$. Tak więc, $c \to d$ oznacza, że automat *może* przejść z konfiguracji c do konfiguracji d.



Pamiętajmy, że rozważane automaty są niedeterministyczne, więc może zachodzić $c \to d$ oraz $c \to d'$ dla dwóch różnych konfiguracji d, d'.

Biegi oraz słowa akceptowane. *Biegiem akceptującym* automatu ze stosem po słowie w jest ciąg konfiguracji

$$c_0 \to c_1 \to \cdots \to c_n,$$

gdzie $n \geqslant 0$, c_0 jest konfiguracją początkową dla słowa w, c_n jest konfiguracją akceptującą, oraz $c_i \to c_{i+1}$ dla $0 \leqslant i \leqslant n$. Automat ze stosem $\mathcal{A}$ *akceptuje* słowo w jeśli ma po nim bieg akceptujący. Język *rozpoznawany* przez automat $\mathcal{A}$ to język $L(\mathcal{A})$ składający się ze wszystkich słów przez niego akceptowanych.

$$
\begin{aligned}
[q_0 : \#]aabbba &\xrightarrow{\mathbf{pop}(\#),a,\mathbf{push}(\#+1)} \\
a[q_0 : \# + 1]abbba &\xrightarrow{\mathbf{pop}(+1),a,\mathbf{push}(+1+1)} \\
aa[q_0 : \# + 1 + 1]bbba &\xrightarrow{\mathbf{pop}(+1),b,\mathbf{push}(\varepsilon)} \\
aab[q_0 : \# + 1]bba &\xrightarrow{\mathbf{pop}(+1),b,\mathbf{push}(\varepsilon)} \\
aabb[q_0 : \#]ba &\xrightarrow{\mathbf{pop}(\#),b,\mathbf{push}(\#-1)} \\
aabbb[q_0 : \# - 1]a &\xrightarrow{\mathbf{pop}(-1),a,\mathbf{push}(\varepsilon)} \\
aabbba[q_0 : \#] &\xrightarrow{\mathbf{pop}(\#),\varepsilon,\mathbf{push}(\varepsilon)} \\
aabbba[q_f : \varepsilon].&
\end{aligned}
$$

Przykład 35. Obok wypisany jest bieg akceptujący automatu $\mathcal{A}$ opisanego w Przykładzie 30 po słowie *aabbba*. Łatwo widać, że dla każdego słowa $w \in \{a, b\}^*$ istnieje bieg akceptujący tego automatu wtedy, i tylko wtedy, gdy $\#_a(w) = \#_b(w)$. Tak więc, $L(\mathcal{A}) = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$. ⌟

Poniższe twierdzenie mówi, że automaty ze stosem rozpoznają dokładnie języki bezkontekstowe.

Twierdzenie 8. *Następujące warunki są równoważne dla języka $L \subseteq A^*$:*

1. *język L jest bezkontekstowy, tj. $L = L(\mathcal{G})$ dla pewnej gramatyki $\mathcal{G}$,*
2. *język L jest rozpoznawany przez pewien automat ze stosem, tj. $L = L(\mathcal{A})$ dla pewnego automatu ze stosem $\mathcal{A}$.*

Zanim omówimy dowód Twierdzenia 8, omówimy różne warianty automatów ze stosem.

Deterministyczne automaty ze stosem. Automat ze stosem jest *deterministyczny* jeżeli każda konfiguracja ma najwyżej jeden następnik, tj. dla każdych konfiguracji c, d, d', jeśli $c \to d$ oraz $c \to d'$, to $d = d'$. Języki definiowane przez deterministyczne automaty ze stosem są nazywane *deterministycznymi językami bezkontekstowymi*.

Deterministyczny automat ze stosem na podstawie aktualnego stanu q oraz symbolu Z na szczycie stosu, podejmuje jednoznacznie decyzję, czy dokonać ε-przejścia, czy wczytać kolejną literę. Gdy zdecyduje się dokonać ε-przejścia, to jednoznacznie na podstawie q i Z decyduje się, jaką operację stosową wykonać, i do jakiego stanu przejść. Gdy decyduje się wczytać kolejną literę, to jednoznacznie na jej podstawie oraz stanu q i symbolu Z decyduje się, jaką operację stosową wykonać, i do jakiego stanu przejść.

Przykład 36. Rozważmy języki

$$
\begin{aligned}
K_1 &= \{a^n b^n \mid n \geqslant 1\}, \\
K_2 &= \{a^n b^{2n} \mid n \geqslant 1\}, \\
K &= K_1 \cup K_2.
\end{aligned}
$$

Dla języka K_1 nietrudno konstruujemy rozpoznający go automat ze stosem $\mathcal{A}_1$. Automat ten, widząc literę a, wrzuca na stos symbol A i pozostaje w stanie q_1, a widząc literę b, przechodzi do stanu q_2 i zdejmuje ze stosu symbol A. Gdy automat jest w stanie q_2 oraz na szczycie stosu jest symbol końca stosu, to automat dokonuje ε-przejścia do stanu akceptującego q_f. Zauważmy, że automat $\mathcal{A}_1$ jest deterministyczny.

Tranzycje automatu $\mathcal{A}_1$:

$$
\begin{aligned}
q_1 &\xrightarrow{\mathbf{pop}(\#),a,\mathbf{push}(\#A)} q_1 \\
q_1 &\xrightarrow{\mathbf{pop}(A),a,\mathbf{push}(AA)} q_1 \\
q_1 &\xrightarrow{\mathbf{pop}(A),b,\mathbf{push}(\varepsilon)} q_2 \\
q_2 &\xrightarrow{\mathbf{pop}(A),b,\mathbf{push}(\varepsilon)} q_2 \\
q_2 &\xrightarrow{\mathbf{pop}(\#),\varepsilon,\mathbf{push}(\#)} q_f.
\end{aligned}
$$

Podobnie możemy skonstruować deterministyczny automat ze stosem $\mathcal{A}_2$ rozpoznający język K_2. Różnica jest taka, że widząc literę a, tym razem automat wrzuca na stos dwa symbole AA (przy literze b nadal zdejmowany jest tylko jeden symbol).

Język K jest też rozpoznawalny przez automat ze stosem, który otrzymujemy biorąc sumę rozłączną automatów $\mathcal{A}_1$ oraz $\mathcal{A}_2$, oraz dodając nowy stan początkowy, z którego jest ε-przejście do stanu początkowego automatu $\mathcal{A}_1$ oraz do stanu początkowego automatu $\mathcal{A}_2$. Otrzymany automat nie jest więc deterministyczny. Co więcej,

można pokazać[10], że nie istnieje deterministyczny automat ze stosem rozpoznający język K.

A zatem, deterministyczne i niedeterministyczne automaty ze stosem nie są równoważne. Ponadto, deterministyczne języki bezkontekstowe nie są zamknięte na sumy, bo, jak widzieliśmy, K_1 i K_2 są deterministyczne a $K_1 \cup K_2$ nie jest. Są za to zamknięte na dopełnienia[11]. To odwrotnie, niż języki bezkontekstowe, które nie są zamknięte na dopełnienia, ale są zamknięte na sumy. ⌟

[10] NR137

Podobnie, język palindromów nie jest deterministyczny, zob. NR138.

[11] Ćwiczenie. Podstawowa idea jest taka, jak przy dopełnieniach dla deterministycznych automatów skończonych: zamieniamy stany akceptujące z nieakceptującymi. Trzeba jednak dodatkowo uważać na nieskończone obliczenia używające ε-przejść.

Akceptacja przez pusty stos. W powyższej definicji, automat akceptuje słowo w, jeśli automat ma bieg który kończy w konfiguracji ze stanem akceptującym (na końcu słowa). Jest też inny tryb akceptacji, zwany *akceptacją przez pusty stos*. Zamiast przechodzić do stanu akceptującego, automat opróżnia cały stos (łącznie z symbolem końca stosu #). Łatwo widać, że te dwa tryby są równoważne: po przejściu do stanu akceptującego q_f, automat może dodatkowo, korzystając z ε-przejść, opróżnić cały stos. Symulacja w drugą stronę (akceptacji przez pusty stos za pomocą akceptacji przez przejście do stanu akceptującego) jest też prosta. Po co w takim razie rozważamy w ogóle akceptację przez pusty stos? To dlatego, że przydaje się rozważać automaty z *jednym stanem*. Jeśli automat ma jeden stan, to nie bardzo jest sens rozważanie akceptacji przez przejście do stanu akceptującego[12]. Ale ma sens rozważanie automatów ze stosem które akceptują przez pusty stos. Okazuje się, że każdy automat ze stosem jest równoważny automatowi ze stosem, który ma tylko *jeden* stan:

[12] Takie automaty rozpoznają jedynie języki zamknięte na branie prefiksów.

Lemat 14. *Dla każdego automatu ze stosem $\mathcal{A}$ istnieje automat ze stosem $\mathcal{A}'$, który ma tylko jeden stan (akceptacja jest przez pusty stos), i jest równoważny automatowi $\mathcal{A}$, tj. $L(\mathcal{A}) = L(\mathcal{A}')$.*

Dowód. Ćwiczenie. ■

Wskazówka: automat $\mathcal{A}'$ na swoim stosie będzie trzymał krotki składające się z symbolu stosowego oraz dwóch stanów automatu $\mathcal{A}$.

Podamy teraz szkic dowodu równoważności gramatyk oraz automatów ze stosem.

Szkic dowodu Twierdzenia 8. Zauważmy wpierw, że na mocy Lematu 14, w sformułowaniu twierdzenia wystarczy rozważać automaty ze stosem z jednym stanem q_0, który będziemy pomijać w notacji, tj. zamiast pisać $q_0 \xrightarrow{\mathbf{pop}(\alpha),a,\mathbf{push}(\gamma_1 \ldots \gamma_k)} q_0$ będziemy pisać po prostu $\mathbf{pop}(\alpha), a, \mathbf{push}(\gamma_1 \ldots \gamma_k)$. Ponadto, gdy $k = 0$, to piszemy po prostu $\mathbf{pop}(\alpha), a$, a gdy $a = \varepsilon$ to piszemy $\mathbf{pop}(\alpha), \mathbf{push}(\gamma_1 \ldots \gamma_k)$. Zauważmy też, że ewentualnie modyfikując nieco automat, możemy założyć, że jego tranzycje są postaci:

(1) $\mathbf{pop}(\alpha); a$ lub

(2) $\mathbf{pop}(\alpha); \mathbf{push}(\gamma_1 \ldots \gamma_k)$.

Podobnie, bez straty ogólności możemy założyć, że rozważane gramatyki $\mathcal{G}$ mają tę własność, że każda produkcja jest jednej z dwóch postaci:

(1′) $\alpha \to a$ lub

(2′) $\alpha \to \gamma_1 \dots \gamma_k$

Jest naturalna odpowiedniość pomiędzy automatami $\mathcal{A}$ powyższej postaci a gramatykami $\mathcal{G}$ powyższej postaci. Mianowicie, mając automat $\mathcal{A}$ powyższej postaci konstruujemy gramatykę $\mathcal{G}$, tworząc, dla każdej tranzycji automatu $\mathcal{A}$ postaci (1) produkcję (1′) gramatyki $\mathcal{G}$, a dla każdej tranzycji postaci (2), tworzymy produkcję (2′) gramatyki $\mathcal{G}$. Ponadto, symbol startowy gramatyki $\mathcal{G}$ to symbol dna stosu automatu $\mathcal{A}$. Odwrotnie dla gramatyki $\mathcal{G}$, możemy stworzyć automat $\mathcal{A}$.

A zatem, gramatyki bezkontekstowe powyższej postaci odpowiadają – przynajmniej syntaktycznie – automatom ze stosem powyższej postaci. By udowodnić twierdzenie, pozostaje wykazać, że jeżeli $\mathcal{A}$ jest automatem ze stosem oraz $\mathcal{G}$ jest odpowiadającą mu gramatyką, to $L(\mathcal{A}) = L(\mathcal{G})$.

Pokażemy teraz inkluzję $L(\mathcal{G}) \subseteq L(\mathcal{A})$. Rozważmy dowolne słowo $w \in L(\mathcal{G})$. A zatem, istnieje drzewo parsowania t dla słowa w, którego plonem jest słowo w. Pokażemy, jak na podstawie drzewa parsowania t skonstruować bieg akceptujący automatu $\mathcal{A}$. W tej konstrukcji, będziemy kolejno odwiedzać wierzchołki drzewa t w porządku preorder, zdefiniowanym poniżej.

Dla (skończonego) drzewa t, definiujemy *porządek preorder* na jego wierzchołkach, następująco: dla dwóch wierzchołków u, v piszemy $u \preceq_{\text{pre}} v$ jeżeli zachodzi jeden z następujących warunków:

- $u = v$,
- u jest potomkiem v, albo
- pewien potomek wierzchołka u jest lewym bratem pewnego potomka wierzchołka v.

(rysunek)

Nietrudno sprawdzić, że relacja $u \preceq_{\text{pre}} v$ jest porządkiem liniowym na wszystkich wierzchołkach drzewa t. Równoważna definicja porządku $\preceq_{\text{pre}}$ jest następująca. Wypisujemy w ciągu wierzchołki drzewa t, zgodnie z następującą procedurą rekurencyjną. Wpierw wypisujemy korzeń drzewa t, a następnie, dla każdego jego dziecka u kolejno – od lewej do prawej – wypisujemy rekurencyjnie wierzchołki poddrzewa ukorzenionego w wierzchołku u. Wówczas, $u \preceq_{\text{pre}} v$ jeżeli u jest wypisane przed v (lub $u = v$).

Niech t będzie drzewem parsowania słowa w oraz niech v będzie dowolnym jego wierzchołkiem. Definiujemy następującą konfigurację c_v automatu $\mathcal{A}$:

- wczytane słowo to słowo utworzone z etykiet wszystkich liści l drzewa t takich, że $l \preceq_{\text{pre}} v$.
- zawartość stosu to słowo utworzone z etykiet wszystkich wierzchołków wewnętrznych u drzewa t (w porządku $\preceq_{\text{pre}}$) takich, że $v \preceq_{\text{pre}} u$ oraz u jest synem pewnego potomka wierzchołka v.

Zauważmy, że jeśli v jest korzeniem drzewa t, to konfiguracja c_v jest konfiguracją początkową, oraz jeśli v jest najbardziej prawym liściem drzewa t, to c_v jest konfiguracją akceptującą automatu $\mathcal{A}$ dla słowa wejściowego w.

Niech $v_0, \ldots, v_k$ będą wszystkimi wierzchołkami wewnętrznymi drzewa t, w porządku preorder. Niech c_i oznacza konfigurację c_{v_i}. Pokażemy, że dla każdego $i = 1, 2, \ldots, k$, automat może dokonać tranzycji z konfiguracji c_{i-1} do konfiguracji c_i. A zatem, ciąg $c_0, c_1, \ldots, c_k$ tworzy bieg akceptujący automatu $\mathcal{A}$ po słowie w, dowodząc, że $w \in L(\mathcal{A})$.

Wystarczy więc pokazać, że jeśli v i v' są dwoma kolejnymi wierzchołkami w porządku preorder, to automat może dokonać tranzycji z konfiguracji $c := c_v$ do konfiguracji $c' := c_{v'}$.

Niech $1 \leqslant i \leqslant k$. Przypuśćmy, że v jest wierzchołkiem o etykiecie α, że ma dzieci o etykietach $\gamma_1, \ldots, \gamma_\ell$ kolejno, oraz że v' jest pierwszym z lewej synem wierzchołka v, który nie jest liściem. Wówczas, automat przechodzi z konfiguracji c do konfiguracji c' wykonując tranzycję $\text{pop}(\alpha); \text{push}(\gamma_1, \ldots, \gamma_\ell)$. Z kolei, jeśli v' jest liściem o etykiecie a, to automat przechodzi z konfiguracji c do konfiguracji c' wykonując tranzycję $\text{pop}(\alpha); a$.

Wreszcie, jeżeli v jest liściem, to konfiguracje c oraz c' są sobie równe.

A zatem, automat $\mathcal{A}$ ma bieg akceptujący po słowie w, stworzony z konfiguracji $c_0, c_1, \ldots, c_k$. W szczególności, $w \in L(\mathcal{A})$. To dowodzi inkluzji $L(\mathcal{G}) \subseteq L(\mathcal{A})$

Inkluzję w drugą stronę pozostawiamy jako ćwiczenie. ■

3.4 *Algorytmy*

WYKŁAD 7

W tym wykładzie rozważamy algorytmiczne problemy związane z językami bezkontekstowymi. Najważniejszym problemem jest problem *parsowania* – czy dane słowo należy do języka generowanego przez daną gramatykę. Wpierw rozważmy prostszy problem, *pustości gramatyki* – czy dana gramatyka generuje jakiekolwiek słowo.

3.4.1 *Pustość gramatyki*

Problem: CFL-EMPTINESS
Dane: gramatyka $\mathcal{G}$
Rozstrzygnąć: czy $L(\mathcal{G}) = \emptyset$?

Powiemy, że nieterminal q gramatyki $\mathcal{G}$ jest *produktywny* jeżeli istnieje drzewo parsowania które w korzeniu ma symbol q. Jasne jest, że $L(\mathcal{G}) \neq \emptyset$ wtedy, i tylko wtedy, gdy symbol startowy gramatyki $\mathcal{G}$ jest produktywny.

Zbiór nieterminali produktywnych gramatyki $\mathcal{G}$ możemy obliczyć prostym algorytmem stałopunktowym: jest to najmniejszy[13] zbiór nieterminali $P \subseteq N$ o tej własności, że jeśli $q \to w$ jest taką produkcją gramatyki, że $w \in (P \cup A)^*$, to $q \in P$.

```
function PRODUCTIVE(Grammar 𝒢)
    P ← ∅
    while P changes do
        for (q → w) production of 𝒢 do
            if w ∈ (P ∪ A)* then
                P ← P ∪ {q}
    return P
```

[13] Dowód poprawności algorytmu. Dla $n \geqslant 1$, niech P_n oznacza zbiór będący wartością zmiennej P przed n-tym wykonaniem ciała pętli **while**, zakładając, że pętlę wykonujemy w nieskończoność. Zachodzi następujący niezmiennik: P_n to zbiór tych nieterminali q, że istnieje drzewo parsowania t o głębokości co najwyżej n i etykiecie q w korzeniu. Dowód przebiega przez łatwą indukcję po $n \geqslant 1$.

Ponieważ ciąg $P_1, P_2, \ldots$ jest rosnącym ciągiem podzbiorów zbioru nieterminali, to istnieje liczba k taka, że $P_1 \subseteq P_2 \subseteq \ldots \subseteq P_k = P_{k+1}$, co dowodzi terminacji algorytmu. Wtedy też $P_k = P_{k+1} = P_{k+2} = \ldots$ bo przy kolejnych wykonaniach pętli zmienna P nie zmienia wartości. Stąd wynika, że $P_k = \bigcup_{n=1}^{\infty} P_n$. Z niezmiennika wynika więc, że P_k jest zbiorem wszystkich nieterminali produktywnych, dowodząc poprawność algorytmu.

Poniżej zademonstrujemy algorytm CYK, który stosuje metodę programowania dynamicznego i działa w czasie sześciennym ze względu na długość słowa w.

3.4.2 *Problem uniwersalności gramatyk*

Rozważmy następujący problem *uniwersalności* gramatyk.

Problem: CFL-UNIVERSALITY
Dane: gramatyka $\mathcal{G}$ nad alfabetem A
Rozstrzygnąć: czy $L(\mathcal{G}) = A^*$?

Analogiczny problem dla automatów niedeterministycznych $\mathcal{A}$ można rozwiązać obliczając wpierw automat $\mathcal{B}$ dla języka $A^* - L(\mathcal{A})$ (korzystając z zamkniętości języków regularnych na dopełnienie), a następnie, sprawdzając pustość języka $L(\mathcal{B})$. Jak wiemy, języki bezkontekstowe nie są zamknięte na dopełnienia, więc ta metoda tu nie zadziała. Co więcej, jak pokażemy w Rozdziale 4, nie istnieje *żaden* algorytm, który by poprawnie rozwiązywał problem uniwersalności gramatyk dla wszystkich gramatyk (nie ma żadnych ograniczeń na czas działania algorytmu). Mówimy, że problem ten jest *nierozstrzygalny*.

Przy okazji wspomnimy tu, że problem pustości automatów *z dwoma stosami* jest nierozstrzygalny. Taki automat może niezależnie operować na obydwu stosach.

3.4.3 *Parsowanie słów względem gramatyki*

Rozważymy tu problem *parsowania* dla gramatyk:

Problem: CFL-MEMBERSHIP
Dane: gramatyka $\mathcal{G}$ nad alfabetem A oraz słowo $w \in A^*$
Rozstrzygnąć: czy $w \in L(\mathcal{G})$?

Twierdzenie 9. *Istnieje algorytm dla problemu parsowania gramatyk, działający w czasie*[14]

$$\mathcal{O}(|\mathcal{G}|^2 \cdot n^3),$$

dla gramatyki $\mathcal{G}$ oraz słowa wejściowego długości n.

[14] Przez $|\mathcal{G}|$ rozumiemy łączną długość wszystkich produkcji w gramatyce. Dla gramatyk w postaci normalnej Chomsky'ego opisanej poniżej, czas działania to $\mathcal{O}(|\mathcal{G}| \cdot n^3)$.

Idea jest prosta: kolejno dla $i = 1, 2, \ldots, |w|$, dla każdego infiksu u słowa w długości i obliczamy zbiór nieterminali N_u, z których da się wygenerować słowo u. Infiks długości i można na $i - 1$ sposobów rozbić jako konkatenację $u = u_1 \cdot u_2$ infiksów u_1, u_2 długości mniejszej niż i, dla których mamy już obliczone zbiory N_{u_1} i N_{u_2}. Na podstawie tych informacji, jesteśmy w stanie obliczyć zbiór N_u.

Postać normalna. Żeby powyższa idea zadziałała, wpierw przekształcamy gramatykę $\mathcal{G}$ do postaci *normalnej*[15], w której każda produkcja jest jednej z następujących postaci:

[15] pełna nazwa to *postać normalna Chomsky'ego*

$$\begin{aligned} q &\to pr \quad \text{gdzie } p, r \text{ są nieterminalami,} \\ q &\to a \quad \text{gdzie } a \text{ jest terminalem.} \end{aligned}$$

Można to zrobić, zamieniając każdą produkcję w gramatyce $\mathcal{G}$ postaci

$$q \to b_1 \cdots b_n$$

na zbiór produkcji

$$\begin{aligned} q &\to [q_1][q_2 \cdots q_n], \\ [q_2 \cdots q_n] &\to [q_2][q_3 \cdots q_n], \\ [q_3 \cdots q_n] &\to [q_3][q_4 \cdots q_n], \\ &\cdots\cdots \\ [q_i] &\to b_i, \end{aligned}$$

gdzie dla $i = 1, \ldots, n$ mamy nowe nieterminale $[q_i]$ oraz $[q_i \cdots q_n]$. Otrzymana gramatyka $\mathcal{H}$ generuje ten sam język, co $\mathcal{G}$. Następnie, otrzymaną gramatykę $\mathcal{H}$ przekształcamy do gramatyki $\mathcal{K}$, w której wszystkie produkcje są efektywne oraz $L(\mathcal{K}) = L(\mathcal{H}) - \{\varepsilon\}$. Ewentualna utrata słowa pustego ε w języku nie jest problemem, bo to słowo łatwo jest parsować osobno i nietrudno jest stwierdzić, czy gramatyka $\mathcal{G}$ generuje słowo puste[16]. Wynikowa gramatyka $\mathcal{K}$ jest w postaci normalnej.

[16] można na przykład z gramatyki $\mathcal{G}$ usunąć wszystkie produkcje w których pojawia się jakiś nieterminal, sprawdzić pustość wynikowej gramatyki

Przykład 37. Rozważmy gramatykę $\mathcal{G}$ generującą język $\{a^n b^n \mid n \geqslant 1\}$:

$$S \to aSb \mid ab$$

Przekształcając do postaci normalnej otrzymujemy następującą

gramatykę $\mathcal{K}$:

$$S \to AC \mid AB$$
$$C \to SB$$
$$A \to a$$
$$B \to b.$$

⌟

[17] Konwersja gramatyki $\mathcal{G}$ do postaci normalnej działa w czasie $\mathcal{O}(|\mathcal{G}|^2)$

Tak więc, możemy zakładać[17], że gramatyka $\mathcal{G}$ jest w postaci normalnej. Opisana powyżej idea teraz już działa i można ją łatwo zaimplementować:

function PARSE(grammar $\mathcal{G}$, word w) ▷ *$\mathcal{G}$ jest w postaci normalnej*
  $n \leftarrow |w|$
  T : macierz $n \times n$ podzbiorów Q, początkowo wypełniona $\emptyset$
  for $i \in [1, n]$ **do**
    for $(p \to a)$ production of $\mathcal{G}$ **do**
      if $w[i] = a$ **then**
        add p to $T[i, i]$
  for $l \in [2, n]$ **do** ▷ *długość infiksu*
    for $i \in [1, n - l + 1]$ **do** ▷ *początek infiksu*
      for $j \in (i, i + l)$ **do** ▷ *podział infiksu*
        for $(p \to qr)$ production of $\mathcal{G}$ **do**
          if $q \in T[i, j]$ *and* $r \in T[j + 1, i + l - 1]$ **then**
            add p to $T[i, i + l - 1]$
  return $(S \in T[1, n])$

Przykład 38. Rozważmy gramatykę $\mathcal{K}$ z Przykładu 37. Symulację algorytmu CYK na słowie *aabbb* można zobaczyć np. tutaj: https://www.xarg.org/tools/cyk-algorithm/. ⌟

Powyższy algorytm łatwo zmodyfikować tak, by konstruował drzewo wyprowadzenia słowa w, gdy takie istnieje.

[18] Twórcą tego algorytmu jest Leslie Valiant.

Istnieje teoretycznie szybszy algorytm parsowania[18], którego czas działania dla ustalonej gramatyki $\mathcal{G}$ jest taki sam, jak czas potrzebny do wymnożenia dwóch macierzy $n \times n$ o wartościach 0 lub 1. Infimum liczb $r \in \mathbb{R}$, dla których istnieje algorytm działający w czasie $\mathcal{O}(n^r)$ mnożący takie macierze jest oznaczany liczbą ω. Obecnie wiadomo, że $2 \leqslant \omega \leqslant 2.373$, jednak stałe ukryte w notacji $\mathcal{O}$ mogą być gigantyczne.

Dla wielu języków programowania istnieją znacznie szybsze parsery: są to często *deterministyczne* języki bezkontekstowe, tzn. są rozpoznawalne przez deterministyczny automat ze stosem. Sprawdzenie czy dane słowo należy do takiego języka można więc wykonać w czasie $\mathcal{O}(|w|)$, symulując deterministyczny automat ze stosem.

4
Teoria obliczeń

Czym są "automatyczne" obliczenia? Nieco precyzyjniej, dla jakich języków $L \subseteq A^*$ istnieje "automatyczna" lub "algorytmiczna" metoda stwierdzenia, czy dane słowo w należy do języka L? Definicja Alana Turinga z roku 1936, którą zaraz poznamy, ma na celu uchwycenie tego pojęcia za pomocą zwięzłej matematycznej definicji. Języki ją spełniającę nazywają się *rekurencyjnie przeliczalnymi*. Turing starał się wyobrazić jak wyglądają obliczenia rachmistrza, wykonującego pewne z góry określone rachunki – ma on nieograniczony dostęp do papieru, na którym może zapisywać znaki z jakiegoś ustalonego alfabetu. Zakładamy, że papier jest podzielony w kratki, i że w każdej kratce może być zapisana najwyżej jedna litera z ustalonego, skończonego alfabetu. Początkowo, na papierze jest zapisane słowo wejściowe w. W każdym momencie rachmistrz widzi jedną kratkę i może zmienić literę która się w niej znajduje, oraz przesunąć swój wzrok na następnej lub poprzedniej kratki. Dodatkowo, w każdym momencie rachmistrz jest w pewnym określonym stanie wewnętrznym, będącym jednym ze skończonej liczby stanów. Na podstawie swojego stanu wewnętrznego i zawartości widzianej kratki, rachmistrz podejmuje decyzję, jaką literę zapisać i w którą stronę przesunąć swój wzrok, oraz jak zmienić swój stan wewnętrzny. To wszystko ma się odbywać zgodnie z góry określonym, skończonym zbiorem tranzycji – regułami rachowania, specyficznymi dla wykonywanego rachunku. Dodatkowo, w dowolnym momencie, na podstawie aktualnego stanu wewnętrznego, rachmistrz może zdecydować się podać odpowiedź: tak lub nie. Oto cała idea definicji maszyny Turinga.

Przykład 39. Jest wiele dostępnych symulatorów maszyn Turinga, np. `http://morphett.info/turing/`. Można zobaczyć np. symulację maszyny rozpoznającej, czy dane słowo jest palindromem. ⌟

Z dzisiejszej perspektywy, być może naturalniej byłoby zdefiniować pojęcie języków obliczalnych za pomocą jakiegoś wysokopoziomowego języka programowania: język $L \subseteq A^*$ jest obliczalny jeżeli

istnieje program w języku C, który dostawszy na wejściu słowo w wypisuje na wyjściu tak lub nie, w zależności od tego, czy w należy do L czy nie. Nasze współczesne bogate doświadczenie z językami programowania mówi nam, że ta definicja *nie zależy* od wybranego języka programowania: jeśli byśmy wybrali język C++, Haskell, OCaml, PHP, JavaScript, Python czy BASIC, dostalibyśmy dokładnie tę samą klasę języków. To dlatego że doświadczony programista, dla każdych dwóch z powyższych języków A i B byłby w stanie napisać interpreter języka A w języku B, tzn. program I, który wczytuje program w języku A i tłumaczy go na równoważny program w języku B. Jednak za czasów Turinga języki programowania nie istniały.

Programowanie maszyn Turinga raczej przypomina niskopoziomowe języki programowania, typu asembler, z którymi dzisiaj rzadko programiści mają bezpośredni kontakt. Teoretycznie dałoby się zinterpretować dowolny z powyższych języków w maszynie Turinga, choć byłaby to bardzo żmudna robota – jak zobaczymy, programowanie w maszynach Turinga nie jest wygodne. Dlaczego więc wciąż je rozważamy? Poza powodem historycznym, jest istotniejszy powód: matematyczna defnicja maszyn Turinga jest o wiele prostsza niż definicja funkcjonalnego języka programowania. Przykładowo, formalna semantyka języka C – tj. precyzyjna, matematyczna definicja mówiąca co robi dany program – nie została nigdy zapisana. Ponadto, dzięki niskopoziomowości definicji Turinga, wiele pojęć, np. złożoność czasową i pamięciową, jest dużo łatwiej zdefiniować dla maszyn Turinga niż np. dla języka C, w którym wiele operacji jest transparentnych dla programisty (np. zarządzanie pamięcią, stosem wywołań, itd.). Motywacja Turinga była jednak inna. Jego głównym celem było udowodnienie, że nie istnieje algorytm stwierdzający, czy dane zdanie matematyczne jest prawdziwe. Jest to słynny problem nazywany *Entscheidungsproblem*, postawiony przez Davida Hilberta w roku 1928. Problem został rozwiązany (negatywnie) niezależnie przez Alonso Churcha i Alana Turinga w roku 1936. Żeby odpowiedzieć na to pytanie, wpierw należało podać matematyczną definicję algorytmu. Definicja Turinga opierała się na maszynach Turinga (podejście "imperatywne"), a Church wymyślił λ-rachunek (podejście "funkcyjne"). Natychmiast zauważono, że oba formalizmy są sobie równoważne. Nieformalna *teza Churcha-Turinga* mówi, że wszystkie sensowne formalizacje pojęcia algorytmu są równoważne maszynom Turinga czy λ-rachunkowi. Na tym wykładzie będziemy badali maszyny Turinga, bo jak wspomnieliśmy, łatwiej się dla nich definiuje zużycie czasu i pamięci. Udowodnimy przy ich pomocy twierdzenie Turinga, podając konkretne przykłady problemów, których nie da się rozwiązać algorytmicznie, m.in. problem pełności gramatyk bezkontekstowych.

4.1 *Maszyny Turinga i funkcje obliczalne*

Maszyna Turinga $\mathcal{M}$ ma następujące składniki:

- alfabet wejściowy A,
- alfabet roboczy B zawierający A, oraz do którego należy specjalny symbol $\llcorner$ oznaczający puste pole;

symbol $\llcorner$ czytamy "blank"

- zbiór stanów Q;
- stan początkowy $q_0 \in Q$;
- stan końcowy $q_{fin} \in Q$;
- funkcję przejścia $\delta\colon Q \times B \to B \times \{\leftarrow, \rightarrow\} \times Q$.

Tutaj rozważamy deterministyczne, jednotaśmowe maszyny Turinga. Później będziemy rozważali inne warianty: niedeterministyczne, wielotaśmowe itd.

Maszyna operuje na taśmie składającej się z (jednostronnie) nieskończonego ciągu komórek, przesuwając głowicę wzdłuż taśmy. W każdym momencie, jej głowica jest umieszczona nad jedną komórką. Maszyna może wykonać tranzycję $t = (p, a, b, d, q)$ jeżeli znajduje się w stanie p, w obecnej komórce taśmy widzi literę a. Wykonując tranzycję t, maszyna zapisuje w obecnej komórce literę b (zamazując poprzednią zawartość), przesuwa głowicę w lewo bądź w prawo, w zależności od kierunku $d \in \{\leftarrow, \rightarrow\}$, i przechodzi do stanu q.

Formalnie, *konfiguracja* maszyny Turinga $\mathcal{M}$ składa się z:

- zawartości taśmy u, która jest nieskończonym ciągiem liter postaci

$$b_1 b_2 \dots b_n \llcorner\llcorner\llcorner\llcorner\llcorner\llcorner\llcorner\llcorner\llcorner\llcorner \dots$$

  dla pewnego skończonego słowa $u = b_1 \cdots b_n \in B^*$,
- pozycji głowicy na taśmie, która jest liczbą naturalną $n \geqslant 1$,
- obecnego stanu $p \in Q$.

Oznaczmy powyższą konfigurację przez $[u, n, p]$. *Konfiguracją początkową* dla słowa wejściowego $w \in A^*$ jest konfiguracja $[w\llcorner\llcorner\llcorner \cdots, 1, q_0]$. *Konfiguracją końcową* jest dowolna konfiguracja postaci $[u, n, q_{fin}]$. Dla dwóch konfiguracji $c = [u, n, p], d = [v, m, q]$, gdzie q nie jest stanem końcowym, piszemy $c \to d$ jeżeli wykonanie jednej instrukcji maszyny $\mathcal{M}$ w konfiguracji c skutkuje przejściem do konfiguracji d. Formalnie, $c \to d$ jeżeli zachodzi następujący warunek:

(Rysunek).

$\delta(p, u[n]) = (b, d, q)$, gdzie $b \in B, d \in \{\leftarrow, \rightarrow\}$, oraz $m = n - 1$ gdy $d = \leftarrow$, $m = n + 1$ gdy $d = \rightarrow$, $v[n] = b$ oraz $v[k] = u[k]$ dla $k \geqslant 1$ oraz $k \neq n$.

(Rysunek tranzycji).

Biegiem maszyny po słowie w jest ciąg konfiguracji (skończony lub nie) postaci

$$c_0 \to c_1 \to c_2 \to c_3 \to \dots$$

gdzie c_0 jest konfiguracją początkową dla słowa w oraz każde dwie kolejne konfiguracje są w relacji $\to$. Bieg jest *terminujący* jeżeli się kończy w konfiguracji końcowej, tj. postaci $[u, n, q_{fin}]$. Jeżeli taki bieg istnieje, to mówimy, że maszyna *terminuje* na słowie w. Wówczas *wynikiem obliczenia* maszyny $\mathcal{M}$ na słowie w jest jedyne takie słowo $v \in B^*$, że v nie kończy się literą $\sqcup$ i zawartość taśmy w konfiguracji końcowej to słowo $v\sqcup\sqcup\sqcup\cdots$.

Przez $L(\mathcal{M})$ oznaczamy zbiór tych słów $w \in A^*$, na których maszyna $\mathcal{M}$ terminuje. Przez $f_{\mathcal{M}}(w)$ oznaczamy funkcję częściową $f_{\mathcal{M}}(w)\colon A^* \rightharpoonup B^*$ która słowu $w \in L(\mathcal{M})$ przypisuje wynik obliczenia maszyny $\mathcal{M}$ na słowie w, a dla pozostałych słów jest nieokreślona. Mówimy, że maszyna $\mathcal{M}$ *oblicza* funkcję częściową $f_{\mathcal{M}}$. Funkcja częściowa $f\colon A^* \rightharpoonup B^*$ jest *obliczalna* jeśli istnieje maszyna Turinga która ją oblicza.

Przykład 40. Funkcja odwracająca słowo $\mathsf{R}\colon A^* \to A^*$ oraz funkcja duplikująca słowo $dup\colon A^* \to A^*$ są obliczalne. Obliczalne są funkcje charakterystyczne języków: $\{w\#w \mid w \in \{a,b\}^*\}$ oraz $\{ww \mid w \in \{a,b\}^*\}$.

Funkcja zmieniająca kodowanie binarne na unarne i odwrotnie. Dla funkcji $\mathbb{N}^k \to \mathbb{N}^l$, wejście i wyjście reprezentujemy jako słowa nad alfabetem $\{0,1,\#\}$, i kodujemy liczby binarnie, oddzielając je symbolem #. Funkcje $+, -, \times, div : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ są obliczalne. Funkcje obliczalne są zamknięte na złożenia. Funkcje obliczalne są zamknięte na teorio-mnogościowe sumy (tj. jeżeli $f, g\colon A^* \rightharpoonup B^*$ są obliczalne i zgodne na części wspólnej dziedziny, to ich suma też jest funkcją obliczalną). ⌟

WYKŁAD 8

Funkcje obliczalne są głównym obiektem badań w teorii obliczeń; maszyny Turinga są tylko jednym z wielu równoważnych sposobów na ich wprowadzenie. Według przyjętej definicji, funkcje obliczalne to funkcje przekształcające słowa w słowa, tzn. postaci $f\colon A^* \rightharpoonup B^*$. To, że w tej definicji pojawiają się słowa jest kwestią drugorzędną, i trochę przypadkową. Wynika to z wyboru słów, dokonanego przez Turinga, jako uniwersalnego sposobu opisywania skończonych obiektów matematycznych. Można by zdefiniować funkcje obliczalne – i takie było podejście Churcha – jako funkcje *liczbowe*, postaci $f\colon \mathbb{N} \to \mathbb{N}$, albo funkcje przekształcające skończone grafy w skończone grafy, albo jeszcze w inny sposób[1].

Kluczowe jest, że są ustalone i – przynajmniej intuicyjnie – łatwo obliczalne sposoby kodowania słów jako liczby, liczb jako słowa, grafów jako słowa, zbiorów jako grafy, itd. Przykładowo, każdą liczbę $n \in \mathbb{N}$ możemy kodować jako słowo np. unarnie jako $[n]_1$ lub binarnie jako $[n]_2$. Z kolei, słowo $w = a_1a_2\ldots a_n$ nad alfabetem $\{0,1\}$ możemy reprezentować jako taką liczbę[2] $val_2(w)$, której binarna reprezentacja to $a_1a_1\ldots a_n$. Przy pomocy powyższej odpo-

[1] Z punktu widzenia teorii zbiorów, naturalnie jest rozważać funkcje na zbiorach *dziedzicznie skończonych*, tj. zbiorach skończonych, których elementy też są skończone, itd. Z punktu widzenia języków programowania opartych o listy, naturalnie jest rozważać funkcje operujące na dziedzicznych listach (tzn. na listach, których elementy są listami, których elementy są listami, itd.).

[2] jeśli ważne jest, by różne słowa miały różne reprezentacje, to do słowa w dopisujemy cyfrę 1 na początku, i reprezentujemy jako liczbę $val_2(1w)$

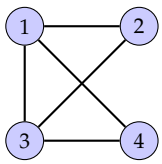
wiedniości, możemy zdefiniować pojęcie *obliczalnej funkcji liczbowej* $f\colon \mathbb{N} \to \mathbb{N}$ jako funkcji, dla których istnieje maszyna Turinga $\mathcal{M}$ która dostawszy na wejściu liczbę n zapisaną binarnie, oblicza na wyjściu reprezentację binarną liczby $f(n)$. Wybierając reprezentację unarną zamiast binarnej otrzymalibyśmy inną definicję obliczalnych funkcji liczbowych, która okazuje się jej być równoważna, z tego powodu, że przekształcenie zapisu unarnego do binarnego i odwrotnie są funkcjami obliczalnymi.

Dokonując tego typu translacji, możemy mówić przykładowo o *obliczalnych funkcjach na grafach*. Stosujemy wtedy jedno z dwóch standardowych opisów grafu za pomocą słów: za pomocą macierzy incydencji spłaszczonych do słów lub za pomocą list sąsiedztw (przy czym uznajemy, że wierzchołki grafu są liczbami naturalnymi).

Przykład 41. Funkcja, która dla danego słowa $w \in \{0,1\}^*$ zwraca 1 jeżeli jest ono opisem grafu spójnego (za pomocą macierzy incydencji) oraz 0 w przeciwnym przypadku, jest funkcją obliczalną. Idea jest taka, by symulować algorytm przeszukiwania wszerz (BFS, od *breadth-first search*). Na taśmie możemy zapisywać zbiór dotychczas odwiedzonych wierzchołków itd.

Zamiast spójności moglibyśmy tu podać inne warunki, np. eulerowskość lub hamiltonowskość. ⌟

Rozważmy poniższy graf.



Jego macierz incydencji to:

$$\begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0, \end{matrix}$$

a słowo ją opisujące to 0111101011011010. Reprezentacja powyższego grafu za pomocą list incydencji wyglądałaby mniej więcej tak:
$1 : (10, 11, 100), 10 : (11), 11 : (1, 10, 100), 100 : (1, 11)$, przy czym używamy tu alfabetu składającego się z nawiasów, dwukropka, przecinka, oraz cyfr $0, 1$.

4.1.1 Teza Churcha-Turinga

Teza Churcha-Turinga to nieformalne stwierdzenie mówiące, że dowolna formalizacja intuicyjnego pojęcia algorytmicznej obliczalności jest równoważne maszynom Turinga i λ-rachunkowi. Teza ta jest współcześnie mocno ugruntowana faktem, że wszystkie matematyczne modele komputerów[3] i języki programowania są – przy nieograniczonych zasobach – równoważne maszynom Turinga. W szczególności, różne warianty maszyn Turinga – wielotaśmowe, niedeterministyczne, z taśmą dwustronnie nieskończoną, z taśmą dwuwymiarową, itd. – są sobie wszystkie równoważne.

Maszyny wielotaśmowe. Rozważamy rozszerzenie maszyn Turinga, nazywane *maszynami wielotaśmowymi*, które są wyposażone w k taśm, gdzie $k \in \mathbb{N}$. Taka maszyna ma wiele głowic – po jednej dla każdej z k taśm – lecz tylko jeden stan kontrolny. W dowolnym momencie, maszyna podejmuje decyzje na podstawie zawartości komórek pod każdą z k głowic i obecnego stanu. W wyniku, pod każdą z głowic jest zapisywana jakaś litera (niezależnie), każda z głowic przesuwa się w jakąś stronę, oraz maszyna przechodzi do zadanego stanu. Formalnie, funkcja przejścia takiej maszyny jest postaci:

$$\delta\colon Q \times B^k \to (B \times \{\leftarrow, \rightarrow\})^k \times Q.$$

[3] To dotyczy także modeli korzystających z efektów kwantowych, które, choć trudne do zaimplementowania fizycznie, matematycznie są precyzyjnie opisane i badane.

Pierwszą z k taśm nazwiemy *taśmą wejściową* – na niej początkowo będzie zapisane słowo wejściowe – a ostatnią z k taśm nazwiemy *taśmą wyjściową* – na niej pojawi się ostateczny wynik obliczenia maszyny.

Nieco dokładniej, maszyna k taśmowa $\mathcal{M}$, działa na danym słowie wejściowym $w \in A^*$ następująco. Maszyna zaczyna w konfiguracji, w której na pierwszej taśmie jest zapisane słowo w, a pozostałe taśmy są puste (tj. wypełnione samymi literami $\lrcorner$). Wszystkie głowice ustawione są na pierwszych komórkach odpowiednich taśm. Następnie, uruchamiamy maszynę $\mathcal{M}$. Jeżeli w pewnym momencie swojego biegu, maszyna przejdzie do stanu końcowego, to mówimy, ze $\mathcal{M}$ terminuje na słowie w, oraz że dla tego słowa wejściowego, wynikiem maszyny jest słowo v znajdujące się na taśmie wyjściowej (z usuniętymi końcowymi literami $\lrcorner$). W ten sposób, maszyna $\mathcal{M}$ definiuje funkcję częściową

$$f \colon A^* \rightharpoonup B^*.$$

Mówimy, że funkcja częściowa f jest *obliczana* przez maszynę wielotaśmową $\mathcal{M}$.

Rysunek

Twierdzenie 10. *Niech A oraz B będą dwoma alfabetami. Wówczas następujące warunki są równoważne dla funkcji częściowej $f \colon A^* \rightharpoonup B^*$:*

1. *funkcja f jest obliczana przez pewną maszynę k-taśmową $\mathcal{M}$, dla pewnej liczby $k \in \mathbb{N}$*
2. *funkcja f jest obliczalna, tj. jest obliczana przez pewną maszynę jednotaśmową $\mathcal{N}$.*

Szkic dowodu. Implikacja 2→1 jest trywialna: maszyna jednotaśmowa jest maszyną k-taśmową dla $k = 1$.

Pokażemy implikację 1→2. Niech $\mathcal{M}$ będzie maszyną k-taśmową obliczającą funkcję $f \colon A^* \to B^*$. Bez straty ogólności możemy założyć[4], że alfabet roboczy maszyny $\mathcal{M}$ to B. Konstruujemy maszynę jednotaśmową $\mathcal{N}$, która "symuluje" maszynę $\mathcal{M}$. Maszyna $\mathcal{N}$ ma alfabet roboczy $B' = \{b \mid b \in B\} \cup \{\underline{b} \mid b \in B\} \cup Q \cup \{\#, @\}$. Tutaj, $\underline{b}$ to "podkreślona litera" b. Dla słowa $w \in B^*$ oraz liczby n takiej, że $1 \leqslant n \leqslant w$, niech $t(w, n)$ oznacza słowo w, w którym n-tą litera jest podkreślona. Dla każdego kroku maszyny $\mathcal{M}$, maszyna $\mathcal{N}$ dokona wielu kroków, tak, żeby zachowany był następujący niezmiennik.

[4] jeśli alfabet roboczy to $C \supset B$, to traktujemy funkcję f jako funkcję $f \colon A^* \rightharpoonup C^*$.

> Jeżeli aktualnie maszyna $\mathcal{M}$ ma na i-tej taśmie słowo $w_i \in B^*$, jej i-ta głowica jest ustawiona na pozycji n_i, oraz jest w stanie q, to maszyna $\mathcal{N}$ ma na taśmie zapisane słowo
>
> $$w = q@t(w_1, n_1)\#t(w_2, n_2)\# \ldots \#t(w_k, n_k)@.$$

Innymi słowy, maszyna $\mathcal{N}$ utrzymuje na swojej taśmie opis aktualnej konfiguracji maszyny $\mathcal{M}$.

Żeby utrzymać powyższy niezmiennik, należy tak zaprogramować maszynę $\mathcal{N}$, żeby potrafiła zasymulować każdą pojedynczą instrukcję maszyny $\mathcal{M}$. Pojedyncze instrukcje maszyny $\mathcal{M}$ polegają na zapisaniu odpowiednich znaków pod każdą z głowic, oraz na przesunięciu każdej z głowic w lewo lub w prawo, i przejściu do nowego stanu, zależnego od poprzedniego stanu oraz znaków pod głowicami.

Maszyna $\mathcal{N}$ działa w kilku krokach.

1. Stwierdź, która tranzycja zostanie kolejno dokonana przez maszynę $\mathcal{M}$. W tym celu, maszyna $\mathcal{N}$ wczytuje pierwszy znak na swojej taśmie (to jest, aktualny stan q maszyny $\mathcal{N}$) oraz kolejno odszukuje wszystkie podkreślone litery na taśmie, i zapamiętuje je w swoim stanie.

   Maszyna $\mathcal{N}$ będzie mieć stany postaci $qb_1 \dots b_i \in Q \cdot B^*$, gdzie $0 \leqslant i \leqslant k$. Będąc w stanie $qb_1 \dots b_i$, maszyna przesuwa głowicę w prawo do momentu napotkania kolejnej podkreślonej litery. Jeżeli ta litera to $\underline{b}$, to maszyna przechodzi do stanu $qb_1 \dots b_i b_{i+1}$, gdzie $b_{i+1} = b$, oraz kontynuuje działanie tak długo, jak $i < k$. Gdy $i = k$, maszyna przesuwa się na początek taśmy i przechodzi do następnego kroku, opisanego poniżej.

   Przypuśćmy, że maszyna $\mathcal{M}$ ma tranzycję:

   $$\delta(q, b_1 b_2 \dots b_k) = ((c_1, d_1), (c_2, d_2), \dots, (c_k, d_k), q'),$$

   gdzie $c_1, \dots, c_k \in B$ oraz $d_1, \dots, d_k \in \{\leftarrow, \rightarrow\}$.

   Krotkę $((c_1, d_1), \dots, (c_k, d_k), q')$ maszyna $\mathcal{N}$ będzie przechowywać w aktualnym stanie.

2. Zastąp podkreślone znaki odpowiednimi znakami. Dokładniej, i-ty z kolei podkreślony znak $\underline{b_i}$ jest zastąpiony znakiem $\underline{c_i}$.

3. Przesuń podkreślenia w odpowiednie strony. Dokładniej, i-te podkreślenie jest przesunięte w lewo, jeżeli $d_i = \leftarrow$ oraz litera z lewej jest różna od @ oraz #. Jeżeli jest to jedna z liter @, #, to maszyna kończy przechodzi do stanu q_{reject} w którym już nic nie będzie robić. Z kolei, jeśli $d_i = \rightarrow$, to i-te podkreślenie należy przesunąć w prawo. Tego można łatwo dokonać, jeżeli litera z prawej jest różna od # oraz od @. Jeżeli zaś jest to jedna z tych dwóch liter, to należy zrobić miejsce, przesuwając całą resztę słowa o jeden w prawo, i zapisując w docelowym miejscu literę $\underline{\sqcup}$ (podkreślony $\sqcup$).

4. Zamień pierwszą literę q na taśmie wejściowej na literę q'.

Po wykonaniu powyższych kroków, maszyna $\mathcal{N}$ ma na taśmie reprezentację kolejnej konfiguracji maszyny $\mathcal{M}$.

W momencie, gdy na pierwszej pozycji taśmy pojawi się stan akceptujący maszyny $\mathcal{M}$, maszyna $\mathcal{N}$ jeszcze musi "posprzątać": zastępuje wówczas słowo $q@t(w_1, n_1)\#t(w_2, n_2)\#\ldots\#t(w_k, n_k)@$, które aktualnie jest zapisane na jej taśmie, słowem w_k, oraz przechodzi do stanu akceptującego i kończy bieg. ■

Od tej pory, o ile nie wskażemy inaczej, przez maszynę Turinga będziemy rozumieć maszynę wielotaśmową.

While-programy. Zilustrujemy tezę Churcha-Turinga na przykładzie *while-programów*. *While-program* to ogólny termin na bardzo proste programy które używają tylko pętli while, a nie używają rekurencji. Przykładowa ich składnia zawiera następujące składniki:

- Skończony zbiór zmiennych V o wartościach całkowitoliczbowych,
- Instrukcje bazowe: $x := y + z$, $x := y - z$, $x := 1$, $x := 0$, gdzie x, y to są zmienne.

Ponadto, jeśli I, J są instrukcjami, to instrukcjami też są:

- Instrukcja **if** x **then** I, instrukcja **if** x **then** I **else** J oraz instrukcja **while** x **do** I, gdzie x jest zmienną,
- Instrukcja sekwencyjna $I; J$.

Zdefiniujemy teraz semantykę takiego języka programowania. *Stan* to funkcja S przypisująca liczby zmiennym programu, tzn. $S\colon V \to \mathbb{N}$, lub równoważnie, $S \in \mathbb{N}^V$. Semantyka instrukcji I jest to funkcja częściowa $[\![I]\!]\colon \mathbb{N}^V \rightharpoonup \mathbb{N}^V$ która przekształca stany w stany. Jest ona definiowana przez indukcję po budowie instrukcji I, następująco:

Może być, że stan $[\![I]\!](S)$ jest nieokreślony dla niektórych stanów S (tak się dzieje gdy instrukcja się pętli). Jeżeli w definicji obok, stan po prawej jest niezdefiniowany, to wynik też jest niezdefiniowany.

$$[\![I; J]\!](S) = [\![J]\!]([\![I]\!](S))$$

$$[\![\textbf{if } x \textbf{ then } I \textbf{ else } J]\!](S) = \begin{cases} [\![I]\!](S) & \text{jeżeli } S(x) > 0 \\ [\![J]\!](S) & \text{jeżeli } S(x) = 0 \end{cases}$$

$$[\![\textbf{while } x \textbf{ do } I]\!](S) = \begin{cases} S & \text{jeżeli } S(x) = 0 \\ [\![\textbf{while } x \textbf{ do } I]\!]([\![I]\!](S)) & \text{jeżeli } S(x) > 0. \end{cases}$$

Bazą induktywnej definicji jest semantyka instrukcji bazowych, zdefiniowana w oczywisty sposób.

While-program P to jest dowolna instrukcja I używająca zbioru zmiennych V. Wyróżniamy dwie zmienne spośród zmiennych V, które nazwiemy *zmienną wejściową* i *zmienną wyjściową*. Semantyką takiego programu jest funkcja częściowa $f_P\colon \mathbb{N} \rightharpoonup \mathbb{N}$ której wynik na wejściu $n \in \mathbb{N}$ jest otrzymany następująco:

1. Zdefiniuj wartościowanie $S \in \mathbb{N}^V$, przez przypisanie zmiennej wejściowej wartości n, a pozostałym zmiennym – wartość n;

2. Zastosuj do S semantykę instrukcji P, otrzymując wartościowanie $T = [\![P]\!](S) \in \mathbb{N}^V$;

3. Wynik $f_P(n)$ jest równy wartości T na zmiennej wyjściowej.

(Rysunek)

Twierdzenie 11. *Dla każdego while-programu I istnieje wielotaśmowa maszyna Turinga $\mathcal{M}_I$ obliczającą funkcję $f_P \colon \mathbb{N} \rightharpoonup \mathbb{N}$.*

Szkic dowodu. Pokazujemy to przez indukcję po budowie instrukcji I. Gdy $I = J; K$, wystarczy sekwencyjnie złożyć maszyny otrzymane z założenia indukcyjnego. Podobnie, w pozostałych przypadkach: używamy definicji semantyki $[\![I]\!]$ oraz założenia indukcyjnego. Bazą indukcji są instrukcje bazowe, dla których ręcznie konstruujemy maszyny Turinga. ∎

Terminologia. Spojrzenie na maszyny Turinga jako na urządzenia implementujące algorytmy prowadzi do stosowania następującej terminologii, bardziej intuicyjnej od rygorystycznej terminologii języków i maszyn Turinga, co czasem odbywa się kosztem precyzji.

Funkcja $f \colon A^* \to B^*$ często nazywana jest *problemem obliczeniowym*, podczas gdy język $L \subseteq A^*$ często nazywany jest *problemem decyzyjnym*. Tych terminów używa się w kontekście, gdy należy opisać algorytm który *rozwiązuje* dany problem obliczeniowy bądź decyzyjny, tj. algorytm który – w pierwszym przypadku – dla danego wejścia $w \in A^*$ *oblicza* wynik $f(w)$, lub – w drugim przypadku – *decyduje* bądź *rozstrzyga*, czy $w \in L$, czy też $w \notin L$ (taki algorytm ma zawsze terminować). Często, specyfikację problemu opisuje się nie w postaci języka, tylko zadania, jak w poniższym przykładzie.

Problem: HAMILTONICITY
Dane: graf G
Rozstrzygnąć: czy G zawiera ścieżkę Hamiltona?

Instancją problemu jest dany na wejściu opis – w powyższym przypadku, przypuszczalnie jest to opis grafu[5] przy pewnej ustalonej reprezentacji grafów. Zakładamy, że język zdefiniowany przez taki opis problemu decyzyjnego składa się dokładnie z tych instancji, dla których odpowiedź na pytanie brzmi "TAK"; te instancje są nazywane czasem tak-instancjami. Należy zwrócić uwagę, że tego typu opis problemu nie wyznacza odpowiadającego mu języka w sposób zupełnie jednoznaczny[6]. Ta niejednoznaczność nie powinna mieć jednak wpływu na badane zagadnienie (np. obliczalność).

W dowodach, często będziemy się odwoływać do wyżej opisanej, bardziej intuicyjnej terminologii. Również opisując algorytmy, nie zawsze będziemy definiowali precyzyjnie maszyny Turinga, lecz

[5] są też instancje źle uformowane, które nie opisują żadnego grafu. Zazwyczaj istotne jest, by instancje źle uformowane dało się łatwo rozpoznać, tj. by istniała maszyna Turinga rozstrzygająca, czy dana instancja jest dobrze uformowana, czy nie.

[6] w szczególności, nie jest jasne, jak zakodowany jest graf G. W praktyce, stosowane są dwa standardowe kodowania grafów (przez macierz incydencji lub przez listy sąsiedztwa), i dla większości problemów, wybór jednego lub drugiego nie wpływa na naturę badanego problemu.

zazwyczaj poprzestaniemy na w miarę dokładnym pseudokodzie lub opisie za pomocą słów. Należy jednak mieć na uwadze, by wszystkie opisy dało się w pełni sformalizować – opisywane problemy obliczeniowe mają opisywać języki, a algorytmy – maszyny Turinga, w taki sposób, żeby niejednoznaczności wynikające z różnych interpretacji opisu nie miały wpływu na poprawność argumentacji.

Tablica 4.1: Słownik terminologii.

Teoria obliczeń	Algorytmika
maszyna Turinga	algorytm
słowo	instancja
język	problem decyzyjny
— obliczalny	— rozstrzygalny
— częściowo obliczalny (lub rekurencyjnie przeliczalny)	— półrozstrzygalny
funkcja	problem obliczeniowy
— obliczalna	— da się rozwiązać algorytmem

4.2 *Języki obliczalne i częściowo obliczalne*

Język $L \subseteq A^*$ jest *obliczalny*, jeżeli jego funkcja charakterystyczna[7] jest obliczalna. Inaczej mówiąc, istnieje maszyna Turinga, która dla dowolnego słowa wejściowego $w \in A^*$ terminuje, oraz zwraca w wyniku 1 lub 0, w zależności od tego, czy $w \in L$ czy też $w \notin L$.

[7] zadana wzorem $f(w) = 1$ dla $w \in L$ oraz $f(w) = 0$ dla $w \notin L$.

W terminologii problemów decyzyjnych, jeżeli język L jest obliczalny, to mówimy że odpowiadający mu problem decyzyjny jest *rozstrzygalny*. A zatem, problem decyzyjny jest rozstrzygalny jeśli istnieje algorytm który dla każdego słowa wejściowego w terminuje, i odpowiada "TAK"/"NIE" w zależności od tego, czy $w \in L$ czy nie.

Przykład 42. Języki regularne są obliczalne. Języki bezkontekstowe są obliczalne. ⌟

Lemat 15. *Języki obliczalne są zamknięte na:*

- *Sumy, przecięcia, dopełnienia,*
- *Przeciwobrazy przy funkcjach obliczalnych: jeśli $f\colon A^* \rightharpoonup B^*$ jest funkcją obliczalną oraz $L \subseteq B^*$ jest językiem obliczalnym, to język $f^{-1}(L)$ jest obliczalny.*

Dowód. Ćwiczenie. ■

Języki częściowo obliczalne. Język $L \subseteq A^*$ jest *częściowo obliczalny* jeżeli istnieje maszyna Turinga $\mathcal{M}$ która terminuje dla wszystkich słów $w \in L$, a dla wszystkich słów $w \in A^* - L$ nie terminuje. Każdy język obliczalny jest też częściowo obliczalny, bo zamiast zwracać 0, maszyna może się wieszać.

W terminologii problemów decyzyjnych, jeżeli język L jest częściowo obliczalny, to mówimy że odpowiadający mu problem decyzyjny jest *półrozstrzygalny*. A zatem, problem decyzyjny jest półrozstrzygalny jeśli istnieje algorytm który dla każdego słowa wejściowego w odpowiada "TAK" dokładnie wtedy, gdy $w \in L$, a dla pozostałych słów wejściowych $w \notin L$ algorytm może się wieszać.

Podobnie jak w Lemacie 15, mamy następujący wynik.

Lemat 16. *Języki częściowo obliczalne są zamknięte na sumy, przecięcia, obrazy i przeciwobrazy przy funkcjach obliczalnych.*

Dowód. Ćwiczenie. ∎

Fakt 1. *Język $L \subseteq A^*$ jest obliczalny wtedy i tylko wtedy, gdy zarówno L jak i $A^* - L$ są językami częściowo obliczalnymi.*

Innymi słowy, powyższy fakt mówi, że aby rozstrzygnąć problem obliczeniowy $L \subseteq A^*$, wystarczy podać dwie tzw. *półprocedury*:

- algorytm, który terminuje dokładnie dla słów $w \in L$,
- algorytm, który terminuje dokładnie dla słów $w \notin L$.

Z tych półprocedur można skonstruować algorytm, który odpowiada na pytanie, czy $w \in L$, symulując jednocześnie działanie obydwu półprocedur, aż do momentu, gdy pierwsza z nich zakończy.

Jak zobaczymy w Rozdziale 4.3, istnieją języki które są częściowo obliczalne, ale nie są obliczalne. A zatem, języki częściowo obliczalne *nie są* zamknięte na dopełnienia.

Dowód faktu 1. Implikacja z lewej w prawo jest natychmiastowa: jeśli L jest obliczalny, to również $A^* - L$ jest obliczalny, a każdy język obliczalny jest częściowo obliczalny,

Przypuśćmy, że języki L i $A^* - L$ są częściowo obliczalne. A zatem funkcja częściowa $f\colon A^* \rightharpoonup \{0,1\}^*$ taka, że $f(w) = 1$ dla $w \in L$ oraz $f(w)$ jest nieokreślone dla $w \notin L$ jest obliczalna. Podobnie, funkcja częściowa $g\colon A^* \rightharpoonup \{0,1\}^*$ taka, że $f(w) = 0$ dla $w \in L - A^*$ oraz $g(w)$ jest nieokreślone dla $w \in L$ jest obliczalna. Suma teoriomnogościowa funkcji f i g to funkcja charakterystyczna języka L, i jest obliczalna, bo jest sumą dwóch funkcji obliczalnych. ∎

Przykład 43. (Przykład) ⌟

Enumeratorem nazwiemy maszynę Turinga o jednej taśmie roboczej i jednej taśmie wyjściowej po której maszyna przesuwa się tylko w prawo i wypisuje symbole z alfabetu $A \cup \{\#\}$. Mówimy, że taka maszyna *wypisuje* słowo $w \in A^*$ jej bieg na słowie pustym w pewnym momencie wypisuje na taśmie wyjściowej słowo $\#w\#$. Jeżeli $L \subseteq A^*$ jest zbiorem wszystkich słów wypisywanych przez enumerator $\mathcal{M}$, to mówimy, że enumerator $\mathcal{M}$ *enumeruje* język L. Język, dla którego istnieje enumerator, nazywamy *rekurencyjnie przeliczalnym*[8].

[8] po angielsku, *recursively enumerable*

Fakt 2. *Języki rekurencyjnie przeliczalne to są dokładnie języki częściowo obliczalne.*

Dowód. Przypuśćmy, że język $L \subseteq A^*$ jest rekurencyjnie przeliczalny i niech $\mathcal{M}$ będzie jego enumeratorem. Opiszemy maszynę $\mathcal{N}$ która terminuje dokładnie dla słów należących do języka L. Maszyna ta, dostawszy na wejściu słowo w, symuluje działanie maszyny $\mathcal{M}$ na słowie pustym tak długo, aż na taśmie wyjściowej pojawi się słowo w;

wówczas maszyna $\mathcal{N}$ terminuje. Tak więc, gdy słowo w nie należy do języka L, to maszyna $\mathcal{N}$ nie terminuje, bo czeka w nieskończoność aż pojawi się słowo w. Zatem język L jest częściowo obliczalny.

Teraz załóżmy, że język L jest częściowo obliczalny. Niech $\mathcal{N}$ będzie maszyną która terminuje na wejściu w dokładnie wtedy, gdy $w \in L$. Skonstruujemy maszynę $\mathcal{N}$ która enumeruję język L. Maszyna ta działa w kolejnych fazach o numerach $1, 2, 3, \ldots$, gdzie faza n wygląda następująco:

> kolejno dla każdego słowa w długości co najwyżej n nad alfabetem A (w kolejności leksykograficznej), zasymuluj działanie maszyny $\mathcal{N}$ na słowie w wykonując tylko n kroków jej obliczeń. Jeśli w przeciągu tych n kroków, maszyna $\mathcal{N}$ osiągnęła stan końcowy, to wypisz słowo w na taśmie wyjściowej.

Każda faza zajmuje jedynie skończony czas, gdyż jest tylko skończenie wiele słów długości co najwyżej n, i na każdym z nich jest wykonywane jedynie skończone obliczenie. Po wykonaniu fazy n, maszyna $\mathcal{M}$ wykonuje fazę $n+1$, i tak w nieskończoność.

Niech K będzie językiem enumerowanym przez maszynę $\mathcal{M}$. Pokażemy, że $K = L$. Jasne jest, że $K \subseteq L$, bo każde słowo wypisane przez maszynę $\mathcal{M}$ jest wypisane dlatego, że maszyna $\mathcal{N}$ na nim terminuje. Żeby pokazać drugą inkluzję, $K \supseteq L$, rozważmy dowolne słowo $w \in L$. A zatem, maszyna $\mathcal{N}$ akceptuje słowo w po pewnej liczbie kroków k. Niech n będzie równe $\max(|w|, k)$. Wtedy słowo w jest wypisane przez maszynę $\mathcal{M}$ w fazie n. To dowodzi, że $K = L$, czyli że maszyna $\mathcal{M}$ enumeruje język L. ∎

Twierdzenie 12. *Następujące warunki są równoważne dla języka $L \subseteq A^*$:*

- *L jest częściowo obliczalny,*
- *L jest obrazem całkowitej funkcji obliczalnej $f\colon B^* \to A^*$, t.j. $L = \{f(w) \mid w \in A^*\}$, dla pewnego alfabetu B.*

(Rysunek: języki obliczalne: przeciwobrazy $\{1\}$ przy funkcjach obliczalnych $f\colon A^* \to \{0,1\}$; języki częściowo obliczalne: obrazy funkcji obliczalnych $f\colon B^* \to A^*$.)

WYKŁAD 9

4.3 Nierozstrzygalność

Ponieważ maszyn Turinga jest jedynie przeliczalna ilość, a języków jest nieprzeliczalnie wiele, następujący fakt nie powinien być zaskakujący:

Fakt 3. *Istnieją języki nieobliczalne.*

Dowód. Pokażemy, że istnieją nieobliczalne języki nad alfabetem unarnym $A = \{1\}$. Niech M będzie zbiorem wszystkich maszyn Turinga nad alfabetem wejściowym A, które terminują dla każdego słowa wejściowego w i zwracają w wyniku 0 lub 1; jest to zbiór przeliczalny[9]. Gdyby każdy język $L \subseteq A^*$ był obliczalny, to funkcja $L\colon M \to P(A^*)$ taka, że $\mathcal{M} \mapsto \{w \in A^* \mid \mathcal{M}(w) = 1\}$, byłaby na $P(A^*)$. To jest niemożliwe, bo nie istnieje funkcja z przeliczalnego zbioru M na nieprzeliczalny zbiór $P(A^*)$. ∎

[9] Dla każdej liczby $n \in \mathbb{N}$, zbiór $M_n \subseteq M$ składający się z tych maszyn, których zbiór stanów i alfabet roboczy są zawarte w $\{1, \ldots, n\}$ jest skończony. A zatem, zbiór $M = \bigcup_{n \geqslant 1} M_n$ jest przeliczalny.

Powyższy dowód nie daje nam jednak opisu konkretnego języka nieobliczalnego. Żeby taki język skonstruować, przypomnijmy sobie dowód twierdzenia Cantora, pokazujący, że nie istnieje funkcja różnowartościowa z $P(\mathbb{N})$ w $\mathbb{N}$.

4.3.1 Metoda przekątniowa

W pierwszych latach XX w. Bertrand Russell posługiwał się następującą historią o fryzjerze, by ilustrować pewne problemy występujące w próbach rozwijania teorii zbiorów.

Paradoks Russella. W miejscowości M jest fryzjer, nazwijmy go *superfryzjerem*, który strzyże tych i tylko tych mieszkańców miejscowości, którzy nie strzygą siebie samych. Czy superfryzjer strzyże siebie samego? Chwila namysłu pokazuje, że obie możliwości są wykluczone: nie może on strzyc siebie samego, bo strzyże tylko tych, którzy siebie sami nie strzygą; gdyby zaś sam się nie strzygł, to musiałby się strzyc, bo strzyże wszystkich tych, którzy sami się nie strzygą. A zatem, superfryzjer nie może istnieć! Pokażemy jak z powyższego faktu otrzymać różne twierdzenia matematyczne, odpowiednio definiując mieszkańców miejscowości M oraz to, kto kogo strzyże.

Twierdzenie Cantora. Paradoks Russella był zainspirowany dziesięć lat starszą metodą przekątniową Georga Cantora, którą teraz przypomnimy.

Niech $c_1, c_2, c_3, \ldots$ będzie ciągiem liczb rzeczywistych z przedziału $(0, 1)$. Zdefiniujmy liczbę $x \in (0, 1)$ następująco: jej n-ta cyfra to 1 jeśli n-ta cyfra liczby c_n to 0, i 0 w przeciwnym przypadku. Pokażemy, że x nie pojawia się w ciągu: gdyby $x = c_k$ dla pewnego k, to k byłoby superfryzjerem w miejscowości M zamieszkałą przez liczby naturalne, w której m strzyże n wtedy, i tylko wtedy, gdy n-ta cyfra liczby c_m to 1. Sprzeczność.

Twierdzenie Turinga. Wybierzmy swój ulubiony język programowania, np. Java, C++, Pascal czy Python. *Kod źródłowy programu* jest to napis używający znaków dostępnych na klawiaturze komputera,

który jest zgodny ze składnią wybranego języka. Rozważmy programy których kod źródłowy jest szczególnej postaci $(*)$: pierwszą wykonywaną instrukcją jest "wczytaj napis N wprowadzony przez użytkownika", potem następuje ciąg instrukcji niedokonujących interakcji z użytkownikiem, a ostatnią instrukcją programu jest wypisanie słowa "koniec". Z punktu widzenia nieśmiertelnego użytkownika który uruchamia taki program i wprowadza napis N, są dwa możliwe scenariusze: albo program po pewnym czasie napisze "koniec" – mówimy wtedy, że program *akceptuje* napis N – albo nigdy nic nie napisze, tzn. "zawiesi się". Dla wielu programów, wynik działania na danym napisie N bardzo łatwo przewidzieć. Można by się pokusić się o napisanie programu Q, który dostaje na wejściu kod źródłowy dowolnego programu P postaci $(*)$ oraz napis N, po czym napisze "wiesza się" jeżeli program P się wiesza dostawszy na wejściu N, oraz napisze "akceptuje" w przeciwnym przypadku. Twierdzenie Turinga, które teraz udowodnimy, mówi, że taki program Q nie istnieje. Gdyby istniał, to skonstruowalibyśmy program F który, dostawszy kod dowolnego programu P, akceptuje go wtedy, i tylko wtedy, gdy program P uruchomiony na wejściu P się wiesza, co można stwierdzić, uruchamiając program Q na parze P, P. Wtedy F byłby superfryzjerem w miejscowości M zamieszkałej przez kody źródłowe postaci $(*)$, w której P strzyże K jeśli program P akceptuje K. Sprzeczność.

Sformalizujemy teraz powyższy szkicowy dowód, w języku maszyn Turinga.

4.3.2 *Problem stopu*

Rozważmy następujący problem decyzyjny, nazywany czasem problemem *stopu*:

Problem: HALT
Dane: Jednotaśmowa maszyna Turinga $\mathcal{M}$ oraz słowo w
Rozstrzygnąć: Czy $\mathcal{M}$ terminuje na słowie w?

Pokażemy, że ten problem jest nierozstrzygalny. Formalnie, rozważamy język HALT $\subseteq \{0,1,\$\}^*$ składający ze słów postaci $kod\$w$, gdzie $kod \in \{0,1\}^*$ jest kodem jednotaśmowej maszyny Turinga $\mathcal{M}$ i $w \in \{0,1\}^*$ jest słowem wejściowym, oraz maszyna $\mathcal{M}$ zatrzymuje się na słowie w. Pozostaje sprecyzować sposób kodowania maszyn Turinga. Możliwych kodowań jest wiele; poniżej podajemy jeden, arbitrarnie wybrany.

Niech $\mathcal{M}$ będzie maszyną Turinga o alfabecie wejściowym $A = \{a_0, a_1, \ldots, a_k\}$, alfabecie roboczym $B = \{b_0, \ldots, b_l\}$, przy czym $b_l = \sqcup$, $k \geqslant l$ oraz $b_i = a_i$ dla $i = 0, \ldots, k$, o zbiorze stanów $Q =$

$\{q_0, \ldots, q_m\}$, stanie początkowym q_0 oraz stanie akceptującym q_1. Wówczas *kodem* maszyny $\mathcal{M}$ jest słowo

$$1^k 0 1^l 0 1^m 0 w_\delta \in \{0,1\}^*$$

gdzie w_δ jest konkatenacją (w dowolnej kolejności) wszystkich krotek $1^i 0 1^j 0 1^{i'} 0 1^{j'} 0 1^d$ takich, że $\delta(q_i, b_j) = (q_{i'}, b_{j'}, d)$, gdzie $1^{\leftarrow}$ oznacza 1 oraz $1^{\rightarrow}$ oznacza 11.

Udowodnimy:

Twierdzenie 13. *Język* HALT *jest częściowo obliczalny, ale nie jest obliczalny.*

4.3.3 Maszyna uniwersalna

Udowodimy, że język HALT jest częściowo obliczalny. W tym celu pokażemy, że maszyny Turinga są na tyle potężne, że są w stanie symulować inne maszyny Turinga. Dokładniej, skonstruujemy maszynę Turinga $\mathcal{U}$, nazywaną *uniwersalną maszyną Turinga*, która rozwiązuje następujący problem obliczeniowy:

Problem: INTERPRET
Dane: kod maszyny $\mathcal{M}$ oraz słowo $w \in \{0,1\}^*$
Obliczyć: wynik działania maszyny $\mathcal{M}$ na słowie w, jeśli on istnieje (w przeciwnym przypadku, wieszaj się)

Maszyna uniwersalna. Pokażemy teraz, że istnieje maszyna rozwiązująca następujący problem INTERPRET. Jest to sformalizowane w poniższym lemacie.

Lemat 17. *Istnieje jednotaśmowa maszyna Turinga $\mathcal{U}$ która otrzymawszy na wejściu słowo $u \in (A \cup \{0,1,\#,\$\})^*$, terminuje wtedy i tylko wtedy, gdy u jest postaci kod\$w, gdzie kod jest kodem maszyny Turinga $\mathcal{M}$ a $w \in \{0,1,\#\}^*$ jest takim słowem, że $\mathcal{M}$ terminuje na słowie wejściowym w. Ponadto, gdy $\mathcal{U}$ terminuje, to na taśmie znajduje się wynik obliczenia maszyny $\mathcal{M}$ na słowie w.*

Dowód. Idea jest prosta: maszyna $\mathcal{U}$ sprawdza, że u jest poprawnej postaci, po czym symuluje maszynę $\mathcal{M}$ na wejściu w. W każdym kroku symulacji, zagląda do funkcji przejść maszyny $\mathcal{M}$, którą ma zapisaną na taśmie wejściowej, by wykonać kolejny krok. Opiszemy konstrukcję nieco formalniej.

Skonstruujemy maszynę *wielotaśmową* $\mathcal{V}$ która otrzymuje dwa wejścia, u oraz w, i terminuje wtedy i tylko wtedy, gdy spełniony jest warunek opisany w tezie lematu. Z tego już lemat wynika, bo maszynę wielotaśmową $\mathcal{V}$ możemy zamienić na równoważną jej jednotaśmową maszynę $\mathcal{U}$, stosując Twierdzenie 10.

Maszyna $\mathcal{V}$ ma trzy taśmy – dwie pierwsze są wejściowe a trzecia robocza. Na pierwszej taśmie wejściowej podany jest słowo u które ma opisywać pewną maszynę Turinga. W pierwszej kolejności sprawdzamy, że słowo u poprawnie opisuje maszynę, nazwijmy tę maszynę $\mathcal{M}$. Tak więc, należy sprawdzić, że słowo u jest postaci

$$1^{n_A}01^{n_B}0\ 1^{n_Q}0\ t_1\ t_2 \ldots\ t_k,$$

gdzie:

- $n_B \geqslant 4$,
- każde słowo t_i ma postać $[p][a][q][b][d] \in \{0,1\}^*$, gdzie $[p], [q]$ są binarnymi kodami liczb ze zbioru $\{1,\ldots,n_Q\}$ oraz $[a], [b]$ są binarnymi kodami liczb ze zbioru $\{1,\ldots,n_B\}$, i $d \in \{0,1\}$.
- zbiór $\{t_1,\ldots,t_k\}$ opisuje funkcję $\delta\colon Q \times B \to Q \times B \times \{\leftarrow,\rightarrow\}$, gdzie $Q = \{1,\ldots,n_Q\}$ oraz $B = \{1,\ldots,n_B\}$.

Na drugiej taśmie wejściowej podane jest słowo $w \in \{0,1,\#\}^*$. W drugiej kolejności, maszyna $\mathcal{V}$ przepisuje słowo w, zamieniając każdą literę i w słowie w na jej binarną reprezentację długości ℓ_B. Taśma druga będzie służyła do przechowywania reprezentacji taśmy symulowanej maszyny $\mathcal{M}$, w kolejnych krokach jej obliczeń. Pozycja głowicy na taśmie drugiej maszyny $\mathcal{V}$ będzie odpowiadała pozycji głowicy na maszyny $\mathcal{M}$ na jej jedynej taśmie.

Na trzeciej taśmie (roboczej) będzie przechowywany numer aktualnego stanu maszyny $\mathcal{M}$, w postaci zapisu binarnego długości ℓ_B. Początkowo, maszyna $\mathcal{V}$ wpisuje tam reprezentację stanu początkowego 0.

Maszyna $\mathcal{V}$ następnie w kółko wykonuje krok polegający na znalezieniu na pierwszej taśmie odpowiedniej tranzycji, i wykonanie jej. ■

Wniosek 2. *Język* HALT *jest częściowo obliczalny.*

4.3.4 Nierozstrzygalność problemu stopu

Żeby dokończyć dowód Twierdzenia 13, pozostaje pokazać, że język HALT nie jest obliczalny.

Dowód. Przypuśćmy, że język HALT jest obliczalny. Wówczas istnieje maszyna Turinga $\mathcal{H}$ która dla danego słowa wejściowego $u \in \{0,1,\$\}^*$, terminuje i zwraca 1 gdy $u \in$ HALT, oraz terminuje i zwraca 0 gdy $u \notin$ HALT. Skonstruujemy maszynę $\mathcal{S}$ która[10] terminuje na kodzie *kod* maszyny $\mathcal{M}$ wtedy i tylko wtedy, gdy maszyna $\mathcal{M}$ nie terminuje na słowie *kod*. Bardziej formalnie, dla dwóch maszyn $\mathcal{M}, \mathcal{N}$,

[10] Maszyna $\mathcal{S}$ byłaby więc superfryzjerem, który jak wiemy, nie może istnieć.

napiszemy $T(\mathcal{M}, \mathcal{N})$ jeżeli maszyna $\mathcal{M}$ terminuje na (każdym) słowie będącym kodem maszyny $\mathcal{N}$. Skonstruujemy taką maszynę $\mathcal{S}$, że dla każdej maszyny $\mathcal{M}$ zachodzi

$$T(\mathcal{S}, \mathcal{M}) \iff \neg T(\mathcal{M}, \mathcal{M}). \tag{4.1}$$

To jest niemożliwe, bo podstawiając $\mathcal{M} := \mathcal{S}$ dostajemy $T(\mathcal{S}, \mathcal{S}) \iff \neg T(\mathcal{S}, \mathcal{S})$, co jest absurdem.

Maszyna $\mathcal{S}$ działa następująco: dane na wejściu słowo $w \in \{0,1\}^*$, zastępuje słowem $w\$w$, a następnie symuluje działanie maszyny $\mathcal{H}$ na tym słowie, do momentu, gdy ta zakończy. Jeśli $\mathcal{H}$ zwróciła 1, to maszyna $\mathcal{S}$ wchodzi w nieskończoną pętle i się wiesza, w przeciwnym przypadku, jeśli $\mathcal{H}$ zwróciła 0, to maszyna $\mathcal{S}$ terminuje. A zatem, zachodzi równoważność (4.1), co jest niemożliwe. Tak więc, problem HALT nie jest obliczalny. ■

4.3.5 *Wnioski i redukcje*

Wniosek 3. *Dopełnienie języka* HALT *nie jest częściowo obliczalne.*

Dowód: gdyby było, to na mocy Faktu 1 i Lematu 17, język HALT byłby obliczalny.

Niech $\text{HALT}_\varepsilon \subseteq \{0,1\}^*$ będzie językiem składającym się z słów postaci $kod(\mathcal{M})$, gdzie $\mathcal{M}$ jest jednotaśmową maszyną Turinga zatrzymującą się na słowie pustym.

Wniosek 4. *Język* HALT_ε *nie jest obliczalny.*

Dowód. Zobaczymy tutaj metodę *redukcji*, którą poniżej sformalizujemy.

Mając daną maszynę $\mathcal{M}$ oraz słowo wejściowe w, możemy skonstruować maszynę $\mathcal{M}_w$ która działa tak: wymazuje swoje wejście, zastępując je słowem w, na którym następnie symuluje działanie maszyny $\mathcal{M}$. Jasne jest, że maszyna $\mathcal{M}_w$ zatrzymuje się na słowie pustym wtedy i tylko wtedy, gdy maszyna $\mathcal{M}$ zatrzymuje się na słowie w. Ponadto, funkcja $(\mathcal{M}, w) \mapsto \mathcal{M}_w$ jest obliczalna: mając dany kod maszyny $\mathcal{M}$ oraz słowo w, możemy obliczyć kod maszyny $\mathcal{M}_w$.

Dokładniej, można pokazać co następuje:

- dla danego kodu *kod* oraz słowa $w \in A^*$ istnieje taki kod kod_w, że

$$kod\$w \in \text{HALT} \iff kod_w \in \text{HALT}_\varepsilon$$

- Ponadto, funkcja $(kod, w) \mapsto kod_w$ jest obliczalna.

Z tych dwóch faktów wynika nieobliczalność języka HALT_ε: gdyby $\text{HALT}_\varepsilon = L(\mathcal{M})$ dla pewnej maszynę $\mathcal{M}$, to skonstruowalibyśmy maszynę $\mathcal{N}$ dla języka HALT, która dla słów postaci (kod, w) oblicza słowo kod_w i uruchamia na nim maszynę $\mathcal{M}$. Maszyna $\mathcal{N}$ akceptowałaby język HALT, co jest niemożliwe. ■

To można wyrazić precyzyjniej tak: język HALT jest przeciwobrazem języka HALT_ε przy funkcji obliczalnej. Ponieważ zbiory obliczalne są zamknięte na przeciwobrazy przy funkcjach obliczalnych, a HALT nie jest obliczalny, to HALT_ε nie może być obliczalny.

Następujący wniosek, który pozostawiamy jako ćwiczenie, mówi, że problem pustości dla maszyn Turinga jest nierozstrzygalny. Przypomnijmy, że problem pustości jest rozstrzygalny dla automatów skończonych i dla gramatyk bezkontekstowych.

Wniosek 5. *Nie istnieje algorytm*[11]*, który dla danej maszyny Turinga* $\mathcal{M}$ *stwierdza, czy* $L(\mathcal{M}) = \emptyset$.

[11] Przez "algorytm" oczywiście rozumiemy maszynę Turinga $\mathcal{E}$, która na wejściu dostaje kod innej maszyny Turinga $\mathcal{M}$, oraz zawsze terminuje i wypisuje na taśmie 0 lub 1, w zależności od tego, czy $L(\mathcal{M}) = \emptyset$.

Wniosek 6. *Istnieje język nad alfabetem unarnym który jest częściowo obliczalny, ale nie obliczalny.*

Dowód. Słowo $w = a_1 a_2 \dots a_n$ nad alfabetem $\{0,1\}$ traktujemy jako zapisem binarnym pewnej liczby $n_w \in \mathbb{N}$ (dopiszmy do w wiodącą cyfrę 1 żeby uniknąć niejednoznaczności). Funkcja $w \mapsto 1^{n_w}$ z $\{0,1\}^*$ w $\{1\}^*$ jest różnowartościowa i obliczalna. A zatem, język $\{1^{n_w} \mid w \in \text{HALT}\}$ spełnia tezę wniosku[12]. ∎

[12] Język ten jest częściowo obliczalny, jako obraz języka częściowo obliczalnego przy funkcji obliczalnej, a nie jest obliczalny, bo inaczej HALT byłby obliczalny, jako jego przeciwobraz przy funkcji obliczalnej.

Redukcja problemu $L \subseteq A^*$ do problemu $K \subseteq B^*$ jest to taka funkcja $f\colon A^* \to B^*$, że[13] dla każdego słowa $w \in A^*$ zachodzi równoważność

$$w \in L \iff f(w) \in K.$$

[13] zwięźle: $f^{-1}(K) = L$

Redukcja f jest *obliczalna* jeżeli funkcja f jest obliczalna. Będziemy tylko rozważali obliczalne redukcje, więc będziemy pomijali to słowo. Metodę redukcji można ująć następująco:

> Jeżeli istnieje redukcja języka L do języka K oraz język K jest obliczalny, to język L też jest obliczalny.

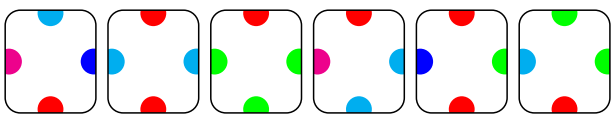
Jest to zatem przydatna metoda dowodzenia, że dany język L jest nieobliczalny. We Wniosku 4 skonstruowaliśmy (obliczalną) redukcję języka HALT do języka HALT_ε, co dowodzi, że HALT_ε nie jest obliczalny, gdyż HALT nie jest obliczalny.

4.3.6 *Inne problemy nierozstrzygalne*

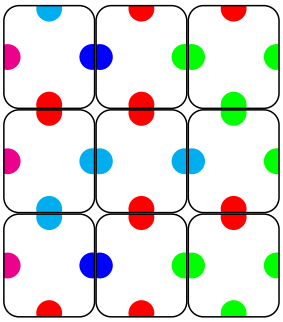
WYKŁAD 10

Język HALT jest punktem wyjścia do pokazywania nierozstrzygalności wielu problemów decyzyjnych. Poniżej jest kilka innych przykładów.

Kafelkowanie nieograniczonego kwadratu. W problemie kafelkowania, dana jest (skończona) kolekcja kafelków, gdzie każdy kafelek jest kwadratem 1×1 o pokolorowanych krawędziach. Przykładowo:



Celem jest wykafelkowanie kwadratu $k \times k$ (gdzie k jest dowolne i nieznane) w taki sposób, że dwa sąsiadujące kwadraty stykają się wzdłuż krawędzi o tym samym kolorze. Kafelków z kolekcji nie można obracać, ale można ich używać wielokrotnie. Przykładowo, używając powyższych kafelków, można następująco wykafelkować kwadrat 3×3:



Dodatkowo, żądamy, żeby w lewym-dolnym rogu wykafelkowanego kwadratu $k \times k$ był pierwszy kafelek z kolekcji, a prawym-górym rogu ma być drugi kafelek z kolekcji. Pytanie brzmi: czy da się w taki sposób wykafelkować pewien kwadrat $k \times k$?

Problem: UNBOUNDED SQUARE TILING
Dane: kolekcja kafelków o pokolorowanych krawędziach
Rozstrzygnąć: czy istnieje takie $k \in \mathbb{N}$ że da się wyłożyć kwadrat $k \times k$ danymi kafelkami, tak by każde dwie przylegające krawędzie miały ten sam kolor, oraz w lewym dolnym rogu kwadratu był pierwszy kafelek z kolekcji, a w prawym górym rogu drugi kafelek kolekcji?

Udowodnimy, że powyższy problem jest nierozstrzygalny. Przypuśćmy bowiem, że jest. Pokażemy, że problem HALT_ε też jest rozstrzygalny. To da sprzeczność, bo problem HALT_ε nie jest rozstrzygalny.

Żeby pokazać, że problem HALT_ε jest rozstrzygalny, rozważmy maszynę Turinga $\mathcal{M}$ daną na wejściu do tego problemu. Należy stwierdzić, czy maszyna $\mathcal{M}$ terminuje na słowie pustym. Innymi słowy, pytamy, czy maszyna $\mathcal{M}$ ma po słowie $\mathcal{M}$ skończony bieg, który kończy w konfiguracji ze stanem końcowym q_{fin}.

Łatwo można zmodyfikować maszynę $\mathcal{M}$ w taki sposób, żeby spełniała następujące warunki:

1. przed przejściem do stanu q_{fin} czyściła zawartość taśmy, wpisując wszędzie $\sqcup$,

2. każdy stan maszyny wyznacza jednoznacznie kierunek następnego ruchu głowicy, i kierunek poprzedniego ruchu głowicy. Dokładniej, zachodzą następujące warunki:

(*a*) istnieje taka funkcja $N\colon Q \to \{\leftarrow, \rightarrow\}$, że dla każdego stanu $q \in Q$ oraz symbolu $b \in B$, trójka $\delta(q, b) \in B \times \{\leftarrow, \rightarrow\} \times Q$ ma na drugiej współrzędnej kierunek $N(q)$

(*b*) istnieje taka funkcja $P\colon Q \to \{\leftarrow, \rightarrow\}$, że dla każdego stanu $p \in Q$ oraz symbolu $b \in B$, jeżeli przez $(c, d, q) \in B \times \{\leftarrow, \rightarrow\} \times Q$ oznaczymy trójkę $\delta(p, b)$, to $P(q) = d$.

Szukany bieg maszyny $\mathcal{M}$ wygląda teraz mniej więcej tak:

Pierwsza konfiguracja jest w dolnym wierszu macierzy, ostatnia konfiguracja jest w górnym wierszu macierzy. jeżeli głowica znajduje się na pozycji i taśmy, w której zapisana jest litera b, a obecny stan maszyny to q, to piszemy $\overset{q}{b}$ na i-tej pozycji konfiguracji.

W stanach q_0, r, q, maszyna pójdzie w prawo, tj. $N(q_0) = N(r) = N(q) = \rightarrow$, a w stanach s, t pójdzie w lewo. Przechodząc do stanu q_{fin}, r, s, maszyna idzie w prawo, tj. $P(q_{fin}) = P(r) = P(s) = \rightarrow$, a przechodząc do stanów q, t, maszyna idzie w lewo.

␣	␣	␣	␣	$\overset{q_{fin}}{␣}$	␣	␣	␣	␣	␣	␣
␣	␣	␣	$\overset{r}{␣}$	␣	␣	␣	␣	␣	␣	␣
␣	␣	$\overset{r}{b}$	␣	␣	␣	␣	␣	␣	␣	␣
␣	$\overset{r}{b}$	b	␣	␣	␣	␣	␣	␣	␣	␣
$\overset{q}{a}$	b	b	␣	␣	␣	␣	␣	␣	␣	␣
a	$\overset{t}{b}$	b	␣	␣	␣	␣	␣	␣	␣	␣
a	b	$\overset{s}{a}$	␣	␣	␣	␣	␣	␣	␣	␣
a	$\overset{q}{b}$	a	␣	␣	␣	␣	␣	␣	␣	␣
a	b	$\overset{s}{␣}$	␣	␣	␣	␣	␣	␣	␣	␣
a	$\overset{p}{␣}$	␣	␣	␣	␣	␣	␣	␣	␣	␣
$\overset{q_0}{␣}$	␣	␣	␣	␣	␣	␣	␣	␣	␣	␣

Na mocy warunku (1), następujące warunki są równoważne:

- Maszyna $\mathcal{M}$ terminuje na słowie pustym,
- Istnieje liczba k oraz etykietowanie pól kwadratu $k \times k$ elementami zbioru $B \cup Q \times B$, które jest zgodne z funkcją przejścia maszyny $\mathcal{M}$, w pierwszym wierszu ma kolejno etykiety

$$\overset{q_0}{␣}\, ␣\, ␣\, \cdots ␣\, ␣,$$

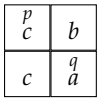
w ostatnim wierszu ma kolejno etykiety

$$␣\, ␣ \cdots ␣\, \overset{q_{fin}}{␣}\, ␣ \cdots ␣\, ␣.$$

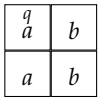
Teraz opiszemy, jak z maszyny $\mathcal{M}$ stworzyć kolekcję kafelków $K_{\mathcal{M}}$. Kolekcja $K_{\mathcal{M}}$ składa się z kafelków, które są fragmentami 2×2 biegów, które są zgodne z funkcją przejścia maszyny $\mathcal{M}$. Konkretnie, dla każdych $a, b \in B$ oraz $q \in Q$, jeżeli $\delta(q, a) = (b, \rightarrow, p)$, to dla każdej litery c mamy w kolekcji kafelek:

b	$\overset{p}{c}$
$\overset{q}{a}$	c

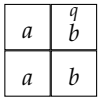
a jeżeli $\delta(q,a) = (b, \leftarrow, p)$, to dla każdej litery c mamy w kolekcji kafelek:



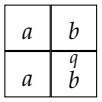
Ponadto, dla każdych $a, b \in B$ oraz $q \in Q$ mamy w kolekcji kafelki:



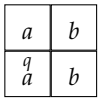
o ile maszyna przechodząc do stanu q idzie w prawo, tj. $P(q) = \rightarrow$,



o ile maszyna przechodząc do stanu q idzie w lewo, tj. $P(q) = \leftarrow$,

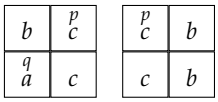


o ile maszyna ze stanu q idzie w prawo, tj. $N(q) = \rightarrow$,

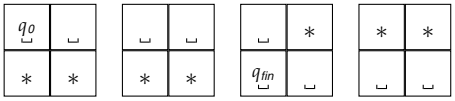


o ile maszyna ze stanu q idzie w lewo, tj. $N(q) = \leftarrow$,

Takie fragmenty traktujemy jako kafelki o czterech krawędziach, gdzie kolorem krawędzi jest ciąg symboli wypisanych wzdłuż tej krawędzi. Tak więc, można obok siebie położyć np. poniższe kafelki:



Dodatkowo, są jeszcze następujące, specjalne kafelki:



Kafelkując kwadrat, wymagamy, by w lewym dolnym rogu znalazł się pierwszy kafelek narysowany powyżej, oraz by prawym górym rogu znalazł się ostatni kafelek narysowany powyżej.

Nietrudno zobaczyć, że teraz następujące warunki są równoważne[14]:

- Maszyna $\mathcal{M}$ terminuje na słowie pustym,
- Istnieje liczba k oraz kafelkowanie kwadratu $k \times k$ kafelkami z kolekcji $K_{\mathcal{M}}$, spełniające powyższe wymagania.

Podsumowując, pokazaliśmy następujący lemat.

[14] Korzystamy tutaj z warunku (2), żeby udowodnić, że w każdym wierszu poprawnego kafelkowania głowica jest "tylko w jednym miejscu".

Lemat 18. *Istnieje taka obliczalna funkcja f, która maszynę Turinga $\mathcal{M}$ przekształca w taką kolekcję kafelków $K_{\mathcal{M}}$, że następujące warunki są równoważne dla każdej liczby $n \geqslant 0$:*

- *Maszyna $\mathcal{M}$ terminuje na słowie pustym w n krokach,*
- *Istnieje kafelkowanie kwadratu $n \times n$ za pomocą kafelków z kolekcji $K_{\mathcal{M}}$, spełniające powyższe wymagania.*

Obliczalność funkcji f wynika stąd, że mając opis maszyny $\mathcal{M}$, kolekcję $\mathcal{M}$ konstruujemy bezpośrednio na podstawie funkcji przejścia maszyny $\mathcal{M}$. Lemat 18 można krócej wyrazić tak: istnieje redukcja problemu HALT_ε do problemu kafelkowania nieograniczonego kwadratu.

Z Lematu 18 wynika, że gdyby problem kafelkowania był rozstrzygalny, to problem HALT_ε też byłby rozstrzygalny: otrzymawszy opis maszyny $\mathcal{M}$, wpierw obliczamy kolekcję kafelków $K_{\mathcal{M}}$ korzystając z obliczalnej funkcji f, a potem, pytamy się, czy kafelkami z tej kolekcji $K_{\mathcal{M}}$ można wykafelkować pewien kwadrat. Tak więc, problem kafelkowania nieograniczonego kwadratu jest nierozstrzygalny.

Kafelkowanie płaszczyzny. Rozważamy inny wariant, w którym wymagamy, by cała płaszczyzna była wykafelkowana.

Problem: TILING
Dane: kolekcja kafelków
Rozstrzygnąć: czy da się wyłożyć całą płaszczyznę kafelkami?

Nierozstrzygalność problemu kafelkowania udowodnił Robert Berger obalając hipotezę swojego promotora, Hao Wanga[15]. Dowód jest skomplikowany i pominiemy go tutaj.

Zauważmy, że problem kafelkowania nieograniczonego kwadratu jest częściowo obliczalny: wystarczy testować wszystkie możliwe kafelkowania kwadratów 1×1, 2×2, 3×3, itd., aż do napotkania poprawnego kafelkowania. Tymczasem *dopełnienie* problemu kafelkowania płaszczyzny jest częściowo obliczalne. Nietrudno pokazać bowiem[16], że istnieje kafelkowanie całej płaszczyzny wtedy, i tylko wtedy, gdy dla każdego $m \geqslant 0$ istnieje kafelkowanie kwadratu $m \times m$. Więc żeby stwierdzić, że całej płaszczyzny nie da się wykafelkować, wystarczy testować wszystkie możliwe kafelkowania kwadratów 1×1, 2×2, 3×3, itd., aż do napotkania kwadratu, którego *nie da się* wykafelkować. To dowodzi częściową obliczalność dopełnienia problemu kafelkowania płaszczyzny. A zatem, problem ten *nie jest* częściowo obliczalny – gdyby był, to na mocy Faktu 1, byłby obliczalny, co jest sprzeczne z twierdzeniem Bergera.

Tak więc, sytuacje w przypadku problemu kafelkowania nieograniczonego kwadratu oraz problemu kafelkowania płaszczyzny są

[15] Wang przypuszczał, że dla każdego zestawu kafelków K, jeżeli istnieje jakieś kafelkowanie płaszczyzny, to istnieje kafelkowanie *okresowe*, tj. takie, które jest niezmiennicze ze względu na pewne przesunięcie płaszczyzny. Nietrudno zobaczyć, że to by implikowało rozstrzygalność problemu. Tak więc, wynik Bergera implikuje istnienie zestawów kafelków K za pomocą których można wykafelkować płaszczyznę, ale tylko w sposób nieokresowy. Oryginalny zestaw kafelków Bergera zawierał 20,426 kafelków, później tę liczbę zmniejszył do 104. Obecnie najmniejszy znany taki zestaw składa się z 11 kafelków i używa czterech kolorów, i tych liczb nie da się już zmniejszyć.

[16] To wynika z Lematu Königa lub z twierdzenia Tychonowa o zwartości, lub z twierdzenia o zwartości dla logiki pierwszego rzędu.

dualne: pierwszy problem jest częściowo rozstrzygalny a jego dopełnienie nie jest, a dla drugiego problemu sytuacja jest odwrotna.

W szczególności, na próżno by szukać redukcji problemu HALT_ε do problemu kafelkowania płaszczyzny – taka redukcja nie istnieje[17]. Żeby udowodnić twierdzenie Bergera, trzeba skonstruować redukcję *dopełnienia* problemu HALT_ε do problemu kafelkowania płaszczyzny. Inaczej mówiąc, dla danej maszyny Turinga $\mathcal{M}$ należy obliczyć taki zestaw kafelków $K_\mathcal{M}$, że $\mathcal{M}$ *nie zatrzymuje się* na słowie pustym wtedy, i tylko wtedy, gdy kafelkami $K_\mathcal{M}$ da się wykafelkować całą płaszczyznę.

[17] Gdyby istniała, to dopełnienie problemu HALT_ε byłoby częściowo obliczalne. Ponieważ problem HALT_ε jest częściowo obliczalny, to na mocy Faktu 1, byłby obliczalny, a nie jest.

WYKŁAD 11

Problem uniwersalności gramatyk (NR179):

Problem: CFL-UNIVERSALITY
Dane: gramatyka $\mathcal{G}$ nad alfabetem A
Rozstrzygnąć: czy $L(\mathcal{G}) = A^*$?

Idea: dla danej maszyny $\mathcal{M}$ konstruujemy taką gramatykę $\mathcal{G}_\mathcal{M}$, że $\mathcal{G}_\mathcal{M}$ akceptuje dokładnie te słowa, które nie są poprawnymi kodowaniami akceptujących biegów maszyny $\mathcal{M}$. (Rysunek).

Problem Posta (ang. *Post Correspondence Problem*, NR178):

Problem: PCP
Dane: zbiór par słów $\{(u_i, v_i) : i = 1, \ldots, k\}$
Rozstrzygnąć: czy istnieje taki ciąg liczb $i_1, \ldots, i_l \in \{1, \ldots, k\}$, że $u_{i_1} u_{i_2} \cdots u_{i_l} = v_{i_1} v_{i_2} \cdots v_{i_l}$?

Idea: dla danej maszyny Turinga $\mathcal{M}$ konstruujemy instancję PCP, której rozwiązania odpowiadają biegom maszyny $\mathcal{M}$.

Busy beaver oraz liczba kroków. *Bóbr* to maszyna Turinga o alfabecie roboczym $\{0, \sqcup\}$, która terminuje na słowie pustym. *Rozmiar* bobra to jego liczba stanów (nie uwzględniając stanu akceptującego), a *wynikiem* bobra jest liczba wystąpień litery o na taśmie w momencie terminacji. Niech $W(n)$ będzie największym wynikiem bobra o n stanach, i niech $K(n)$ będzie największą liczbą kroków wykonanych przez bobra o n stanach. Pokażemy, że funkcje $W(n)$ oraz $K(n)$ nie są obliczalne. Inaczej mówiąc, następujące problemy *obliczeniowe* są nieobliczalne.

Problem: BUSY BEAVER SCORE
Dane: liczba n
Obliczyć: $W(n)$ – największy wynik bobra rozmiaru n

Problem: BUSY BEAVER STEPS
Dane: liczba n
Obliczyć: $K(n)$ – największa liczba kroków wykonanych przez bobra rozmiaru n

Pokażemy, że nie istnieje maszyna Turinga obliczająca funkcję $K\colon \mathbb{N} \to \mathbb{N}$. Przypuśćmy bowiem, że taka maszyna $\mathcal{K}$ istnieje. Pokażemy, że wtedy umielibyśmy rozstrzygnąć problem terminacji na słowie pustym dla maszyn Turinga o alfabecie roboczym $\{0, \sqcup\}$. Ten problem jest nierozstrzygalny[18], co daje sprzeczność.

[18] Każdą maszynę Turinga bez wejścia można zamienić na równoważną, która używa alfabetu roboczego $\{0, \sqcup\}$.

Dla danej maszyny $\mathcal{M}$ o alfabecie roboczym $\{0, \sqcup\}$ i n stanach, żeby stwierdzić, czy $\mathcal{M}$ terminuje na słowie pustym, postępujemy następująco. Uruchamiamy maszynę $\mathcal{M}$ na $K(n)$ kroków (gdzie $f(n)$ obliczamy używając maszyny $\mathcal{K}$) na słowie ε. Jeżeli w tym czasie, maszyna $\mathcal{M}$ się nie zatrzyma, to wiemy, że $\mathcal{M}$ się nigdy nie zatrzyma, więc wypisujemy "NIE". Gdy się zatrzyma, to wypisujemy "TAK". To by dało algorytm rozstrzygający, czy dana maszyna $\mathcal{M}$ o alfabecie roboczym $\{0, 1, \sqcup\}$ terminuje na słowie pustym, a taki algorytm nie istnieje.

Nietrudno zobaczyć, że każda funkcja obliczalna $f\colon \mathbb{N} \to \mathbb{N}$ jest od pewnego momentu mniejsza, niż funkcja K zdefiniowana powyżej[19].

[19] Tj. $\exists n.\forall m \geqslant n.g(m) \leqslant W(m)$.

Nieobliczalność funkcji W można wywnioskować z nieobliczalności funkcji K. Pominiemy argument tutaj.

Wartości $W(n)$ można są znane dla bardzo małych liczb n, np. dla $n = 0, 1, 2, 3$.

> The current 5-state busy beaver champion produces 4098 1s, using 47176870 steps (discovered by Heiner Marxen and Jürgen Buntrock in 1989), but there remain 28 machines with non-regular behavior which are believed to never halt, but which have not yet been proven to run infinitely. At the moment the record 6-state champion produces over 3.51510^{18267} 1s, using over 7.41210^{36534} steps (found by Pavel Kropitz in 2010). As noted above, these are 2-symbol Turing machines.
>
> A simple extension of the 6-state machine leads to a 7-state machine which will write more than $10^{10^{10^{10^{18705353}}}}$ 1s to the tape, but there are undoubtedly much busier 7-state machines[20].

[20] Cytat z Wikipedii, https://en.wikipedia.org/wiki/Busy_beaver

Podobnie jak funkcja K, można pokazać, że dla dowolnej obliczalnej funkcji $f\colon \mathbb{N} \to \mathbb{N}$, funkcja W jest większa niż f od pewnego momentu[21]. Wiadomo, że dla $n = 1919$, dokładna wartość $W(n)$ jest niezależna od aksjomatów teorii zbiorów[22]. Więcej o tym w Rozdziale 4.4.

[21] zauważmy, że $W \leqslant K$, co jest innym sposobem pokazania, że funkcja K jest nieobliczalna

[22] Z grubsza to znaczy, że wartość liczby $W(1919)$ nie jest możliwa do ustalenia, korzystając jedynie z aksjomatów ZFC. Dokładniej, istnieje taka (bardzo duża) liczba $m \in \mathbb{N}$, że obydwa stwierdzenia $W(n) > m$ oraz $W(n) \leqslant m$ są niesprzeczne z aksjomatami ZFC.

Matrix mortality problem. Mając dany zbiór X sześciu macierzy 3×3 stwierdzić, czy istnieje taki ciąg $M_1, M_2, \ldots, M_n \in X$ (być może z powtórzeniami), że iloczyn $M_1 \cdots M_n$ jest macierzą zerową.

Teoria ZFC. Rozważamy zdania logiki pierwszego rzędu w języku teorii zbiorów, tj. zdania zbudowane z logicznych symboli

$\forall_x, \exists_x, \wedge, \vee, \neg, \to, \leftrightarrow$ oraz symboli $\in, =$. Przykładowe zdanie to:

$$\forall_x \big(\underbrace{(\neg \exists_z z \in x)}_{x \text{ jest zbiorem pustym}} \to \forall_y \underbrace{(\forall_z z \in x \to z \in y)}_{x \subseteq y} \big). \tag{4.2}$$

Powyższe zdanie mówi, że każdy zbiór x, który jest pusty, jest zawarty w każdym innym zbiorze y.

Aksjomaty ZFC to pewien zbiór takich zdań, które powszechnie uważa się, za aksjomaty matematyki. Przykładowe aksjomaty mówią, że dwa zbiory są sobie równe wtedy, i tylko wtedy, gdy mają te same elementy[23], lub że istnieje zbiór pusty[24], lub że dla każdego zbioru istnieje zbiór wszystkich jego podzbiorów[25]. Są też bardziej skomplikowane aksjomaty, jak aksjomat indukcji (istnieje taki zbiór N, że $\emptyset \in N$, oraz dla każdego x, jeśli $x \in N$ to $x \cup \{x\} \in N$).

Jest pewien naturalny system dowodzenia, który pozwala dowodzić bardziej skomplikowanych zdań z prostszych zdań oraz aksjomatów. System składa się z pewnych reguł dowodzenia, typu: jeśli potrafimy udowodnić φ oraz ψ, to potrafimy udowodnić $\varphi \wedge \psi$. Podobnie, jeżeli potrafimy udowodnić, zdanie $\forall_x \varphi(x)$ to potrafimy też udowodnić zdanie φ w którym za zmienną x podstawiony jest dowolna stała. Przykładowo, zdanie (4.2) można udowodnić z aksjomatów ZFC. *Twierdzenia* to są te zdania, które mają dowody.

Rozważamy następujący problem:

Problem: TEORIA ZFC
Dane: zdanie φ
Rozstrzygnąć: czy zdanie φ można udowodnić z aksjomatów ZFC?

Powyższy problem jest nierozstrzygalny. Dowód tego twierdzenia podany jest w Rozdziale 4.4. Zauważmy, że powyższy problem jest półrozstrzygalny, tj. istnieje algorytm, który odpowiada 'TAK' dla wszystkich zdań φ które mają dowód, a dla pozostałych się wiesza[26]. Tak więc, nie istnieje algorytm, który odpowiada 'NIE' dla wszystkich zdań φ które nie mają dowodu, a dla pozostałych się wiesza. W szczególności, istnieją takie zdania φ, że zarówno φ jak i $\neg\varphi$ nie mają dowodu. Takie zdania nazywają się zdaniami *niezależnymi od aksjomatów ZFC*.[27]

Teoria pierwszego rzędu $(\mathbb{N}, +, \times)$. Mając dane zdanie logiki pierwszego rzędu w języku arytmetyki, tj. zdanie zbudowane z logicznych symboli $\forall_x, \exists_x, \wedge, \vee, \neg$ oraz symboli $+, \times$, stwierdzić, czy zachodzi ono w zbiorze liczb naturalnych. Ten problem jest nierozstrzygalny, co udowodnił Kurt Gödel.

Co ciekawe, jeżeli zamiast liczb naturalnych $\mathbb{N}$ rozważymy zbiór liczb rzeczywistych, to problem staje się rozstrzygalny, co udowodnił Alfred Tarski[28]. Z kolei, jeżeli zostaniemy przy liczbach naturalnych,

[23] $\forall_x \forall_y (x = y) \leftrightarrow (\forall_z (z \in x) \leftrightarrow (z \in y))$
[24] $\exists_x \neg \exists_z z \in x$
[25] $\forall_x \exists_y \forall_z (z \in y \leftrightarrow \forall_t (t \in z \to t \in x))$

Jest jeszcze nieskończony *schemat aksjomatów*: dla każdej formuły $\varphi(x)$ logiki pierwszego rzędu, jest aksjomat mówiący, że jeśli istnieje zbiór z, to zbiór $\{x \mid x \in z, \varphi(x)\}$ też istnieje. Formalnie: $\forall_z \exists_u \forall_x (x \in u \leftrightarrow x \in z \wedge \varphi(x))$.

[26] ten algorytm przeszukuje wszystkie możliwe dowody

[27] Przykładowe takie zdanie to *hipoteza continuum*: nie istnieje zbiór mocy pomiędzy $\aleph_0$ i $\mathfrak{c}$, tzn. każdy nieprzeliczalny podzbiór $P(\mathbb{N})$ jest równoliczny z $P(\mathbb{N})$.

[28] A zatem, można napisać program komputerowy, który teoretycznie potrafi rozwiązywać wszystkie zadania z geometrii płaskiej – takie zadania to są zdania mówiące o elementach $\mathbb{R}^2$, prostych i okręgach. Jednak nie istnieje program, który rozwiązuje wszystkie zadania z teorii liczb.

ale zabronimy używania symbolu $\times$, to problem też jest rozstrzygalny, co udowodnił Mojżesz Presburger, student Alfreda Tarskiego. Jeden z możliwych dowodów tego twierdzenia używa automatów i nie jest skomplikowany.

10 problem Hilberta – rozwiązywanie równań diofantycznych. Układ równań diofantycznych to układ równań postaci

$$\begin{aligned} p_1(x_1, \ldots, x_k) &= 0, \\ p_2(x_1, \ldots, x_k) &= 0, \\ &\ldots \\ p_r(x_1, \ldots, x_k) &= 0, \end{aligned}$$

gdzie $p_1, \ldots, p_r$ są wielomianami o k zmiennych i o współczynnikach całkowitych, gdzie szukamy rozwiązań w dziedzinie liczb całkowitych, tj. szukamy k liczb całkowitych $a_1, \ldots, a_k \in \mathbb{Z}$, że $p_i(a_1, \ldots, a_k) = 0$ dla $i = 1, \ldots, r$. Przykładowy taki układ to:

$$\begin{aligned} x^{15} + y^{15} - z^{15} &= 0, \\ x_1^2 + x_2^2 + x_3^2 + x_4^2 + 1 - x &= 0, \\ y_1^2 + y_2^2 + y_3^2 + y_4^2 + 1 - y &= 0, \\ z_1^2 + z_2^2 + z_3^2 + z_4^2 + 1 - z &= 0. \end{aligned}$$

Spełnialność drugiego równania, na mocy twierdzenia Lagrange'a o rozkładach liczb naturalnych, jest równoważna nierówności $x \geqslant 1$, podobnie trzecie i czwarte są równoważne $y \geqslant 1$ oraz $z \geqslant 1$, odpowiednio. Pierwsze równanie jest niespełnialne w dodatnich liczbach całkowitych, na mocy Wielkiego Twierdzenia Fermata, udowodnionego przez A. Wilesa.

który, jak wynika z Wielkiego Twierdzenia Fermata, nie ma rozwiązań w liczbach całkowitych.

W roku 1910 Hilbert postawił pytanie, czy istnieje algorytm stwierdzający, czy dany układ równań diofantycznych ma rozwiązanie całkowite. W roku 1970 udowodniono, że ten problem jest nierozstrzygalny, autorami tego wyniku są Martin Davis, Jurij Matiyasevich, Hilary Putnam oraz Julia Robinson.

Dowód tego twierdzenia jest bardzo trudny. Ten wynik wzmacnia wynik o nierozstrzygalności teorii $(\mathbb{N}, +, \times)$, gdyż istnienie rozwiązania układu równań wielomianowych można zapisać jako zdanie w języku arytmetyki[29]. Jeżeli natomiast dozwolimy rozwiązania rzeczywiste wielomianów (tj. $a_1, \ldots, a_k \in \mathbb{R}$), to problem jest rozstrzygalny, co wynika natychmiast z wyniku Tarskiego o rozstrzygalności teorii $(\mathbb{R}, +, \times)$.

[29] Np. $\exists_x \exists_y (x \cdot y + y \cdot y + y \cdot y = y) \wedge (x \cdot x = y)$.

Twierdzenie Rice'a. Niech $\mathcal{L}$ będzie zbiorem języków. Zbiór $\mathcal{L}$ nazywamy *własnością* języków, i powiemy, że język L ma własność $\mathcal{L}$ jeżeli $L \in \mathcal{L}$. Przykładowe własności to "niepustość", odpowiadające zbiorowi języków niepustych[30], bądź "regularność", odpowiadające zbiorowi języków regularnych. Własność $\mathcal{L}$ jest *trywialna* jeśli $\mathcal{L}$ nie zawiera żadnych języków częściowo obliczalnych, albo zawiera wszystkie języki częściowo obliczalne.

[30] Formalnie, żeby to był zbiór, ograniczmy się do alfabetu $\{0, 1\}$.

Niech $\mathcal{L}$ będzie własnością języków, oraz rozważmy następujący problem: dla danej maszyny Turinga $\mathcal{M}$ stwierdzić, czy $L(\mathcal{M})$ ma własność $\mathcal{L}$. Oczywiście, jeżeli $\mathcal{L}$ jest trywialną własnością języków, to powyższy problem jest rozstrzygalny[31]. Twierdzenie Rice'a mówi, że powyższy problem jest nierozstrzygalny, o ile $\mathcal{L}$ jest nietrywialną własnością języków.

Przykładowo, nie istnieje algorytm rozstrzygający, czy dla danej maszyny Turinga $\mathcal{M}$, język $L(\mathcal{M})$ jest regularny.

[31] rozwiązuje go maszyna Turinga która zawsze odpowiada TAK, bądź maszyna Turinga, która zawsze odpowiada NIE.

Ćwiczenie. Udowodnić twierdzenie Rice'a.

4.4 *Twierdzenie Gödla*

Naszkicujemy teraz, jak z twierdzenia Turinga wynika twierdzenie Gödla o niezupełności. W uproszczeniu, twierdzenie to mówi, że istnieje stwierdzenie matematyczne którego ani się nie da dowieść, ani którego się nie da obalić, korzystając z aksjomatów ZFC. Takie stwierdzenie nazywa się stwierdzeniem *niezależnym* od aksjomatów ZFC.

Wywnioskujemy twierdzenie Gödla z twierdzenia Turinga. Przez *zdanie* rozumiemy dobrze uformowaną formułę matematyczną logiki pierwszego rzędu w języku teorii zbiorów[32]. O *dowodach* zakładamy jedynie tyle, że zbiór poprawnych dowodów jest językiem obliczalnym, tzn. istnieje maszyna Turinga (lub program), który stwierdza, czy dane słowo w opisuje poprawny dowód danego zdania z, korzystający z aksjomatów ZFC. Z tego wynika, że zbiór zdań dowodliwych jest rekurencyjnie przeliczalny[33].

[32] Tak więc, dopuszczalne są symbole logiczne $\forall, \exists, \wedge, \vee, \neg$ oraz symbole $\subseteq, \in, \cup, \cap, \emptyset$.

[33] Maszyna enumerująca zdania dowodliwe przeszukuje wszystkie pary (w, z) o coraz większej długości, i sprawdza, czy w jest poprawnym dowodem zdania z; jeśli jest, to wypisuje na taśmie wyjściowej zdanie z.

Twierdzenie 14 (Pierwsze Twierdzenie Gödla o niezupełności)**.** *Niech $\mathcal{P}$ będzie taką maszyną Turinga, że język $L(\mathcal{P}) \subseteq A^*$ nie jest obliczalny. Wówczas, o ile aksjomaty są ZFC są niesprzeczne, to istnieje takie słowo $w \in A^*$, że maszyna $\mathcal{P}$ nie terminuje na słowie w, ale zdanie "maszyna $\mathcal{P}$ nie terminuje na słowie w" nie jest dowodliwe z aksjomatów ZFC.*

Dowód. Niech $L = L(\mathcal{P})$. Przypuśćmy, że każde zdanie postaci "maszyna $\mathcal{P}$ nie terminuje na słowie w", gdzie $w \in A^* - L$, posiada dowód. Pokażemy, że wtedy język L jest obliczalny, co jest sprzeczne z założeniem.

Skonstruujemy maszynę Turinga $\mathcal{Q}$, działającą tak: wpierw, $\mathcal{Q}$ wczytuje słowo w i konstruuje zdanie $\mathit{hangs}(w)$ które jest matematyczną formalizacją[34] zdania "maszyna $\mathcal{P}$ wiesza się na wejściu w". Następnie, maszyna $\mathcal{Q}$ jednocześnie uruchamia dwie półprocedury: pierwsza uruchamia maszynę $\mathcal{P}$ na słowie w, a druga szuka dowodu dla zdania $\mathit{hangs}(w)$.

Jeżeli pierwsza półprocedura nigdy nie kończy, to $w \notin L$, a więc na mocy naszego założenia, zdanie $\mathit{hangs}(w)$ jest dowodliwe, czyli

[34] To zdanie otrzymujemy przez rozwinięcie definicji, aż otrzymamy stwierdzenie dotyczące zbiorów, korzystając z faktu, że wszystko w matematyce jest zbiorem. A więc: nie istnieje bieg kończący maszyny $\mathcal{P}$ po słowie w, gdzie bieg kończący to jest pewien skończony ciąg konfiguracji maszyny $\mathcal{P}$, zgodny z funkcją przejścia maszyny $\mathcal{P}$ i słowem w, konfiguracja to też rodzaj skończonego ciągu, zaś *ciąg skończony* jest to funkcja z pewnego zbioru X w zbiór Y, gdzie X jest skończonym początkowym podzbiorem liczb naturalnych (to się też da wyrazić w teorii zbiorów), a funkcja $f\colon X \to Y$ jest to zbiór par spełniający pewne oczywiste warunki, gdzie *para* $(x, y) \in X \times Y$ jest formalnie zbiorem postaci $\{x, \{x, y\}\}$. I tak dalej.

druga półprocedura musi się zakończyć. Z kolei, jeśli druga półprocedura terminuje, to zdanie $hangs(w)$ jest dowodliwe z aksjomatów ZFC. Na mocy niesprzeczności aksjomatów ZFC, zdanie to jest prawdziwe, więc maszyna $\mathcal{P}$ wiesza się na słowie w.

Tak więc, dokładnie jedna z powyższych półprocedur kończy w skończonym czasie. Jeśli pierwsza półprocedura kończy, to maszyna $\mathcal{Q}$ kończy bieg z wynikiem 1, a gdy druga półprocedura kończy, to maszyna $\mathcal{Q}$ kończy bieg z wynikiem 0. Łatwo widać, że, maszyna $\mathcal{Q}$, otrzymawszy na wejściu słowo $w \in A^*$ zawsze kończy bieg i zwraca 1 wtedy i tylko wtedy, gdy $\mathcal{P}$ kończy bieg na słowie w. Tak więc, język L jest obliczalny, co jest sprzeczne z naszym założeniem. ■

A zatem, istnieje zdanie z które nie jest dowodliwe z aksjomatów ZFC, i którego negacja $\neg z$ też nie jest dowodliwa. Co więcej, istnieją takie zdania następujących postaci:

- Maszyna $\mathcal{P}$ kończy na wejściu ε, dla pewnej maszyny Turinga $\mathcal{P}$,
- Maszyna $\mathcal{U}$ kończy na wejściu w, dla pewnego słowa w, gdzie $\mathcal{U}$ jest maszyną z Lematu 17,
- (Przykład z pcp)

Jeżeli φ jest zdaniem oraz A jest strukturą matematyczną, to można zdefiniować[35] co to oznacza, że zdanie φ *zachodzi* w strukturze A. Przykładowo, zdanie $\forall_x(x = 0) \vee (\exists_y x \cdot y = 1)$ zachodzi w każdym ciele.

[35] Ogólniej, definiuje się to dla formuł ze zmiennymi wolnymi, np. $\varphi(x, y)$. Definicja przebiega przez indukcję po budowie formuły φ. Przykładowo, formuła $\exists_y \varphi(x, y)$ zachodzi w strukturze A, dla podstawienia $x \mapsto a$, gdzie $a \in A$, wtedy i tylko wtedy, gdy istnieje takie $b \in A$, że formuła $\varphi(x, y)$ zachodzi w strukturze A, przy podstawieniu $x \mapsto a, y \mapsto b$.

Jeżeli T jest zbiorem zdań, to *modelem* zbioru T jest taka struktura A, która spełnia każde zdanie $\varphi \in T$. Przykładowo, jeżeli T to jest zbiór aksjomatów ciał (mówiące np. o przemienności mnożenia, rozdzielności dodawania względem mnożenia, itd.), to modelami zbioru T są dokładnie wszystkie ciała. Jeżeli T to jest zbiór aksjomatów grup, to modelami zbioru T są dokładnie wszystkie grupy. Jeżeli zaś T to jest zbiór aksjomatów ZFC, to modele zbioru T to są takie struktury A, które "wyglądają" jak uniwersum matematyczne: elementy struktury A nazywamy "zbiorami", możemy stwierdzić, czy jeden zbiór jest zawarty w drugim, istnieje zbiór pusty, relacja zawierania jest przechodnia, itd.

Zbiór zdań T jest *niesprzeczny* jeżeli ma on jakiś model. Każdy niesprzeczny zbiór zdań T, który posiada jakiś model nieskończony, automatycznie posiada wiele bardzo różnych modeli. W szczególności, taki zbiór zdań posiada modele o dowolnej mocy nieskończonej (o tym mówi twierdzenie Lowenheima-Skolema). Na przykład, istnieją ciała dowolnej mocy nieskończonej.

A zatem, dla każdego zbioru zdań T, który ma modele nieskończone, istnieją modele zbioru T o różnych mocach. Często (ale nie

zawsze) istnieją też modele zbioru T które spełniają różne zdania logiki pierwszego rzędu. Przykładowo, równość $1+1=0$ zachodzi w niektórych, ale nie we wszystkich modelach aksjomatów ciał. Inaczej mówiąc, aksjomaty teorii ciał nie są *zupełne*: zbiór zdań T jest *zupełny* jeżeli każde dwa modele zbioru T spełniają dokładnie te same zdania. Przykład zupełnego zbioru zdań to zbiór aksjomatyzujący gęste porządki liniowe bez końców[36]. Twierdzenie Gödla o *pełności* mówi, że zdanie φ zachodzi we wszystkich modelach zbioru T wtedy i tylko wtedy, gdy zdanie φ jest dowodliwe z T, tj. istnieje dowód zdania φ, używający zdań ze zbioru T jako aksjomatów.

[36] zdania te mówią: relacja $\leqslant$ jest przechodnia, zwrotna, antysymetryczna, oraz dla każdych x, y takich, że $x \leqslant y$ oraz $\neg y \neq x$, istnieje taki z, że $x \leqslant z \leqslant y$ oraz $\neg z \leqslant x$ oraz $\neq y \leqslant z$. Wreszcie, nie istnieje element największy i najmniejszy.

Z udowodnionego powyżej twierdzenia o *niezupełności* wynika, że o ile aksjomaty ZFC są niesprzeczne, to istnieje takie zdanie φ, że φ oraz $\neg\varphi$ nie są dowodliwe z aksjomatów ZFC. Z Twierdzenie Gödla o pełności wynika, że zdanie φ zachodzi w niektórych, ale nie wszystkich modelach aksjomatów ZFC. Tak więc, aksjomaty ZFC nie są zupełne – podobnie jak aksjomaty ciał. Można zatem dopisać dodatkowe aksjomaty do aksjomatów ZFC, otrzymując niesprzeczny zbiór aksjomatów (przy założeniu,że ZFC jest niesprzeczny) który jest istotnie silniejszy od ZFC (tzn. dopisane aksjomaty nie są dowodliwe w ZFC). Jednak ten sam dowód w Twierdzeniu 14 zastosowany do tego zbioru aksjomatów znowu pokazuje, że i dla nich istnieją zdania niedowodliwe – jedyne, co w tym dowodzie zakładamy, to że zbiór aksjomatów jest niesprzeczny i rekurencyjnie przeliczalny, oraz że jest wystarczająco bogaty, by w nim opisywać biegi maszyn Turinga.

We wszystkich wynikach w tym rozdziale, aksjomaty ZFC mogą być zastąpione innym zestawem aksjomatów, spełniającym pewne wymagania. Np. nie potrzebujemy aksjomatu wyboru, więc wystarczyłyby aksjomaty ZF, a nawet potrzebujemy zbiorów nieskończonych. To znaczy, że te same wyniki zachodzą dla aksjomatów ZF+$\neg\infty$ (ZF z negacją aksjomatu nieskończoności mówiącego, że istnieje zbiór $\mathbb{N}$). Nieco dokładniej, wszystkie wyniki w tym rozdziale dotyczą dowolnych częściowo obliczalnych systemów aksjomatycznych które są w stanie "symulować" ZF+$\neg\infty$. Aksjomaty Peano mają tę własność.

Powyżej pokazaliśmy, że można znaleźć zdania niezależne od ZFC w jednej z kilku ogólnych postaci. Gödel wskazał jedno *konkretne* zdanie, które jest niezalezne od aksjomatów ZFC: jest to zdanie "aksjomaty ZFC są niesprzeczne". Poniżej pokażemy jego niezależność od aksjomatów ZFC. Innym przykładem takiego zdania jest słynna *hipoteza continuum*[37] – że każdy nieskończony zbiór liczb rzeczywistych jest albo równoliczny z $\mathbb{N}$, albo z $\mathbb{R}$. Tych twierdzeń nie da się udowodnić ani obalić używając jedynie aksjomatów ZFC, ale da się, używając potężniejszych aksjomatów.

[37] Niezależność hipotezy continuum od aksjomatów ZFC udowodnił Paul Cohen w roku 1963. On też pokazał niezależność aksjomatu wyboru od aksjomatów ZF.

Twierdzenie 15 (Drugie Twierdzenie Gödla o niezupełności)**.** *O ile aksjomaty ZFC są niesprzeczne, to zdanie "aksjomaty ZFC są niesprzeczne" jest niezależne od aksjomatów ZFC.*

Dowód. Niech $\mathcal{P}$ będzie taką maszyną Turinga, że język $L = L(\mathcal{P}) \subseteq A^*$ nie jest obliczalny.

Zauważmy, że dowód Twierdzenia 14 odbywa się w ramach aksjomatów ZFC. Gdyby istniał dowód zdania "aksjomaty ZFC są niesprzeczne", to tezę Twierdzenia 14 można by uprościć do następującego twierdzenia, które byłoby dowodliwe w ZFC:

Istnieje takie słowo $w \in A^*$, że maszyna $\mathcal{P}$ nie terminuje na słowie w, ale zdanie "maszyna $\mathcal{P}$ nie terminuje na słowie w" nie jest dowodliwe z aksjomatów ZFC.

Na mocy twierdzenia Gödla o pełności, jeśli zdanie "maszyna $\mathcal{P}$ nie terminuje na słowie w" nie jest dowodliwe, to istnieje model aksjomatów ZFC, w którym to zdanie nie zachodzi, tzn. maszyna $\mathcal{P}$ terminuje na słowie w, w pewnym modelu aksjomatów ZFC. To jest sprzeczne z faktem, że maszyna $\mathcal{P}$ nie terminuje na słowie w.

Tak więc, zdanie "aksjomaty ZFC są niesprzeczne" nie może być dowodliwe, więc jest niezależne od aksjomatów ZFC. ■

Wniosek 7. *Można napisać* konkretny [38] *while-program lub maszynę Turinga, których terminacja (na pustym wejściu) jest niezależna od aksjomatów ZFC.*

[38] To stwierdzenie ma przekazać nie tylko, że taka maszyna $\mathcal{M}$ *istnieje* (to już wynika z Twierdzenia 14 oraz Wniosku 4), ale że czytelnik byłby w stanie napisać ten program lub maszynę na kilku kartkach papieru.

Dowód. Ten program lub maszyna szukają dowodu sprzeczności w aksjomatach ZFC, enumerując wszystkie poprawne dowody, aż natrafią na dowód zdania[39] $\emptyset \neq \emptyset$. Rozbudowując nieco składnię while-programów o struktury danych reprezentujące drzewa, nietrudno napisać while-program przeszukujący wszystkie poprawne dowody z aksjomatów ZFC. Uogólniając odpowiednio Twierdzenie 11, otrzymujemy maszynę $\mathcal{M}$. Nawet zostało to zrobione wprost (odnośnik). ■

[39] równie dobrze, mogłaby szukać dowodu hipotezy continuum, korzystając z twierdzenia Cohena o niezależności tej hipotezy od ZFC

Wniosek 8. *Można skonstruować:*

- *instancję PCP,*
- *gramatykę bezkontekstową,*
- *układ równań diofantycznych,*

których odpowiednio, rozwiązywalność, nieuniwersalność, i rozwiązywalność, są równoważne sprzeczności aksjomatów ZFC, lub hipotezie continuum.

Dowód. Wskazaliśmy konkretne redukcje z problemu HALT do problemów PCP oraz ¬CFL-UNIVERSALITY, a Matiyasevich skonstruował redukcję do problemu rozwiązywalności układów równań diofantycznych. Stosując te redukcje do maszyny $\mathcal{M}$ z poprzedniego wniosku oraz słowa pustego, otrzymamy konkretne instancje tych problemów.[40] ■

[40] Znany jest układ równań diofantycznych kodujący hipotezę continuum.

4.5 Niedeterminizm

WYKŁAD 12

4.5.1 Maszyny niedeterministyczne

Maszyny niedeterministyczne są zdefiniowane tak samo jak deterministyczne, z tym, że zamiast funkcji przejścia mają relację przejścia

$$\delta \subseteq Q \times B \times B \times \{\leftarrow, \rightarrow\} \times Q.$$

W grafie konfiguracji zatem, każda konfiguracja może mieć wiele następników. Biegiem po słowie w jest skończony ciąg konfiguracji, gdzie pierwsza konfiguracja jest początkowa, ostatnia jest akceptująca, a każde dwie kolejne są zgodne z relacją przejścia. Maszyna niedeterministyczna $\mathcal{M}$ *akceptuje* słowo w jeżeli *istnieje* jej bieg akceptujący po słowie w. Przez $L(\mathcal{M})$ oznaczamy zbiór tych słów, które są akceptowane przez maszynę $\mathcal{M}$.

Powyżej zdefiniowaliśmy jednotaśmowe maszyny niedeterministyczne, lecz ta definicja podnosi się w oczywisty sposób do definicji wielotaśmowych maszyn niedeterministycznych, w których relacja przejścia jest postaci

$$\delta \subseteq Q \times B^k \times (B \times \{\leftarrow, \rightarrow\})^k \times Q,$$

gdzie k oznacza liczbę taśm maszyny.

Można by też zdefiniować *relację* $R \subseteq A^* \times B^*$ definiowaną przez maszynę $\mathcal{M}$, składającą się z tych par (w, v), że maszyna $\mathcal{M}$ ma po słowie pewien bieg akceptujący, kończący w konfiguracji w której na taśmie zapisane jest słowo v. Uogólnia to definicję *funkcji* obliczanej przez maszynę deterministyczną $\mathcal{M}$. Możemy powiedzieć, że maszyna niedeterministyczna $\mathcal{M}$ oblicza funkcję częściową jeżeli obliczona przez nią relacja R jest funkcją częściową (tzn. dla każdych w, v, v', jeżeli $(w, v), (w, v') \in R$, to $v = v'$).

Przykład 44. Rozważmy wielotaśmową maszynę niedeterministyczną, która dostawszy graf G (zapisany w postaci macierzy incydencji), maszyna szuka ścieżki w grafie G z wierzchołka 0 do wierzchołka 1. Maszyna działa następująco: zaczyna w wierzchołku 0 oraz w każdym kroku, niedeterministycznie przechodzi do sąsiada obecnego wierzchołka, oraz akceptuje, gdy znajdzie się w wierzchołku 1. Dokładniej, maszyna ma dwie taśmy robocze. Na pierwszej taśmie, przechowywany jest numer i aktualnego wierzchołka. Druga taśma przechowuje licznik j. W każdej fazie, maszyna wykonuje następujące kroki.

1. Ustaw licznik i na 0 (wierzchołek startowy).
2. Ustaw głowicę na początku i-tego wiersza macierzy incydencji grafu G. Ustaw licznik j na 0.
3. Niedeterministycznie wykonaj jedną z dwóch instrukcji:
   - jeśli pod głowicą jest cyfra 1 to przejdź do kroku 4 (decydujemy się przejść do sąsiada j), jeśli nie, to przejdź do stanu odrzucającego.
   - przesuń głowicę w prawo o jeden i zwiększ licznik j o jeden, i przejdź do kroku 5

4. ustaw licznik i na j (w efekcie, aktualny wierzchołek to j).

5. jeśli $i = 1$ akceptuj, jeśli nie, to przejdź do kroku 2.

Maszyna ta ma bieg akceptujący dla danego grafu G wtedy, i tylko wtedy, gdy w grafie G istnieje ścieżka z wierzchołka 0 do wierzchołka 1. ⌟

Nieformalnie, nastepujące twierdzenie mówi, że maszyny Turinga się determinizują, przynajmniej jeżeli chodzi o klasę języków, które są w stanie zdefiniować[41].

[41] W podobny sposób można pokazać, że dla każdej funkcji częściowej f, jeśli istnieje maszyna niedeterministyczna ją obliczająca, to też istnieje maszyna deterministyczna ją obliczająca.

Twierdzenie 16. *Następujące warunki są równoważne dla języka $L \subseteq A^*$:*

1. $L = L(\mathcal{M})$ *dla pewnej niedeterministycznej maszyny Turinga* $\mathcal{M}$,

2. $L = L(\mathcal{M})$ *dla pewnej deterministycznej maszyny Turinga* $\mathcal{M}$.

Dowód. Naszkicujemy teraz implikację 1→2 – druga implikacja jest oczywista. Niech $\mathcal{M}$ będzie maszyną Turinga. Skonstruujemy taką deterministyczną maszynę $\mathcal{K}$, że $L(\mathcal{M}) = L(\mathcal{K})$.

Idea jest następująca: maszyna $\mathcal{K}$ symuluje maszynę $\mathcal{M}$, lecz zamiast dokonywać niedeterministycznie wyboru kolejnej tranzycji, rozdziela się na kilka kopii, w których równolegle testuje wszystkie możliwości.

Nieco dokładniej, maszyna $\mathcal{M}$ działa tak. Dostawszy słowo w, zapisuje na swojej taśmie początkową konfigurację c_0 maszyny $\mathcal{M}$ na słowie w. W każdym momencie swojego biegu, będzie na taśmie miała zapisany ciąg $c_1, c_2, \ldots, c_n$ możliwych konfiguracji maszyny $\mathcal{K}$, i po kolei każdą z nich będzie rozważała. Rozważając konfigurację c_i, w której obecny stan to q oraz litera pod głowicą to a, i zbiór dozwolonych w tej sytuacji tranzycji to

$$\begin{gathered}(q, a, b_1, d_1, q_1)\\(q, a, b_2, d_2, q_2)\\\ldots\\(q, a, b_k, d_k, q_k)\end{gathered}$$

(przy czym $k \leqslant |B| \times 2 \times |Q|$), maszyna $\mathcal{K}$ dopisuje do swojej taśmy k kopii obecnie rozważanej konfiguracji c_i, i w j-tej kopii dokonuje j-tą z powyższych tranzycji.

Gdy w którejś z rozważanych konfiguracji pojawi się stan akceptujący, maszyna $\mathcal{K}$ terminuje. ■

5
Teoria złożoności

5.1 Klasy P *oraz* NP

Powiemy, że język L jest w klasie P (od *polynomial*) jeżeli istnieje deterministyczna maszyna Turinga $\mathcal{M}$ oraz taki wielomian $f\colon \mathbb{N} \to \mathbb{N}$, że spełnione są następujące warunki:

- $L(\mathcal{M}) = L$,
- dla każdego słowa $w \in A^*$, maszyna $\mathcal{M}$ wykonuje co najwyżej $f(|w|)$ kroków na słowie w.

Mniej formalnie: $L \in$ P jeżeli istnieje algorytm rozwiązujący problem L w czasie wielomianowym.

Powiemy, że język L jest w klasie NP (od *nondeterministic-polynomial*) jeżeli istnieje niedeterministyczna maszyna Turinga $\mathcal{M}$ oraz taki wielomian $f\colon \mathbb{N} \to \mathbb{N}$, że spełnione są następujące warunki:

- $L(\mathcal{M}) = L$,
- dla każdego słowa $w \in A^*$, każdy bieg maszyny $\mathcal{M}$ po słowie w wykonuje co najwyżej $f(|w|)$ kroków.

zamiast drugiego warunku można rozważać następujący, słabszy warunek: dla każdego słowa $w \in L$ istnieje bieg maszyny $\mathcal{M}$ po słowie w, mający co najwyżej $f(|w|)$ kroków. Nietrudno pokazać, że obie definicje dają tę samą klasę języków.

Przykład 45. Rozważmy następujący problem: dany jest graf G. Należy rozstrzygnąć, czy G ma cykl eulerowski, tj. taki ciąg krawędzi, $v_1v_2, v_2v_3, \ldots, v_nv_1$, że każda krawędź grafu G występuje w tym ciągu dokładnie raz (nie rozróżniamy kierunków krawędzi, tj. $ab = ba$).

Problem ten jest w klasie NP: podobnie, jak w przykładzie 44, możemy niedeterministycznie wyszukać taki cykl, zaczynając od wierzchołka 0, aż znów trafimy do wierzchołka 0. Robiąc to, zapisujemy sobie na taśmie roboczej wszystkie odwiedzone krawędzi, i w każdym kroku sprawdzamy, czy nie przechodzimy po krawędzi już zużytej, a na koniec sprawdzamy jeszcze, czy wszystkie krawędzie są zużyte.

Istnieje też algorytm, który rozwiązuje ten problem w czasie wielomianowym: znany i prosty fakt mówi, że G ma cykl eulerowski wtedy, i tylko wtedy, gdy jest spójny i każdy jego wierzchołek ma parzystą liczbę sąsiadów. Tak więc, powyższy problem jest też w klasie P. ⌟

Przykład 46. Rozważmy następujący problem: dany jest graf G. Należy rozstrzygnąć, czy G ma cykl hamiltonowski, tj. taki ciąg wierzchołków, $v_1, v_2, \ldots, v_n, v_1$, że każdy wierzchołek grafu G występuje w tym ciągu dokładnie raz (z wyjątkiem v_1, który występuje dwa razy), oraz że $v_1v_2, v_2v_3, \ldots, v_nv_1$ są krawędziami grafu (niekoniecznie wszystkimi).

Znów, jak powyżej, problem ten jest w klasie NP: możemy niedeterministycznie wyszukać taki cykl, zaczynając od wierzchołka 0, aż znów trafimy do wierzchołka 0. Robiąc to, zapisujemy na taśmie roboczej wszystkie odwiedzone wierzchołki, i w każdym kroku sprawdzamy, czy nie przechodzimy do wierzchołka już zużytego, a na koniec sprawdzamy jeszcze, czy wszystkie wierzchołki są zużyte.

Nie wiadomo, czy istnieje algorytm, który rozwiązuje ten problem w czasie wielomianowym. Jak zobaczymy poniżej, implikowałoby to, że *każdy* problem w klasie NP można rozwiązać w czasie wielomianowym. ⌟

Przykład 47. Problem k-kolorowalności grafu (gdzie k jest ustaloną liczbą naturalną) jest następujący.

Problem: k-COLORABILITY
Dane: graf G
Rozstrzygnąć: czy istnieje takie kolorowanie wierzchołków grafu G za pomocą k kolorów, że każde dwa sąsiadujące wierzchołki mają różne kolory?

Dla każdej ustalonej liczby k, ten problem jest w klasie NP: intuicyjnie, kolorowanie można wpierw zgadnąć, a potem sprawdzić jego poprawność. Jednak problem 2-kolorowalności jest szczególny: wiadomo[1], że jest on w klasie P. Przypuszcza się, że problem 3-kolorowalności nie jest w klasie P. Jak zobaczymy poniżej, również implikowałoby to, że każdy problem w klasie NP można rozwiązać w czasie wielomianowym. ⌟

[1] to dlatego, że graf 2-kolorowalne można scharakteryzować jako te, które nie mają cyklu długości nieparzystej

Przykład 48. Problem pierwszości jest następujący.

Problem: PRIMALITY
Dane: liczba n, zapisana binarnie
Rozstrzygnąć: czy liczba n jest pierwsza?

W przeciwieństwie do pytań dotyczących rozstrzygalności, gdy rozważamy złożoność czasową obliczeń, istotne jest, czy liczby zapisujemy unarnie, czy binarnie – zapis unarny liczby n jest wykładniczo dłuższy, niż jej zapis binarny. Tak więc, jeśli liczba n jest zapisana

unarnie, to algorytm wielomianowy ma wykładniczo więcej czasu na jego rozwiązanie, niż gdy gdyby liczba n była zapisana binarnie (bo czas działania ma być wielomianowy względem długości zapisu).

W szczególności, bardzo łatwo widać, że powyższy problem jest wielomianowy, jeżeli liczba n jest zapisana unarnie: algorytm, który kolejno sprawdza podzielność n przez każdą liczbę $1 < k < n$, wykonuje liczbę kroków wielomianową względem liczby n. Jednak czas działania tego algorytmu jest wykładniczy względem długości zapisu binarnego liczby n. W praktyce, ten algorytm jest mało użyteczny: w kryptografii stosuje się liczby pierwsze których zapis binarny ma tysiące cyfr, czyli liczby rzędu 2^{1024} lub większe.

W roku 2002, Manindra Agrawal, Neeraj Kayal, oraz Nitin Saxena odkrli algorytm wielmianowy dla problemu pierwszości, działający w czasie $\mathcal{O}(\ell^{12})$ dla liczb o ℓ bitach[2]. Co ciekawe, wciąż nie wiadomo, jak dla danej liczby w czasie wielomianowym znaleźć jej rozkład na czynniki pierwsze. ⌟

[2] Ten czas został później zmniejszony do $\mathcal{O}(\ell^6)$

Hipoteza P $\neq$ NP. Hipoteza P $\neq$ NP to najważniejszy problem otwarty w informatyce teoretycznej. Mówi on, że klasy NP oraz P są różne. Oczywiście, zachodzi inkluzja P $\subseteq$ NP, więc hipoteza mówi, że istnieje problem który jest w NP ale nie w P. Co więcej – jeśli taki problem istnieje, to np. też problem 3-kolorowalności jest takim językiem. Jest więcej znanych problemów, które są takimi "kandydatami" na języki będące w NP $-$ P. O tym mówi następujący fakt.

Fakt 4. *Równoważne są warunki:*

- *(i)* P $=$ NP,
- *(ii) któryś z problemów z poniższej listy należy do klasy* P,
- *(iii) każdy z problemów z poniższej listy należy do klasy* P.

1. problem istnienia cyklu hamiltonowskiego w zadanym grafie,
2. problem 3-kolorowalność grafu,
3. kafelkowanie zadanego kwadratu: dana kolekcja kafelków oraz liczba k, czy istnieje kafelkowanie kwadratu $k \times k$ za pomocą tych kafelków, które w lewym dolnym rogu ma pierwszy kafelek z kolekcji, a w prawym górnym rogu ma drugi kafelek z kolekcji[3]?
4. problem kliki: dany graf G oraz liczba k. Czy G zawiera klikę rozmiaru k?
5. 3-CNF-SAT. Jest to problem spełnialności formuł w postaci 3-CNF które są koniunkcjami klauzul, gdzie każda klauzula jest

[3] Por. *kafelkowanie nieograniczonego kwadratu* w Rozdziale 4.3

W problemie kliki akurat bez znaczenia jest, czy k zapisujemy unarnie, czy binarnie, bo zawsze możemy założyć, że k jest mniejsze niż długość opisu grafu G.

alternatywą trzech literałów, a każdy literał to zmienna lub negacja zmiennej. Np. następująca formuła jest formułą w postaci 3-CNF $(x \vee y \vee \neg z) \wedge (y \vee \neg t \vee \neg x) \wedge (x)$. W problemie spełnialności pytamy, czy istnieje takie przypisanie zmiennym wartości 0 i 1, by formuła miała wartość 1.

Znanych są tysiące problemów, które tu by można wpisać. Niektóre z nich nadają się, lub są oparte o znane jednoosobowe gry logiczne, typu Sudoku.

6.-1000. ...

Powyższy fakt mówi, że znalezienie algorytmu wielomianowego dla dowolnego z powyższych problemów gwarantuje istnienie algorytmu wielomianowego dla *każdego* z powyższych problemów, a także dla każdego problemu z klasy NP. Ponieważ każdy z problemów na tej liście jest w klasie NP, to również, żeby pokazać że NP $\neq$ P wystarczy pokazać, że jakikolwiek z tych problemów nie jest w klasie P. Przykładowo, hipoteza NP $\neq$ P jest równoważna temu, że problem 3-kolorowalności nie ma algorytmu wielomianowego. Tak więc, żeby rozwiązać hipotezę NP $\neq$ P, wystarczy się skupić na jednym z powyższych problemów. Są tysiące problemów, które można by dopisać do powyższej listy – są to tak zwane problemy NP-*zupełne*. Przypomina to sytuację dotyczącą problemów nierozstrzygalnych – tam też znamy setki problemów, do których redukuje się problem HALT. Z resztą, metody dowodzenia w obu sytuacjach są bardzo podobne. Poniżej podamy przykłady kilku problemów NP-zupełnych.

Wpierw definicja: język $L \subseteq A^*$ jest NP-zupełny, jeżeli jest w klasie NP, oraz każdy język K w klasie NP ma redukcję wielomianową do języka L. Dokładniej, dla każdego języka $K \subseteq B^*$ takiego, że $K \in$ NP, istnieje funkcja $f\colon B^* \to A^*$ spełniająca następujące warunki:

- f jest obliczalna w czasie wielomianowym[4],
- dla każdego słowa $w \in B^*$ zachodzi równoważność

$$w \in K \leftrightarrow f(w) \in L.$$

[4] tj. istnieje wielomian $p\colon \mathbb{N} \to \mathbb{N}$ oraz maszyna Turinga $\mathcal{M}$, która dla danego słowa $w \in B^*$ oblicza słowo $f(w)$ w czasie $p(|w|)$

Jeżeli język L ma powyższą własność, to mówimy, że jest NP-trudny. Intuicyjnie, oznacza to, że jest on co najmniej tak trudny, jak każdy język w klasie NP. Tak więc, język L jest NP-*zupełny* jeżeli jest klasie NP oraz jest NP-trudny.

Lemat 19. *Niech $L \subseteq A^*$ oraz $K \subseteq B^*$ będą dwoma językami. Jeżeli istnieje redukcja wielomianowa $f\colon B^* \to A^*$ problemu K do problemu L oraz $L \in$ P to $K \in$ P.*

Dowód. Mamy następujący algorytm wielomianowy dla problemu K: *dla danego wejścia $w \in B^*$, oblicz $f(w)$ i sprawdź, czy $f(w) \in L$.* ∎

Wniosek 9. *Jeżeli* P $\neq$ NP *to żaden problem* NP*-trudny nie jest w klasie* P*.*

Dowód. Przypuśćmy, że problem $L \subseteq A^*$ jest NP-trudny oraz jest w klasie P. Pokażemy, że wtedy $\text{NP} \subseteq \text{P}$. Niech $K \in \text{NP}$. Na mocy NP-trudności L, istnieje wielomianowa redukcja $f\colon B^* \to A^*$ problemu K do problemu L. Na mocy lematu, $K \in \text{P}$, co dowodzi, że $\text{NP} \subseteq \text{P}$. Ponieważ druga inkluzja jest oczywista, dostajemy, że $\text{P} = \text{NP}$, co jest sprzeczne z założeniem. ∎

To, że istnieją problemy NP-zupełne jest nieoczywiste, a nawet zdumiewające. Poniżej pokażemy NP-zupełność kilku problemów z powyższej listy. Zaczniemy od lematu, który jest analogiem faktu o nierozstrzygalności problemu HALT_ε.

Lemat 20. *Następujący problem jest NP-zupełny: dla danej niedeterministycznej maszyny Turinga $\mathcal{M}$ oraz liczby n (zapisanej unarnie) stwierdzić, czy maszyna $\mathcal{M}$ akceptuje słowo puste w co najwyżej n krokach.*

Dowód. Wpierw zauważmy, że powyższy problem jest w klasie NP. Istotnie: żeby go rozwiązać, niedeterministycznie zgadujemy kolejne przejścia symulowanej maszyny $\mathcal{M}$, przez n kroków. Jeśli osiągniemy konfigurację akceptującą, akceptujemy, jeśli nie – odrzucamy. Jest to algorytm niedeterministyczny działający w czasie wielomianowym.

Teraz pokażemy, że rozważany problem jest NP-trudny. Niech $L \subseteq A^*$ będzie dowolnym językiem w klasie NP. Wówczas istnieje maszyna Turinga $\mathcal{M}$ oraz wielomian $f\colon \mathbb{N} \to \mathbb{N}$ taki, że $L(\mathcal{M}) = L$ oraz każdy bieg maszyny $\mathcal{M}$ na słowie długości n ma co najwyżej $f(n)$ kroków. Pokażemy redukcję języka L do problemu z tezy lematu.

Dla słowa $w \in A^*$, niech $\mathcal{M}_w$ będzie maszyną, która wpierw wypisuje na taśmie słowo w, a potem symuluje działanie maszyny $\mathcal{M}$. W szczególności, $\mathcal{M}_w$ akceptuje słowo puste wtedy i tylko wtedy, gdy $\mathcal{M}$ akceptuje słowo w. Redukcja języka L do problemu w treści lematu wygląda tak. Dla słowa $w \in A^*$, niech $f(w) = \mathit{kod}(\mathcal{M}_w)\#k$, gdzie gdzie k jest unarnym zapisem liczby $f(|w|) + |w|$. Łatwo widać, że liczbę k można obliczyć w czasie wielomianowym, dla danego słowa w. Ponadto, maszyna $\mathcal{M}_w$ akceptuje słowo puste w co najwyżej k krokach wtedy, i tylko wtedy, gdy maszyna $\mathcal{M}$ akceptuje słowo w.

A zatem, f jest redukcją problemu L do problemu w treści lematu. Ponadto, funkcja f jest obliczalna w czasie wielomianowym. Tak więc, pokazaliśmy, że każdy problem $L \in \text{NP}$ ma redukcję wielomianową do rozważanego problemu. A zatem, jest to problem NP-zupełny. ∎

W szczególności, powyższy problem jest w klasie P wtedy i tylko wtedy, gdy $\text{P} = \text{NP}$.

Otrzymaliśmy więc pierwszy problem NP-zupełny. Kolejne problemy otrzymamy redukując powyższy problem i korzystając z następującego lematu, którego dowód pozostawiamy jako ćwiczenie.

Lemat 21. *Jeżeli problem K jest* NP*-trudny oraz ma redukcję wielomianową do problemu L, to problem L też jest* NP*-trudny.*

W następnym kroku, analogicznie do nierozstrzygalności problemu kafelkowania nieograniczonego kwadratu, pokazujemy NP-zupełność problemu kafelkowania zadanego kwadratu.

Lemat 22. *Problem kafelkowania zadanego kwadratu jest* NP*-zupełny.*

Maszyna zgaduje kafelkowanie, a potem sprawdza, że jest ono poprawne.

Dowód. Łatwo zobaczyć, że problem kafelkowania zadanego kwadratu jest w klasie NP. Żeby pokazać, że jest on NP-zupełny, należy pokazać jego NP-trudność. Na mocy Lematu 21, wystarczy pokazać redukcję problemu z Lematu 20 do problemu kafelkowania zadanego kwadratu. W tym celu, wykorzystujemy tę samą redukcję, co podaną w Lemacie 18, i zauważamy, że działa ona także dla niedeterministycznych maszyn Turinga (wpierw musimy przerobić maszynę do specjalnej postaci, w której każdy stan maszyny wyznacza jednoznacznie kierunek następnego ruchu głowicy, i kierunek poprzedniego ruchu głowicy, żeby ruch głowicy maszyny, odczytany z kafelkowania, nie uległ "rozdwojeniu".). Obserwujemy przy tym, że tezę Lematu 18 można wzmocnić: funkcja f jest obliczalna w czasie wielomianowym.

Otrzymujemy więc redukcję wielomianową problemu z Lematu 20 do problemu kafelkowania. Ponieważ tamten problem jest NP-trudny, to wynika stąd, że problem kafelkowania też jest NP-trudny. ■

Kolejnym problemem NP-zupełnym jest problem spełnialności formuł boolowskich, w skrócie SAT. W tym problemie, dana jest formuła boolowska φ, zbudowana z koniunkcji $\wedge$, dyzjunkcji $\vee$, negacji $\neg$, nawiasów, oraz zmiennych $x, y, z, \ldots$ przyjmujących wartości boolowskie 0 oraz 1. Należy rozstrzygnąć, czy istnieje takie podstawienie wartości 0 oraz 1 za zmienne $x, y, z, \ldots$, że formuła φ przyjmuje wartość 1.

Twierdzenie 17 (Twierdzenie Cooka-Levina)**.** *Problem* SAT *jest* NP*-zupełny. W szczególności, jest on w klasie* P *wtedy, i tylko wtedy, gdy* $\mathrm{P} = \mathrm{NP}$.

Niedeterministyczna maszyna Turinga zgaduje wartościowanie zmiennych w danej formule, a potem sprawdza, że przy tym wartościowaniu, formuła ewaluuje się do 1.

Dowód. Łatwo zobaczyć, że problem SAT jest w klasie NP. Pozostaje pokazać, że problem SAT jest NP-trudny. Pokazujemy redukcję wielomianową z problemu kafelkowania zadanego kwadratu. Mając instancję kafelkowania, tj. kolekcję kafelków K oraz liczbę k, konstruujemy w czasie wielomianowym instancję SAT, będącą następującą formułą φ.

W formule φ, dla każdej pary współrzędnych $1 \leqslant i, j \leqslant k$ oraz dla każdego kafelka $t \in K$ tworzymy zmienną o nazwie $x^t_{i,j}$. Intencja jest taka: zmienna ta sygnalizuje, czy na współrzędnej (i, j) kładziemy kafelek t. Formułę φ konstruujemy jako koniunkcję następujących klauzul. Dla każdej pary i, j takiej, że $1 \leqslant i, j \leqslant k$, utwórzmy klauzulę gwarantującą, że dokładnie jedna ze zmiennych $x^t_{i,j}$, dla $t \in K$, ma wartość 1. Jeżeli przez $t_1, \ldots, t_p$ to wszystkie kafelki w kolekcji K, oraz zamiast $x^t_{i,j}$ napiszemy x^t dla uproszczenia notacji, to taka taka klauzula ma postać:

$$(x^{t_1} \vee x^{t_2} \vee \cdots \vee x^{t_p}) \wedge \neg(x^{t_1} \wedge x^{t_2}) \wedge \neg(x^{t_1} \wedge x^{t_3}) \wedge \ldots (x^{t_{p-1}} \wedge x^{t_p}).$$

Dalej, za pomocą tych zmiennych $x^t_{i,j}$, łatwo napisać formułę logiczną odpowiadającą warunkowi, że kafelkowanie jest poprawne. Dla każdych $1 \leqslant i < k$, $1 \leqslant j \leqslant k$, tworzymy klauzulę gwarantującą, że kafelki na pozycjach (i, j) oraz $(i, j + 1)$ są ze sobą zgodne. Jeżeli $(s_1, t_1), (s_2, t_2), \ldots, (s_u, t_u)$ to zbiór wszystkich tych par kafelków (s, t), że s można położyć na lewo od t, to klauzula ta ma postać:

$$(x^{s_1}_{i,j} \wedge x^{t_1}_{i+1,j}) \vee (x^{s_2}_{i,j} \wedge x^{t_2}_{i+1,j}) \vee \cdots \vee (x^{s_u}_{i,j} \wedge x^{t_u}_{i+1,j}).$$

Dodatkowo, tworzymy analogiczne klauzule odpowiadające zgodności kafelków w pionie. Wreszcie, formuła φ ma klauzule $x^{s_0}_{1,1}$ oraz $x^{t_0}_{1,1}$, gdzie s_0 i t_0 są odpowiednio, pierwszym i drugim kafelkiem w kolekcji K. Te klauzule wymuszają odpowiednie kafelki w lewym dolnym i prawym górnym rogu.

Widać, że formułę φ można skonstruować w czasie wielomianowym na podstawie kolekcji K oraz liczby k, oraz że formuła φ jest spełnialna wtedy, i tylko wtedy, gdy para K, k jest pozytywną instancją problemu kafelkowania. Tak więc, opisaliśmy wielomianową redukcję problemu kafelkowania do problemu SAT. Na mocy Lematu 21, dowodzi to NP-trudności problemu SAT. ∎

Żeby udowodnić NP-zupełność innych problemów wymienionych w Fakcie 4, wpierw pokazujemy NP-zupełność problemu 3-CNF-SAT. Z kolei większość pozostałych problemów redukuje się do problemu 3-CNF-SAT.

Lemat 23. *Problem 3-CNF-SAT jest* NP*-zupełny. W szczególności, jest on w klasie* P *wtedy, i tylko wtedy, gdy* P = NP.

Dowód. Jasne jest, że problem 3-CNF-SAT jest w klasie NP. Pokazujemy redukcję wielomianową problemu SAT do problemu 3-CNF-SAT. Idea jest taka. Dla danej formuły φ która jest ponawiasowana w taki sposób, że koniunkcje i dyzjunkcje są tylko binarne[5], tworzymy zmienną x_α dla każdej podformuły α formuły φ (w tym dla $\alpha = \varphi$).

[5] a więc, zamiast $\alpha \wedge \beta \wedge \gamma$ piszemy $(\alpha \wedge \beta) \wedge \gamma$

Konstruujemy formułę ψ jako koniunkcje następujących klauzul. Jeżeli α jest podformułą formuły φ postaci $\beta \vee \gamma$, to tworzymy klauzulę $x_\alpha \leftrightarrow x_\beta \vee x_\gamma$. Podobnie, jeżeli α jest postaci $\beta \wedge \gamma$, to tworzymy klauzulę $x_\alpha \leftrightarrow x_\beta \wedge x_\gamma$. Jeżeli α jest postaci $\neg\beta$, to tworzymy klauzulę $x_\alpha \leftrightarrow \neg x_\beta$. Dodajemy jeszcze klauzulę x_φ. Wreszcie, pozostaje przepisać klauzule postaci $x \leftrightarrow y, x \leftrightarrow y \vee z, x \leftrightarrow y \wedge z$ do postaci 3-CNF. Szczegóły pozostawiamy jako ćwiczenie. ■

Hipoteza $\text{P} \neq \text{NP}$ jest fascynująca między innymi z tego powodu, że całe programistyczne i matematyczne doświadczenie ludzkości zdaje się wyraźnie wskazywać na to, że klasy P i NP są różne, a jednocześnie wydaje się, że rozwiązanie problemu leży daleko poza zasięgiem znanych metod matematycznych[6]. Z drugiej strony, dowód, że $\text{P} = \text{NP}$, potencjalnie byłby brzemienny w skutki odczuwalne dla całej ludzkości[7]. Problem $\text{P} \stackrel{?}{=} \text{NP}$ i pokrewne problemy są tematem przedmiotu *Teoria złożoności*.

[6] Udowodnienie lub obalenie hipotezy będzie nagrodzone nagrodą w wysokości $1 mln.

[7] np. łamałoby by to współczesne systemy kryptograficzne.