KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt pn. "*Wzmocnienie potencjału dydaktycznego UMK w Toruniu w dziedzinach matematyczno-przyrodniczych*"
realizowany w ramach Poddziałania 4.1.1 Programu Operacyjnego Kapitał Ludzki

Języki Formalne i Automaty

dla słuchaczy kierunku Informatyka Stosowana

Miłosz Michalski

Instytut Fizyki UMK

UMK Toruń 2015

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

Spis treści

Wstęp

Skrypt ten powstał na bazie moich notatek, które przygotowywałem prowadząc od 2011 roku wykład z ćwiczeniami pt. "Języki formalne i automaty" dla studentów kierunku informatyka stosowana na Wydziale Fizyki, Astronomii i Informatyki Stosowanej UMK. Pierwotny program tego kursu budowałem opierając się na o bardzo dobrym podręczniku *An Introduction to Formal Languages and Automata* autorstwa P. Linza, [7], skąd pochodzi w szczególności wiele wykorzystanych tu zadań. Układ treści, który przyjąłem jest typowy dla większości książek nt. teorii języków formalnych. Prezentacja rozwijana jest od najwęższej kategorii języków regularnych, przez bardziej ogólne języki i gramatyki bezkontekstowe i kontekstowe, zgodnie z hierachizacją wprowadzoną przez N. Chomsky'ego. Osobny rozdział poświęcony jest maszynom Turinga i językom rekursywnie przeliczalnym, które zgodnie z hipotezą Churcha-Turinga stanowią najogólniejszą klasę języków formalnych, wyczerpujących intuicyjnie rozumianą kategorię wszelkich efektywnie wykonalnych procedur algorytmicznych. Dyskusja obejmuje w szczególności formalne ujęcie zagadnienia nierozstrzygalności, przykłady klasycznych problemów nierozstrzygalnych i przykład dowodu nierozstrzygalności pewnego problemu w klasie języków bezkontekstowych.

Polska wersja skryptu poszerzona została o dwa zagadnienia, które z powodu ograniczeń czasowych nie mieszczą się w programie wykładu. Chodzi o tzw. klasy $\mathcal{LL}(k)$ języków deterministycznych (podrozdział IV.2.2) oraz o zagadnienia z zakresu teorii złożoności obliczeniowej (rozdział VII). Ponieważ literatura w języku polskim, w odróżnieniu od angielskojęzycznej, jest w tym zakresie stosunkowo uboga, zdecydowałem się na opisanie tych zagadnień dla wygody słuchaczy kursu.

Miłosz Michalski
Zakład Fizyki Matematycznej IF UMK

Rozdział I

Podstawowe pojęcia matematyczne

Rozdział niniejszy zawiera przegląd i przypomnienie podstawowych pojęć i faktów matematycznych, których używać będziemy w dalszej części skryptu. Znaczna część prezentowanego tu materiału pojawia się także w kursach analizy matematycznej, algebry i matematyki dyskretnej.

I.1 Zbiory, relacje, funkcje, grafy

Zbiory

Zbiory uważane są za najbardziej fundamentalne pojęcie matematyczne. Najczęściej określamy je przez charakteryzację ich elementów. Na przykład formuła

$$A \;=\; \{n \in \mathbb{N} :\; n = 2k \;\text{ dla pewnego }\; k \in \mathbb{N}\}$$

określa A jako zbiór liczb parzystych. Zwróćmy uwagę, że definicja zbioru odwołuje się mniej lub bardziej jawnie do pewnego *uniwersum*, którym w naszym przykładzie są liczby naturalne $\mathbb{N}$. Chodzi o to, by określić jakiego typu obiekty stanowić będą elementy definiowanych zbiorów: czy będą to np. liczby naturalne, liczby rzeczywiste, trójkąty na płaszczyźnie, skończone ciągi zbudowane z liter 'a' i 'b' itp. W szczególności operacja dopełnienia zbioru, A^c, nie będzie jednoznaczna dopóki nie określimy względem jakiego uniwersum ma być obliczana.

Zbiory skończone określamy przez wypisanie ich elementów, np. $B = \{2, 4, 8, 16\}$. Zbiór pusty oznaczamy symbolem $\emptyset$. Ogólnie rzecz biorąc, zbiory opisywane są przez formuły $\Phi(x)$ zwane w logice funkcjami zdaniowymi, zawierające jedną lub więcej zmiennych x, które to formuły stają się zdaniami prawdziwymi lub fałszywymi po podstawieniu w miejsce x obiektu z sensownie wybranego uniwersum Ω. Mamy więc

$$A \;=\; \{x \in \Omega :\; \Phi(x)\}\,,$$

co wyrażamy słownie mówiąc, że A jest zbiorem tych i tylko tych obiektów x (typu Ω), które posiadają własność Φ lub — równoważnie — które spełniają formułę $\Phi(x)$. W poprzednio rozważanym przykładzie w roli Φ użyliśmy funkcji zdaniowej równoważnej stwierdzeniu, że "n jest podzielne przez 2".

W precyzyjnym, aksjomatycznym sformułowaniu teorii mnogości mówi się o ograniczeniach jakie muszą spełniać formuły, by budowane przy ich pomocy definicje

zbiorów nie prowadziły do logicznych paradoksów. Są to jednak sytuacje, z którymi raczej nie spotkamy się w praktycznych zastosowaniach.

Stwierdzenie, że a jest elementem zbioru A, w zapisie $a \in A$ oznacza, że a spełnia funkcję zdaniową $\Phi(x)$ definiującą zbiór A, a więc, że zdanie $\Phi(a)$ jest prawdziwe. Prócz tego określamy relacje zawierania i równości zbiorów:

$$\begin{aligned} A \subset B \qquad & \text{wtedy i tylko wtedy, gdy} \qquad x \in A \quad \Rightarrow \quad x \in B \\ & \text{wtedy i tylko wtedy, gdy} \qquad \Phi(x) \quad \Rightarrow \quad \Psi(x)\,, \end{aligned}$$

gdzie formuła Φ definiuje zbiór A, a Ψ — zbiór B. Ponadto

$$A = B \qquad \text{wtedy i tylko wtedy, gdy} \qquad A \subset B \quad \text{i} \quad B \subset A\,.$$

Piszemy także

$$A \subseteq B \qquad \text{wtedy i tylko wtedy, gdy} \qquad A \subset B \quad \text{lub} \quad A = B\,.$$

Symbol "$\subseteq$" jest co prawda logicznie równoważny symbolowi "$\subset$", jednak pozwala na wprowadzenie nieco bardziej precyzyjnej notacji. Bowiem jeśli chcemy podkreślić, że zawieranie zbiorów jest właściwe, to znaczy, że $A \subset B$, ale $A \neq B$, piszemy wówczas $A \subsetneq B$. Zapis $A \not\subset B$ oznacza natomiast, że A nie zawiera się w B.

Relacje "$\subseteq$" i "$\subsetneq$" między zbiorami pełnią więc rolę analogiczną do relacji "$\leqslant$" oraz "$<$" między liczbami. O ile jednak zbiór liczb rzeczywistych jest uporządkowany *liniowo*, to znaczy, że między dwiema dowolnymi liczbami jakaś relacja porządku zawsze zachodzi, bo albo $a \leqslant b$ albo $b \leqslant a$, o tyle relacja zawierania opisuje tzw. *częściowy porządek* wśród zbiorów: w gruncie rzeczy większości par zbiorów nie wiąże ani relacja $A \subseteq B$, ani też $B \subseteq A$.

Zwróćmy uwagę, że zbiory same mogą być elementami innych zbiorów, często nazywanych rodzinami zbiorów, np.

$$\mathcal{X} \;=\; \Big\{(a,b) :\; a,b \in \mathbb{R},\; a < b\Big\}$$

jest rodziną ograniczonych przedziałów otwartych zawartych w $\mathbb{R}$. Szczególnie ważny jest tzw. *zbiór potęgowy* zbioru A,

$$\mathcal{P}(A) \;\stackrel{\text{df}}{=}\; \{X :\; X \subseteq A\}\,,$$

będący, zgodnie z określeniem, rodziną wszystkich podzbiorów zbioru A, włączając zbiór pusty $\emptyset$ i sam zbiór A. W szczególnych okolicznościach, gdy zależy nam na wyodrębnieniu jedynie *skończonych* podzbiorów A, piszemy

$$\mathcal{F}(A) \;\stackrel{\text{df}}{=}\; \{X :\; X \subseteq A \quad \text{i} \quad X \text{ ma skończenie wiele elementów}\}\,.$$

Łatwo przekonać się, że A jest skończony wtedy i tylko wtedy, gdy $\mathcal{P}(A) = \mathcal{F}(A)$, por. Zad. 3 d.

Operacje mnogościowe

Przyjmijmy, że $A = \{x \in \Omega :\; \Phi(x)\}$ oraz $B = \{x \in \Omega :\; \Psi(x)\}$. Logiczne operacje

koniunkcji i alternatywy oznaczamy tradycyjnie symbolami "$\wedge$" oraz "$\vee$". Ponadto "$\neg$" oznacza logiczną negację. Przypomnijmy definicje podstawowych operacji na zbiorach:

$$A \cup B \;=\; \{x \in \Omega :\; \Phi(x) \vee \Psi(x)\} \;=\; \{x \in \Omega :\; x \in A \vee x \in B\}$$

oraz

$$A \cap B \;=\; \{x \in \Omega :\; \Phi(x) \wedge \Psi(x)\} \;=\; \{x \in \Omega :\; x \in A \wedge x \in B\}\,.$$

Mamy także

$$A^c \;=\; \{x \in \Omega :\; \neg\Phi(x)\} \;=\; \{x \in \Omega :\; x \notin A\}\,.$$

Dodatkowo wprowadzamy operacje różnicy i różnicy symetrycznej zbiorów:

$$A - B \;=\; \{x \in \Omega :\; \Phi(x) \wedge \neg\Psi(x)\} \;=\; \{x \in \Omega :\; x \in A \wedge x \notin B\} \;=\; A \cap B^c$$

oraz

$$\begin{aligned} A \oplus B \;&=\; (A - B) \,\cup\, (B - A) \;=\; (A \cup B) \,-\, (A \cap B) \\ &=\; (A^c \cup B)^c \,\cup\, (A \cup B^c)^c\,. \end{aligned} \tag{1.1}$$

Operacje na dwóch zbiorach uogólniamy na dowolne rodziny zbiorów:

$$\begin{aligned} \bigcup \mathcal{A} \;&=\; \{x :\; \exists\, A \in \mathcal{A} \quad \text{taki że} \quad x \in A\} \\ \bigcap \mathcal{A} \;&=\; \{x :\; \forall\, A \in \mathcal{A} \quad \text{zachodzi} \quad x \in A\}\,. \end{aligned}$$

Zauważmy w szczególności, że $\bigcup \mathcal{P}(A) = A$, suma mnogościowa jest więc w pewnym sensie operacją odwrotną do potęgowania zbioru.

Uogólnione sumy i przekroje najczęściej pojawiają się w szczególnej postaci, gdy rodzina $\mathcal{A}$ jest *indeksowaną* rodziną zbiorów, to jest gdy elementy $\mathcal{A}$ odróżniamy poprzez przyporządkowane im indeksy α z pewnego zbioru indeksów I:

$$\mathcal{A} \;=\; \{A_\alpha :\; \alpha \in I\}\,.$$

Piszemy wówczas

$$\bigcup \mathcal{A} \;=\; \bigcup_{\alpha \in I} A_\alpha \;=\; \{x :\; \exists\, \alpha \in I \quad \text{taka że} \quad x \in A_\alpha\}$$

i podobnie dla $\bigcap \mathcal{A}$. Gdy $I = \mathbb{N}$ stosujemy znajomą notację

$$\bigcup_{n=1}^{\infty} A_n \qquad \text{oraz} \qquad \bigcap_{n=1}^{\infty} A_n\,.$$

Zarówno dla skończonych jak i nieskończonych rodzin zbiorów zachodzą prawa de Morgana:

$$\left(\bigcup_{\alpha \in I} A_\alpha\right)^c \;=\; \bigcap_{\alpha \in I} A_\alpha^c \qquad \text{oraz} \qquad \left(\bigcap_{\alpha \in I} A_\alpha\right)^c \;=\; \bigcup_{\alpha \in I} A_\alpha^c\,.$$

Innym rodzajem operacji na zbiorach jest mnożenie kartezjańskie, które tworzy zbiory uporządkowanych par elementów swoich czynników,

$$A \times B \;=\; \{(a,b):\; a \in A \wedge b \in B\}\,.$$

W parach uporządkowanych, jak to sugeruje nazwa, istotna jest kolejność elementów, a więc $(a,b) \neq (b,a)$, chyba że $a = b$. Istnieje formalna definicja pary uporządkowanej podana przez K. Kuratowskiego, redukująca to pojęcie do zwykłych zbiorów:

$$(a,b) \;\stackrel{\text{df}}{=}\; \Big\{a, \{a,b\}\Big\}\,.$$

Zauważmy, że istotnie $(a,b) \neq (b,a)$, bo zbiory $\Big\{a, \{a,b\}\Big\}$ i $\Big\{b, \{a,b\}\Big\}$ różnią się pierwszym elementem gdy tylko $a \neq b$.

Podobnie, $A \times B \times C$ jest zbiorem uporządkowanych trójek (a,b,c) elementów odpowiednio zbiorów A, B i C. W przypadku gdy $A = B$, piszemy krótko A^2 zamiast $A \times A$, A^3 zamiast $A \times A \times A$ lub ogólnie A^n na oznaczenie n-elementowych ciągów utworzonych z obiektów $a \in A$. W ten sposób obiekt

$$\bigcup_{n=1}^{\infty} A^n = A \cup A^2 \cup A^3 \cup \ldots$$

jest zbiorem wszystkich *skończonych* ciągów zbudowanych z elementów zbioru A.

Relacje

DEFINICJA 1.1 *Relacją R między elementami zbiorów A i B nazywamy dowolny podzbiór iloczynu kartezjańskiego $A \times B$, a więc $R \subset A \times B$. Jeśli para $(a,b) \in R$ mówimy też, że między elementami a i b zachodzi relacja R.*

Stosujemy także bardziej naturalną notację infiksową $a \sim_R b$, nie zmienia to jednak faktu, że relacja formalnie rzecz biorąc odpowiada zbiorowi par $(a,b) \in A \times B$ obiektów nią powiązanych. Gdy $A = B$, mówimy o relacji w zbiorze A. Przykładem relacji w zbiorze liczb rzeczywistych jest relacja porządku $a \leqslant b$. Relacja podzielności $k|n$ wiąże ze sobą niektóre pary liczb naturalnych. Relacja podobieństwa kojarzy ze sobą pary trójkątów na płaszczyźnie. Jak wspomnieliśmy wyżej, relacja inkluzji $\subseteq$ jest relacją w zbiorze potęgowym $\mathcal{P}(A)$ dowolnego zbioru A.

Oto niektóre z często używanych własności relacji.

DEFINICJA 1.2 *Definiujemy następujące własności relacji R określonej w A:*

a) zwrotność: $\forall\, a \in A \quad (a,a) \in R$;

b) spójność: $\forall\, a,b \in A \quad (a,b) \in R$ *lub* $(b,a) \in R$;

c) symetria: $(a,b) \in R \Rightarrow (b,a) \in R$;

d) antysymetria: jeśli $(a,b) \in R$, *wówczas* $(b,a) \notin R$;

e) słaba antysymetria: jeśli $(a,b) \in R$ *i* $(b,a) \in R$, *wówczas* $a = b$;

f) przechodniość (tranzytywność): jeśli $(a,b) \in R$ *i* $(b,c) \in R$, *to także* $(a,c) \in R$.

Każdą relację, która jest zwrotna, symetryczna i przechodnia nazywamy relacją równoważności w A. Częściowy porządek w A jest relacją o własnościach zwrotności, słabej antysymetrii i przechodniości. Jeśli jest on dodatkowo relacją spójną, wówczas nazywamy go porządkiem liniowym.

Inkluzja jest porządkiem częściowym na zbiorach potęgowych, natomiast zwykła relacja "$\leqslant$" jest porządkiem liniowym w zbiorze liczb rzeczywistych. Oto przykład nietrywialnej relacji równoważności w zbiorze liczb całkowitych. Niech p będzie ustaloną liczbą naturalną. Określmy relację "$\sim_p$" następująco:

$$m \sim_p n \qquad \text{wtedy i tylko wtedy, gdy} \qquad m - n = 0 \,(\mathrm{mod}\, p)\,.$$

Innymi słowy $m \sim_p n$ oznacza, że liczby m i n dają identyczną resztę z dzielenia przez p. Fakt, że jest to relacja równoważności powinien być teraz oczywisty.

Każda relacja równoważności określona na zbiorze jednoznacznie dzieli ten zbiór na rozłączne podzbiory elementów wzajemnie względem niej równoważnych.

DEFINICJA 1.3 *Klasą abstrakcji elementu $a \in A$ względem relacji równoważności R określonej na A nazywamy zbiór*

$$[a]_R \;=\; \{b \in A :\; b \sim_R a\}$$

i mówimy, że a jest reprezentantem tej klasy. Zbiór klas abstrakcji relacji równoważności R,

$$\mathcal{R} \;=\; \{[a]_R : a \in A\}\,,$$

nazywamy zbiorem ilorazowym A względem R i oznaczamy symbolem $A\Big/_R$.

Jak łatwo sprawdzić, każdy element $a \in A$ należy do dokładnie jednej klasy abstrakcji względem R. Jeśli $a \sim_R b$ wówczas $[a]_R = [b]_R$, klasa abstrakcji nie zależy więc od szczególnego wyboru swojego reprezentanta.

Opisana w definicji konstrukcja ilorazowa $A\Big/_R$ jest jedną z najważniejszych technik definiowania nowych obiektów w matematyce. Pozwala ona utożsamić ze sobą te elementy wyjściowego zbioru, które uznajemy za równoważne ze względu na relację R. W przypadku relacji $\sim_p$ na $\mathbb{Z}$ opisanej wyżej, zbiór ilorazowy to

$$\mathbb{Z}\Big/_p \;=\; \{[0],[1],\ldots,[p-1]\} \quad \leftrightarrow \quad \{0,1,\ldots,p-1\}\,,$$

gdzie przez symbol podwójnej strzałki rozumiemy tu wzajemnie jednoznaczną reprezentację zbioru ilorazowego $\mathbb{Z}\Big/_p$ przez skończony zbiór liczb $\{0,1,\ldots,p-1\}$, a więc zbiór reszt z dzielenia przez p.

Dla dowolnej relacji R w zbiorze A można skonstruować tzw. *tranzytywne domknięcie* $\hat{R}$.

DEFINICJA 1.4 *Tranzytywnym domknięciem relacji R na zbiorze A nazywamy najmniejszą w sensie inkluzji relację przechodnią $\hat{R} \subset A \times A$ taką, że $R \subset \hat{R}$.*

Konstrukcyjnie, tranzytywne domknięcie relacji R polega na dodaniu do niej pewnej liczby par $(x,y) \in A \times A$ tak, aby wynikowy zbiór $\hat{R}$ spełniał warunek przechodniości f) z Definicji 1.2.

Funkcje

Funkcje są szczególnym przypadkiem relacji.

DEFINICJA 1.5 *Relację $F \subset A \times B$ nazywamy funkcją, gdy zachodzi warunek*

$$(a,b) \in F \quad i \quad (a,c) \in F \qquad \Rightarrow \qquad b = c\,.$$

Funkcja jest więc relacją, która w sposób jednoznaczny wiąże z elementem $a \in A$ pewien element $b \in B$. W tradycyjnej notacji napiszemy $b = F(a)$ i powiemy, że wartości funkcji obliczają się jednoznacznie.

Nadużywając notacji funkcyjnej do relacji, moglibyśmy napisać $b = R(a)$, gdy $(a,b) \in R$, ale ponieważ relacja R może w ogólności wiązać więcej niż jeden element b z a, prowadziłoby to do niezręcznego zapisu np. $b = R(a) = c$, choć $b \neq c$. Bardziej trafiona byłaby w tej sytuacji następująca notacja: $R(a) = \{b, c\}$. W istocie jest to zalążek konstrukcji, którą wykorzystamy w niedalekiej przyszłości. Wyznaczmy zbiory

$$\forall\, a \in A \qquad R(a) \stackrel{\text{df}}{=} \{b \in B :\ (a,b) \in R\}\,.$$

Dla pewnych a możliwe jest, że $R(a) = \emptyset$, ale to w niczym nie psuje naszej konstrukcji. Określmy teraz nową relację $\tilde{R} \subset A \times \mathcal{P}(B)$ przez

$$\tilde{R} \;=\; \{(a, R(a)) :\ a \in A\}\,. \tag{1.2}$$

Jak łatwo sprawdzić, $\tilde{R}$ jest już funkcją. Funkcja ta niesie taką samą informację o powiązaniach elementów A z elementami B.

W przypadku gdy relacja $F \subset A \times B$ jest funkcją, dla uproszczenia przyjmiemy konwencję, że F określona jest dla każdego $a \in A$. Wrócimy wówczas do tradycyjnej notacji $F : A \to B$ i nazwiemy A dziedziną funkcji F, $D(F) = A$, natomiast B — jej przeciwdziedziną, $W(F) = B$. W przypadku gdy F nie jest określona na całym zbiorze A, mówimy o funkcji *częściowej*, co zapisujemy symbolicznie $F : A \nrightarrow B$. Wówczas oczywiście $D(F) \subset A$. Gdy wyjątkowo chcemy podkreślić, że F nie jest częściowa, wówczas mówimy, że jest funkcją *totalną* (choć oczywiście w przyjętej tu konwencji oznacza to ni mniej nie więcej niż to, że F jest po prostu funkcją).

Dwa użyteczne pojęcia związane z funkcjami to obraz zbioru $U \subset A$ przez $F : A \nrightarrow B$,

$$F(U) \stackrel{\text{df}}{=} \{b \in B :\ \exists\, a \in U \quad \text{takie że} \quad b = F(a)\}$$

oraz przeciwobraz zbioru $V \subset B$ przez F,

$$F^{-1}(V) \stackrel{\text{df}}{=} \{a \in A :\ F(a) \in V\}\,.$$

Dla dowolnych zbiorów $U \subset A$ i $V \subset B$ mamy

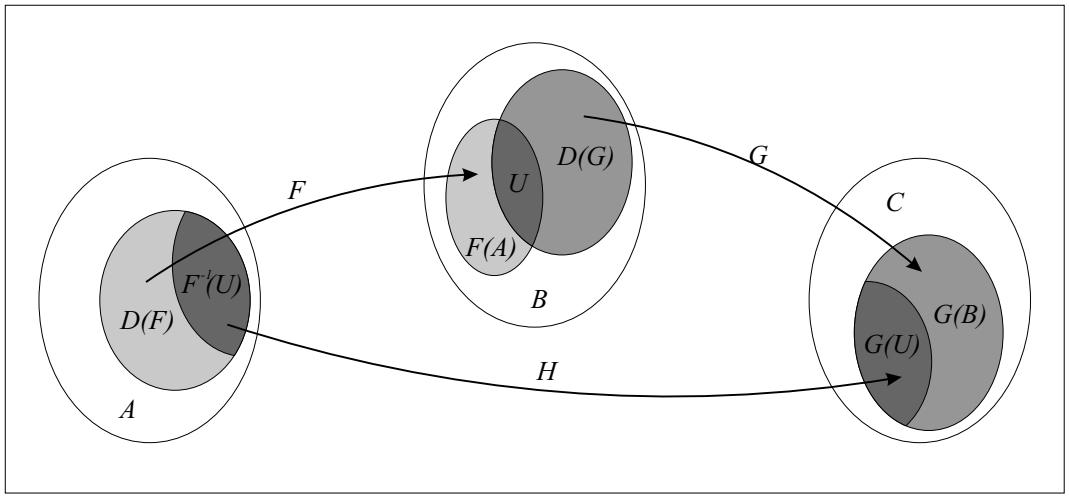
$$F(U) \subseteq W(F) \qquad \text{oraz} \qquad F^{-1}(V) \subseteq D(F)\,,$$

a w szczególności $F^{-1}(B) = D(F)$.

Dla dwóch częściowych odwzorowań $F : A \nrightarrow B$ i $G : B \nrightarrow C$ możemy określić ich złożenie $H = G \circ F : A \nrightarrow C$ za pomocą wzoru

$$H(a) = G(F(a))\,,$$

jeśli tylko spełniony jest warunek kompatybilności: $F(A) \cap D(G) \neq \emptyset$. Dziedziną złożenia jest wówczas zbiór $D(H) = F^{-1}\Big(F(A) \cap D(G)\Big)$.



Rys. 1.1: Złożenie $H = G \circ F$ częściowych odwzorowań $F : A \twoheadrightarrow B$ i $G : B \twoheadrightarrow C$. $U = F(A) \cap D(G)$ oraz $D(H) = F^{-1}(U)$, $H(A) = G(U)$.

DEFINICJA 1.6 *Dla funkcji $F : A \to B$ określone są następujące własności:*

a) różnowartościowość: jeśli $a \neq b$, wówczas także $F(a) \neq F(b)$

b) własność "na": $\forall b \in B \quad \exists a \in A$ takie, że $b = F(a)$;

c) wzajemna jednoznaczność: F jest różnowartościowa i "na".

Funkcje typu a) nazywamy injekcjami, typu b) — surjekcjami, natomiast odwzorowania wzajemnie jednoznaczne określamy mianem bijekcji.

Grafy

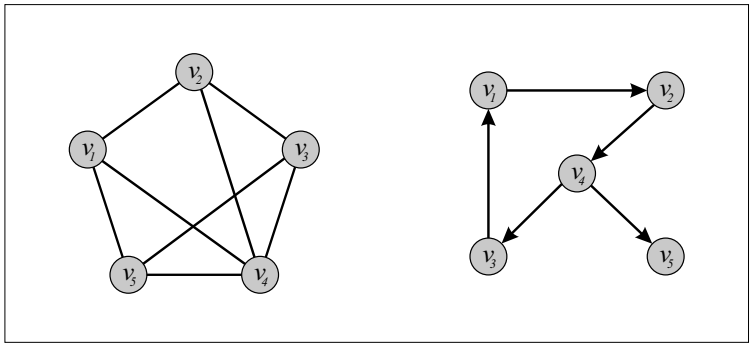
DEFINICJA 1.7 *Grafem nazywamy strukturę $G = (V, E)$ składającą się ze skończonego zbioru wierzchołków V i skończonego zbioru krawędzi łączących ze sobą pary wierzchołków. Krawędzie grafu utożsamiamy z nieuporządkowanymi parami wierzchołków, $e = \{u, v\}$, $u, v \in V$.*

Grafem skierowanym lub digrafem *nazywamy podobną strukturę, w której krawędzie są parami uporządkowanymi, $e = (u, v)$, a więc $E \subset V \times V$.*

Digraf jest więc w istocie pewną relacją na zborze wierzchołków. Jeśli w grafie nieskierowanym potraktujemy każdą krawędź $\{u, v\}$ jako reprezentację obydwu par uporządkowanych (u, v) i (v, u) na raz, wówczas graf ten można utożsamiać z pewną relacją symetryczną.

Istnieje prosta i użyteczna reprezentacja grafów w postaci diagramów, p. Rys. 1.2.

DEFINICJA 1.8 *Ścieżką między wierzchołkami u i v w grafie $G = (V, E)$ nazywamy skończony ciąg krawędzi $e_1, \ldots, e_k \in E$, taki że $e_1 = \{u, w_1\}$, $e_2 = \{w_1, w_2\}$, $\ldots$, $e_{k-1} = \{w_{k-2}, w_{k-1}\}$ oraz $e_k = \{w_{k-1}, v\}$.*

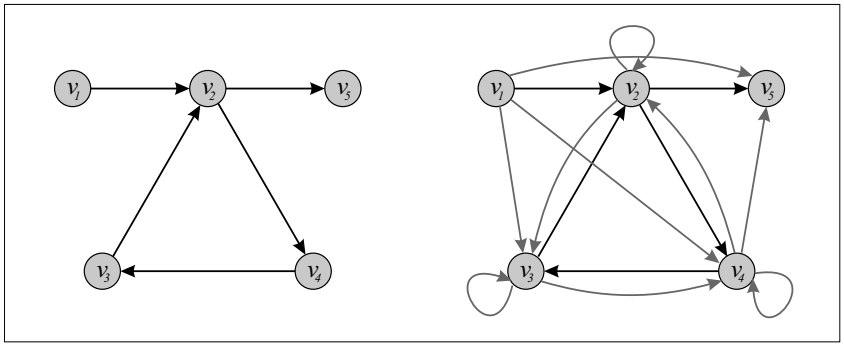


Rys. 1.2: Graf nieskierowany i graf skierowany.

Podobnie definiujemy ścieżkę w grafie skierowanym (istotna jest wówczas orientacja krawędzi). *Ścieżką prostą* nazywamy ścieżkę, która nie przechodzi wielokrotnie przez te same wierzchołki. *Cykle* są to ścieżki zamknięte, tj. takie że $u = v$. Cykle skierowane i cykle proste definiuje się analogicznie.

Graf nieskierowany nazywamy spójnym jeśli między każdą parą jego wierzchołków istnieje ścieżka. W digrafie spójnym dla każdej pary wierzchołków istnieje co najmniej jedna ze ścieżek skierowanych: z u do v lub z v do u. Jeśli zawsze istnieją obie takie ścieżki, digraf nazywamy wówczas *silnie spójnym*.

Tranzytywne domknięcie grafu G jest to najmniejszy (w sensie liczby krawędzi) graf $\hat{G} = (V, \hat{E})$ taki, że $E \subset \hat{E}$ oraz że relacja na V, którą opisuje $\hat{G}$ jest przechodnia. Rys. 1.3 przedstawia przykład digrafu i jego tranzytywnego domknięcia.



Rys. 1.3: Graf i jego tranzytywne domknięcie. Dodane krawędzie oznaczono szarym kolorem.

Korzystać będziemy także z grafów *etykietowanych*, $G = (V, E, \eta)$. Dodatkowym elementem jest tu funkcja $\eta : E \to X$ przyporządkowująca krawędziom grafu "etykiety" ze zbioru X. Jak się przekonamy dalej, grafy etykietowane stanowią bardzo wygodny sposób reprezentacji automatów.

Szczególnym rodzajem grafów są *drzewa*, Rys. 1.4.

DEFINICJA 1.9 *Drzewem nieskierowanym nazywamy graf, w którym między każdą parą wierzchołków istnieje dokładnie jedna ścieżka prosta.*

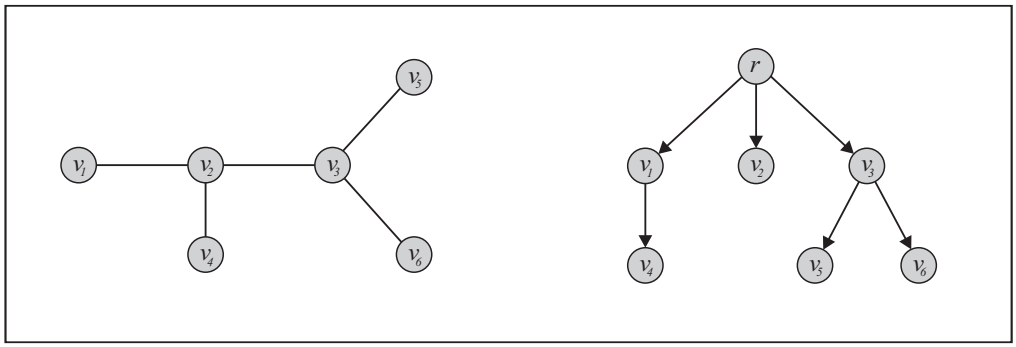
Alternatywnie, graf nieskierowany G spełniający dowolny z poniższych warunków jest drzewem:

— G jest grafem spójnym bez cykli;

— G nie zawiera cykli, lecz dodanie do niego *dowolnej* krawędzi zamyka dokładnie jeden cykl prosty;

— G jest minimalnym grafem spójnym (t.j. usunięcie dowolnej krawędzi powoduje rozspójnienie G);

— G posiada n wierzchołków i $n-1$ krawędzi.



Rys. 1.4: Drzewo nieskierowane i skierowane.

Istnieją różne warianty definicji drzewa skierowanego, dla naszych celów jednak przydatne będzie następujące określenie.

DEFINICJA 1.10 Drzewem skierowanym *nazywamy digraf D, w którym istnieje jeden wyróżniony wierzchołek r zwany* korzeniem*, a każdy inny wierzchołek jest osiągalny z niego dokładnie jedną ścieżka skierowaną. Odległość (mierzoną liczbą krawędzi) wierzchołka v od korzenia r nazywamy* poziomem *v w drzewie D. Zbiór wierzchołków tego samego poziomu m nazywamy m-tym piętrem drzewa D. Wierzchołki końcowe ścieżek w D nazywamy* liśćmi.

I.2 Moc zbiorów: zbiory przeliczalne i nieprzeliczalne

Jedyną własnością, którą posiadają wszystkie zbiory i która jest całkowicie niezależna od charakteru ich elementów, jest ich *moc* czyli liczba zawartych w nich elementów. Chodzi o to, że np. zbiory $\{5, 10, 15, 20\}$ i $\{\spadesuit, \heartsuit, \diamondsuit, \clubsuit\}$ są podobne lub wręcz nierozróżnialne gdy abstrahujemy od tego czym są ich elementy. Obydwa te zbiory są elementami klasy abstrakcji wszystkich zbiorów 4-elementowych. Moc zbioru A oznaczamy symbolem $|A|$. Zajmiemy się obecnie rozróżnieniem pewnych klas zbiorów nieskończonych.

DEFINICJA 1.11 *O dwóch zbiorach X i Y mówimy, że są równoliczne jeśli istnieje bijekcja $F : X \to Y$, a więc odwzorowanie różnowartościowe i "na", ustalające między ich elementami wzajemnie jednoznaczną odpowiedniość. Piszemy wówczas $X \simeq Y$.*

Jak można się przekonać, równoliczność zbiorów jest relacją równoważności. Jest ona zwrotna, bo każdy zbiór X jest równoliczny z samym sobą przez bijekcję identycznościową $F(x) = x$. Jest również symetryczna bo odwzorowanie odwrotne do bijekcji

jest także bijekcją. W końcu, jest to relacja przechodnia dzięki temu, że złożenie bijekcji jest bijekcją. Równoliczność kategoryzuje więc zbiory na klasy abstrakcji zbiorów o identycznej mocy. Liczby naturalne 1,2,3,... uważać można za reprezentację klas abstrakcji zbiorów skończonych o 1, 2, 3 itd. elementach. Dodatkowo liczba 0 reprezentuję klasę abstrakcji zbioru pustego.

Podobnie, możemy określić relację $X \lesssim Y$, która oznacza, że X jest co najwyżej tak liczne jak Y, gdy istnieje odwzorowanie różnowartościowe $F : X \to Y$. Można łatwo przekonać się, że jeśli zarówno $X \lesssim Y$ i $Y \lesssim X$, wówczas $X \simeq Y$. Ponadto w celu wykazania równoliczności pewnych zbiorów często używa się następującego argumentu:

$$\text{Jeśli} \quad X \lesssim Y \lesssim Z \quad \text{oraz} \quad X \simeq Z \quad \text{wówczas} \quad X \simeq Y \simeq Z\,. \tag{1.3}$$

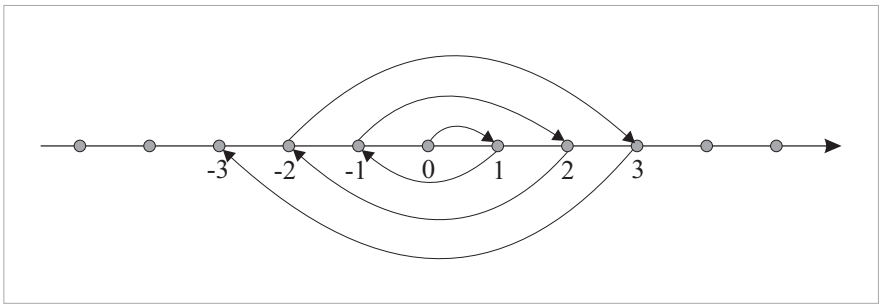
PRZYKŁAD 1.1 Jeśli zbiór A ma n elementów, moc jego zbioru potęgowego $\mathcal{P}(A)$ wynosi 2^n. Z tej przyczyny zbiór potęgowy oznaczany jest często symbolem 2^A. Znajdziemy bijekcję między zbiorem I_n złożonym ze wszystkich n-elementowych ciągów zero-jedynkowych a zbiorem $\mathcal{P}(A)$. Ustalmy ponumerowanie elementów $A = \{a_1, a_2, \ldots, a_n\}$. Dowolny zbiór $U \in \mathcal{P}(A)$ zakodujemy ciągiem binarnym w taki sposób, że w ciągu tym na i-tym miejscu stoi cyfra 1 wtedy i tylko wtedy, gdy $a_i \in U$. Zatem gdy $a_i \notin U$ w kodującym ciągu na i-tej pozycji pojawi się 0. Zbiór pusty reprezentowany jest przez ciąg złożony z samych zer, zaś sam zbiór A — przez ciąg jedynek. Oczywiste jest, że nasze kodowanie ustala wzajemnie jednoznaczną odpowiedniość między ciągami binarnymi a różnymi podzbiorami A: każdy ciąg reprezentuje unikalny podzbiór i każdemu podzbiorowi odpowiada dokładnie jeden ciąg. Stąd zbiór $\mathcal{P}(A)$ jest równoliczny z I_n, a liczba elementów tego ostatniego wynosi 2^n.

Zbiór X nazywamy *przeliczalnym*, jeśli jest on równoliczny ze zbiorem liczb naturalnych $\mathbb{N}$. Istnienie bijekcji $F : \mathbb{N} \to X$ oznacza po prostu, że elementy zbioru X można unikalnie ponumerować, a więc ustawić w ciąg, w którym każdy element X występuje dokładnie 1 raz, $X = \{x_1, x_2, \ldots\}$. Zbiór, który jest skończony lub przeliczalny nazywamy *co najwyżej przeliczalnym*.

PRZYKŁAD 1.2 Zbiór liczb całkowitych jest przeliczalny. Przekonamy się o tym, jeśli wskażemy sposób kompletnego ponumerowania wszystkich liczb całkowitych. Można to zrobić w sposób zilustrowany na Rys. 1.5: liczba 0 otrzymuje numer 1, liczba 1 numer 2, liczba -1 numer 3 itd.

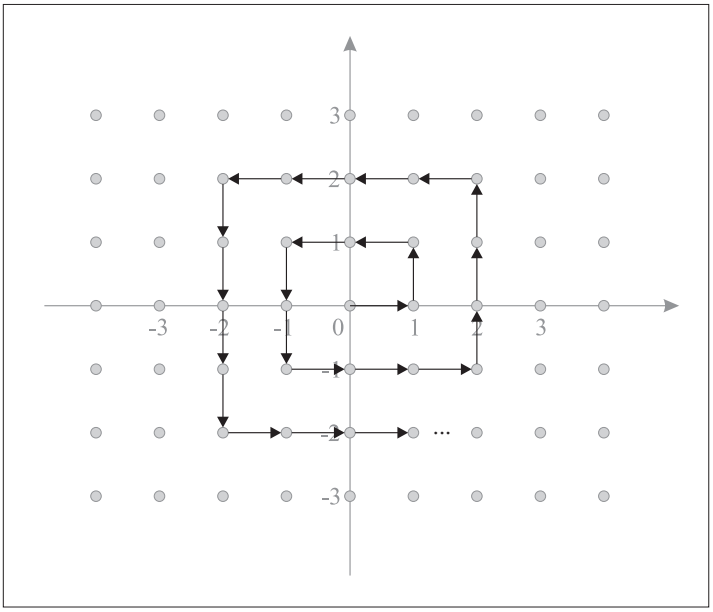
Zwróćmy uwagę na fakt, że pomimo tego iż $\mathbb{N} \subsetneq \mathbb{Z}$, zbiory te są równoliczne. Własność tego typu nie zachodzi z pewnością dla zbiorów skończonych: każdy podzbiór właściwy skończonego zbioru A ma z pewnością mniej elementów niż A. Musimy przyzwyczaić się jednak do zupełnie nieintuicyjnych faktów, gdy badamy relacje między zbiorami nieskończonymi.

PRZYKŁAD 1.3 Zbiór liczb wymiernych jest przeliczalny. By się o tym przekonać, zacznijmy od ponumerowania punktów o współrzędnych całkowitoliczbowych (m, n)



Rys. 1.5: Sposób numeracji liczb całkowitych.

na płaszczyźnie. Rys. 1.6 pokazuje właściwy sposób numeracji: punkt (0,0) otrzymuje numer 1, punkt (1,0) numer 2, punkt (1,1) numer 3 itd. zgodnie ze wskazaniami strzałek. W ten sposób każdy punkt (m, n) otrzymuje unikalny numer N i wszystkie liczby naturalne są wykorzystane w tej numeracji. Zatem zbiór punktów $\mathbb{Z} \times \mathbb{Z}$ jest równoliczny z $\mathbb{N}$. Z kolei liczby wymierne reprezentowane są przez nieskracal-



Rys. 1.6: Sposób numeracji punktów o współrzędnych całkowitoliczbowych na płaszczyźnie.

ne ułamki $\frac{m}{n}$, $m, n \in \mathbb{Z}$. Przyjmijmy przy tym, że liczby całkowite reprezentujemy jako ułamki $\frac{m}{1}$ (w szczególności $0 = \frac{0}{1}$), a w przypadku ułamków ujemnych, minus znajduje się w liczniku. Przy tych ustaleniach funkcja $\Phi : \mathbb{Q} \to \mathbb{Z} \times \mathbb{Z}$ postaci

$$\Phi\left(\frac{m}{n}\right) \;=\; (m, n)$$

jest injekcją odwzorowującą liczby wymierne w punkty o współrzędnych całkowitoliczbowych na płaszczyźnie. Prócz tego odwzorowanie $\Psi : \mathbb{N} \to \mathbb{Q}$ zadane wzorem $\Psi(n) = \frac{n}{1}$ także jest injekcją. Oznacza to, że $\mathbb{N}$ jest co najwyżej tak liczne jak $\mathbb{Q}$, natomiast $\mathbb{Q}$ co najwyżej tak liczne jak $\mathbb{Z} \times \mathbb{Z}$. Mamy więc $\mathbb{N} \lesssim \mathbb{Q} \lesssim \mathbb{Z} \times \mathbb{Z}$ oraz

$\mathbb{N} \simeq \mathbb{Z} \times \mathbb{Z}$, co wykazaliśmy przed chwilą. Zatem na podstawie (1.3) wnioskujemy, że $\mathbb{Q} \simeq \mathbb{N}$.

Ten wynik jest prawdopodobnie jeszcze bardziej zaskakujący niż przeliczalność zbioru $\mathbb{Z}$, ponieważ w każdym — nieważne jak małym — przedziale znajduje się nieskończenie wiele różnych liczb wymiernych.

Udowodnimy teraz jeszcze jeden fakt o przeliczalności zbiorów powstających jako sumy zbiorów przeliczalnych. Pokażemy mianowicie, że suma przeliczalnej liczby zbiorów przeliczalnych,

$$S \;=\; \bigcup_{n=1}^{\infty} A_n\,, \qquad A_n \simeq \mathbb{N} \quad \forall\, n\,,$$

jest zbiorem przeliczalnym. Ponieważ każdy zbiór A_n jest przeliczalny, jego elementy można ustawić w ciąg

$$A_n \;=\; \{a_{n1}, a_{n2}, a_{n3}, \ldots\}\,,$$

gdzie pierwszy indeks n wskazuje do którego zbioru należą te elementy. Jeśli teraz wypiszemy nasze ciągi kolejno względem n w postaci nieskończonej tabeli,

$$\begin{array}{cccc} a_{11} & a_{12} & a_{13} & \cdots \\ a_{21} & a_{22} & a_{23} & \cdots \\ a_{31} & a_{32} & a_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

możemy systematycznie ponumerować wszystkie wypisane elementy, traktowane teraz jako elementy sumy S. Element a_{11} otrzymuje numer 1, a_{12} numer 2, z kolei a_{21} numer 3, a_{31} numer 4, a_{22} numer 5, a_{13} numer 6, itd. zgodnie ze wskazanym sposobem przechodzenia przez tabelę. W ten sposób wnioskujemy, że $S \simeq \mathbb{N}$.

Podamy teraz przykład zbioru nieprzeliczalnego. Jest nim zbiór liczb rzeczywistych $\mathbb{R}$. W dowodzie tego faktu posłużymy się tzw. *metodą przekątniową*, która okaże się użyteczna w naszej analizie własności języków formalnych w dalszej części skryptu.

Przykład 1.4 Ograniczymy się na początek do przedziału $(0, 1)$: każdą liczbę $x \in (0, 1)$ zapiszemy w postaci dziesiętnego ułamka z na ogół nieskończoną liczbą cyfr,

$$x \;=\; 0.\, x_1\, x_2\, x_3\, \ldots$$

Jeśli rozwinięcie dziesiętne x jest skończone, dopełniamy je nieskończoną liczbą zer. Zauważmy przy tym, że liczby z cyfrą '9' w okresie są identyczne z odpowiednimi liczbami, w których ostatnia cyfra różna od '9' jest powiększona o 1, np. $0.34999\ldots = 0.35 = 0.35000\ldots$ Wykluczamy więc liczby z '9' w okresie by uniknąć niejednoznacznej reprezentacji liczb z $(0, 1)$.

Przypuśćmy, że przedział $(0, 1)$ jest przeliczalny. Istnieje więc sposób ustawienia wszystkich liczb $0 < x < 1$ w ponumerowany ciąg, $\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots\}$. Wypiszmy

rozwinięcia dziesiętne kolejnych liczb $x^{(k)}$ w formie tabelarycznej

$$\begin{array}{lcccccc} x^{(1)} & = & 0\,. & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & \dots \\ x^{(2)} & = & 0\,. & x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & \dots \\ x^{(3)} & = & 0\,. & x_1^{(3)} & x_2^{(3)} & x_3^{(3)} & \dots \\ \vdots & & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

gdzie $x_i^{(k)} \in \{0, 1, \dots, 9\}$. Skonstruujemy obecnie liczbę $z \in (0, 1)$, która z pewnością nie znajduje się wśród elementów ciągu $x^{(k)}$, a tym samym założenie o uwzględnieniu w nim *wszystkich* bez wyjątku liczb $x \in (0, 1)$ musi być fałszywe. Tym samym $(0, 1)$ nie może być zbiorem przeliczalnym. Konstrukcja kolejnych cyfr rozwinięcia liczby $z = 0.\, z_1\, z_2\, z_3\, \dots$ jest następująca:

$$\text{Jeśli} \;\; x_k^{(k)} = 0 \;\; \text{wówczas} \;\; z_k = 1, \;\; \text{w przeciwnym razie} \;\; z_k = 0.$$

Widzimy, że z różni się od k-tej liczby $x^{(k)}$ na k-tym miejscu po kropce. Tym samym z różni się od wszystkich liczb ciągu $x^{(k)}$.

W końcu nieprzeliczalność zbioru R uzyskujemy pokazując istnienie bijekcji $f : (0, 1) \to \mathbb{R}$. Przykładem takiej funkcji jest $y = \tan \pi\left(x - \frac{1}{2}\right)$. $(0, 1)$ i $\mathbb{R}$ są więc równoliczne.

Metoda przekątniowa bierze swoją nazwę od sposobu, w jaki elementy z przekątnej tabeli wykorzystywane są w konstrukcji nowego obiektu, z gwarancją, że jest on różny od wszystkich występujących w tabeli. Oto inny ważny przykład zastosowania tej metody.

TWIERDZENIE 1.1 *Zbiór potęgowy zbioru przeliczalnego jest nieprzeliczalny.*

Dowód. Do dowodu twierdzenia wykorzystamy metodę przekątniową i kodowanie podzbiorów przez ciągi binarne. Niech więc $X = \{x_1, x_2, x_3, \dots\}$. Z każdym podzbiorem $U \in \mathcal{P}(X)$ wiążemy jednoznacznie nieskończony ciąg binarny $\{b_1, b_2, b_3, \dots\}$, w taki sposób, że

$$b_k \;=\; \begin{cases} 0 & \text{gdy} \;\; x_k \notin U \\ 1 & \text{gdy} \;\; x_k \in U\,. \end{cases}$$

Przypuśćmy teraz, że $\mathcal{P}(X)$ jest zbiorem przeliczalnym, to znaczy że istnieje sposób ponumerowania liczbami naturalnymi wszystkich bez wyjątku podzbiorów $U \subset X$. Odpowiada to identycznej numeracji ciągów binarnych reprezentujących podzbiory. Wypiszmy więc te ciągi kolejno w tabeli:

$$\begin{array}{cccc} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} & \dots \\ b_1^{(2)} & b_2^{(2)} & b_3^{(2)} & \dots \\ b_1^{(3)} & b_2^{(3)} & b_3^{(3)} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

Podobnie jak poprzednio, możemy teraz utworzyć ciąg $\{c_n\}$, który różni się od wszystkich uwzględnionych w tabeli. Ciąg ten reprezentuje więc podzbiór nieujęty w numeracji. Założenie o przeliczalności $P(X)$ musi być więc fałszywe. Mamy

$$c_k = \begin{cases} 0 & \text{gdy } b_k^{(k)} = 1 \\ 1 & \text{gdy } b_k^{(k)} = 0 \end{cases}$$

co kończy nasz dowód. □

Z twierdzeniem powyższym związana jest słynna *hipoteza continuum* sformułowana przez G. Cantora w II poł. XIX wieku. Stwierdza ona, że nie istnieją zbiory o mocy istotnie większej niż $\mathbb{N}$, a jednocześnie istotnie mniejszej od $\mathbb{R}$. Po około 100 latach, w roku 1963 amerykański matematyk P. Cohen udowodnił, że hipoteza continuum jest niezależna od aksjomatów tradycyjnej teorii mnogości. Oznacza to, że nie można na gruncie tej teorii udowodnić ani prawdziwości hipotezy Cantora ani też prawdziwości jej zaprzeczenia. Hipoteza continuum przyjmowana jest więc jako dodatkowy aksjomat przy formułowaniu podstaw matematyki.

I.3 Notacja asymptotyczna Bachmanna–Landaua

Niemieccy matematycy P. Bachmann i E. Landau na przełomie XIX i XX wieku wprowadzili użyteczną notację pozwalającą w prosty sposób charakteryzować klasy funkcji liczbowych o podobnym tempie wzrostu. Dziś notacja ta wykorzystywana jest intensywnie w teorii złożoności obliczeniowej przy porównywaniu efektywności różnych metod algorytmicznych rozwiązujących podobne problemy obliczeniowe. Stwierdzenie, iż pewna metoda obliczeniowa jest klasy $O(f(n))$ oznacza, że jeśli przetwarzane dane składają się z n obiektów, algorytm produkuje rozwiązanie po wykonaniu "około" $f(n)$ elementarnych kroków. Np. naiwne metody sortowania porządkują ciąg n liczb wykonując $O(n^2)$ porównań i przestawień, natomiast metody bardziej zaawansowane są w stanie wykonać to samo zadanie w liczbie kroków rzędu $O(n \log_2 n)$. Przekłada się to na szybszy czas działania metod o niższej złożoności. Przykładowo sortowanie stu tysięcy liczb z zastosowaniem najprostszych metod wymaga kilku miliardów operacji, podczas gdy bardziej zaawansowane algorytmy potrzebują około miliona kroków, a więc wykonają to samo zadanie kilka tysięcy razy szybciej. Ma to niebagatelne znaczenie w przypadku przetwarzania danych, którego wyniki oczekiwane są w rzeczywistym czasie: dla rozważanego wyżej przykładu sortowania różnica w czasie byłaby taka, jak między sekundą a godziną!

Zdefiniujemy obecnie szczegółowo symbole Bachmana-Landaua.

DEFINICJA 1.12 *Niech $f(n)$ i $g(n)$ będą dwoma funkcjami $f : \mathbb{N} \to \mathbb{R}$ takimi, że* $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$.

1) *Mówimy, że f jest klasy $O(g)$, jeśli istnieją $N \in \mathbb{N}$ oraz $M \in \mathbb{R}$ takie, że dla wszystkich $n > N$ spełniona jest nierówność $f(n) \leqslant M \cdot g(n)$.*
2) *Stwierdzenie, że f jest klasy $o(g)$ oznacza, że* $\lim \frac{f(n)}{g(n)} = 0$.
3) *Mówimy, że f jest klasy $\Omega(g)$, jeśli istnieją $N \in \mathbb{N}$ oraz $M \in \mathbb{R}$ takie, że dla wszystkich $n > N$ spełniona jest nierówność $f(n) \geqslant M \cdot g(n)$.*

4) Mówimy, że f jest klasy $\omega(g)$, jeśli $\lim \frac{g(n)}{f(n)} = 0$.

5) Określenie, że f jest klasy $\Theta(g)$ oznacza, że istnieją stałe $M_1, M_2 > 0$ oraz $N \in \mathbb{N}$, takie że $M_1 g(n) \leqslant f(n) \leqslant M_2 g(n)$ dla wszystkich $n > N$.

Sens tych definicji wysłowimy w bardziej przystępny sposób, mówiąc odpowiednio, że

1) $f(n)$ rośnie wraz z n co najwyżej tak szybko jak $g(n)$;
2) $f(n)$ rośnie wraz z n wolniej niż $g(n)$;
3) $f(n)$ rośnie wraz z n co najmniej tak szybko jak $g(n)$;
4) $f(n)$ rośnie wraz z n szybciej niż $g(n)$;
5) $f(n)$ rośnie wraz z n asymptotycznie w tym samym tempie co $g(n)$.

Nadużywając nieco notacji piszemy często $f \in O(g)$ lub $f = O(g)$ na oznaczenie, iż f jest klasy $O(g)$ i podobnie dla pozostałych symboli.

Zwróćmy uwagę, że choć stwierdzenie, że $f = O(g)$ jest mniej precyzyjne od $f = \Theta(g)$, tym niemniej to pierwsze jest bardziej praktyczne w zastosowaniu do algorytmów, których dokładna złożoność może być bardzo trudna do oszacowania. Jeśli bowiem pewna funkcja $f(n)$ jest np. klasy $O(n^2)$, to jest także, zgodnie z definicją, klasy $O(n^3)$, $O(n^{10})$ lub $O(2^n)$, zwykle jednak staramy się podać możliwie najdokładniejsze oszacowanie tempa wzrostu f. W przypadku mnożenia macierzy $n \times n$ złożoność bezpośredniej, znanej z algebry liniowej metody jest klasy $O(n^3)$, istnieją jednak metody lepsze, np. algorytm Strassena o złożoności ograniczonej przez $O(n^{2.807})$, lub obecnie najszybszy w tej klasie ulepszony algorytm Coppersmitha i Winograda o przybliżonej złożoności $O(n^{2.373})$. W tym wypadku jednak podanie *dokładnego* rzędu metody $\Theta(n^\gamma)$ jest zadaniem tyleż skomplikowanym co i niepraktycznym. W notacji $f = O(g)$ chodzi bowiem o to by oszacować tempo wzrostu skomplikowanej funkcji f przez możliwie proste, przemawiające do wyobraźni wyrażenie g, a więc np. potegowe $O(n^\gamma)$.

Przypomnijmy na koniec dwa powiązane ze sobą fakty, znane z analizy matematycznej, które łatwo wyrazić w notacji Bachmanna:

$$\log_a n = o(n^\gamma) \qquad \text{dla dowolnych} \qquad a > 1, \;\; \gamma > 0$$

oraz

$$n^\gamma = o(a^n) \qquad \text{dla dowolnych} \qquad \gamma > 0, \;\; a > 1\,.$$

I.4 Zadania do Rozdziału I

Zadanie 1

Udowodnić zachodzenie praw de Morgana dla rodzin zbiorów:

$$\left(\bigcup_{n=1}^{\infty} A_n\right)^c = \bigcap_{n=1}^{\infty} A_n^c\,,$$
$$\left(\bigcap_{n=1}^{\infty} A_n\right)^c = \bigcup_{n=1}^{\infty} A_n^c\,.$$

Zadanie 2

a) Wyrazić przekrój, różnicę i różnicę symetryczną zbiorów A i B przez operacje sumy i dopełnienia.

b) Wyrazić przekrój zbiorów A i B przez operację różnicy.

c) Wyrazić przekrój zbiorów A i B przez operacje sumy i różnicy symetrycznej.

Zadanie 3

Pokazać, że

a) $A \subseteq B$ wtedy i tylko wtedy, gdy $A \cup B = B$;

b) $A = B$ wtedy i tylko wtedy, gdy $A \oplus B = \emptyset$;

c) $A \cap B = \emptyset$ wtedy i tylko wtedy, gdy $A = (A \cup B) - B$;

d) A jest skończony wtedy i tylko wtedy, gdy $\mathcal{P}(A) = \mathcal{F}(A)$.

Zadanie 4

Niech x, y oznaczają liczby rzeczywiste, a m, n — liczby naturalne. Zbadaj, które z podanych relacji S są relacjami równoważności odp. w $\mathbb{R}$ lub $\mathbb{N}$. W przypadku pozytywnej odpowiedzi, opisz klasy abstrakcji względem S.

a) $S = \{(x, y) : \ x^2 = y^2\}$

b) $S = \{(x, y) : \ xy \geqslant 0\}$

c) $S = \{(x, y) : \ |x - y| < 1\}$

d) $S = \{(m, n) : \ mn$ jest parzyste $\}$

e) $S = \{(m, n) : \ mn$ jest nieparzyste $\}$

f) $S = \{(m, n) : \ 2m + 3n$ dzieli się przez $5\}$.

Zadanie 5

Relacja między punktami P i Q na płaszczyźnie określona jest jako $P \sim Q$ jeśli $|PO| = |QO|$, gdzie $|PO|$ oznacza odległość punktu P od ustalonego punktu O na tej płaszczyźnie. Udowodnij, że jest to relacja równoważności i opisz jej klasy abstrakcji.

Zadanie 6

W zbiorze $X = \{0, 1, 2, \ldots, 6\}$ określmy relację: $m \sim n$ jeśli $2m + 3n = 1 \pmod 7$. Zbudować graf tej relacji, a następnie jej tranzytywne domknięcie. Czy to domknięcie jest relacją równoważności? Jeśli tak, określić jej klasy abstrakcji.

Zadanie 7

W zbiorze $X = \{0, 1, 2, \ldots, 11\}$ określmy relację: $m \sim n$ jeśli $m^2 = n^2 \pmod{12}$. Sprawdzić, że jest to relacja równoważności i opisać jej klasy abstrakcji.

Zadanie 8

Obliczyć moc zbioru X jeśli

a) X jest zbiorem wszystkich funkcji $f : A \to B$, jeśli $|A| = m$ oraz $|B| = n$;

b) X jest zbiorem wszystkich injekcji $f : A \to B$, jeśli $|A| = m \leqslant n = |B|$;

c) X jest zbiorem wszystkich bijekcji $f : A \to B$, gdy $|A| = |B| = n$.

Zadanie 9

Liczby algebraiczne są to liczby rzeczywiste, będące pierwiastkami pewnego wielomianu o współczynnikach całkowitych. Liczbą algebraiczną jest np. $\sqrt{2}$ ponieważ jest rozwiązaniem równania wielomianowego $x^2 - 2 = 0$. Podobnie proporcja złotego podziału $\frac{1+\sqrt{5}}{2}$ jest liczbą algebraiczną jako pierwiastek równania $x^2 - x - 1 = 0$. W szczególności każda liczba wymierna $\frac{m}{n}$ jest algebraiczna jako pierwiastek równania $nx - m = 0$. Przykładem bardziej skomplikowanej liczby algebraicznej jest $\sqrt[3]{2 + 3\sqrt{5}}$ jako pierwiastek $t^6 - 4t^3 - 41 = 0$. Udowodnić, że zbiór liczb algebraicznych jest przeliczalny.

π i e są przykładem liczb nie-algebraicznych, zwanych przestępnymi. To one sprawiają, że zbiór liczb rzeczywistych jest nieprzeliczalny.

Zadanie 10

Pokazać, że zachodzą następujące relacje asymptotyczne:

a) $n^2 + 2n\sqrt{n} - 5\log n = O(n^2)$

b) $3^n = O(n!)$

c) $n^2 2^n = O(3^n)$

d) $n! = O(n^n)$

e) $1 + 2 + \cdots n = O(n^2)$

f) $1^2 + 2^2 + \cdots n^2 = O(n^3)$

g) $\log 2 + \log 3 + \cdots + \log n = O(n \log n)$

h) $\sqrt[n]{n!} = O(n)$.

Rozdział II

Języki formalne

Języki formalne pojawiają się wśród fundamentalnych pojęć logiki, matematyki i informatyki, a także teoretycznej lingwistyki. Historycznie pojęcie to użyte zostało po raz pierwszy w pismach niemieckiego matematyka i filozofa Gottloba Frege (1848–1920) już w II połowie XIX wieku. Teoria języków formalnych we współczesnym rozumieniu została sformułowana i usystematyzowana pod koniec lat 50-tych XX w. przez Noama Chomsky'ego, amerykańskiego lingwistę, logika i filozofa. Stworzona przez niego hierarchia języków formalnych będzie głównym przedmiotem naszego zainteresowania w niniejszym podręczniku.

II.1 Motywacje

W praktyce programistycznej często porównujemy między sobą różne metody rozwiązania tego samego problemu, oceniając je pod kątem czaso- lub pamięciochłonności. Za "lepsze" uważamy zwykle te algorytmy, które działają szybciej dla podobnych danych lub też angażują mniej pamięci operacyjnej komputera. Na przykład popularne metody sortowania operujące na tablicach porównują między sobą ich elementy i ewentualnie zamieniają je miejscami. I tak sortowanie metodą bąbelkową wykonuje około $n^2/2$ porównań oraz pewną, zależną od początkowego ułożenia ciągu, liczbę przestawień jego elementów. W najgorszym wypadku liczba przestawień także wynosi około $n^2/2$. Z kolei sortowanie stogowe wykonuje na tych samych danych około $n \log n$ porównań i przestawień, a więc gdy np. $n = 10^6$, działa ono mniej więcej 50000 razy szybciej. Gdy przetwarzamy duże zbiory danych, pytanie o czasochłonność zastosowanej metody staje się bardzo istotne.

Stawiamy także pytanie ogólniejsze, które dotyczy samego problemu sortowania, a nie jedynie jego konkretnych algorytmicznych rozwiązań: jaka jest minimalna liczba porównań i zamian elementów niezbędna do uporządkowania dowolnego ciągu o długości n? Staramy się tu ocenić stopień trudności samego zadania, inherentnie w nie wpisany, poprzez określenie najmniejszego niezbędnego nakładu pracy, który musi wykonać każda specyficzna metoda jego rozwiązania. Łatwo zauważyć, że sortowania n elementów nie można zrealizować w serii krótszej niż n operacji, podobnie jak nie można uporządkować rozdanych podczas gry w brydża kart bez odkrycia każdej z nich. Liczba n jest więc dolnym ograniczeniem na liczbę operacji w sortowaniu. Bardziej wnikliwa analiza pokazuje jednak, że wszystkie metody porządkowania ba-

zujące na porównywaniu elementów wymagają nie mniej niż $n \log n$ operacji. Dzięki temu oszacowaniu możemy wnioskować o optymalności sortowania stogowego.[1]

Podobnie możemy się zastanawiać nad złożonością pamięciową określonych zadań obliczeniowych. Jeśli np. rozważymy prosty problem wyznaczania średniej arytmetycznej n liczb, łatwo przekonamy się, że do jego realizacji nie jest wymagane przechowanie wszystkich tych liczb w pamięci podczas obliczeń. Wystarczy bowiem czytać z wejścia kolejne liczby i sumować je w pojedynczej zmiennej S oraz zliczać przy pomocy drugiej zmiennej n ile ich było. Obliczenie wyniku S/n angażuje więc jedynie 2 zmienne niezależnie od długości ciągu danych. Z kolei wyznaczenie mediany tego samego ciągu wymaga wcześniejszego posortowania jego elementów, a więc angażuje n komórek pamięci komputera. W tym sensie obliczanie mediany jest zadaniem bardziej kosztownym.

Wyznaczanie precyzyjnych oszacowań minimalnej liczby operacji lub objętości pamięci roboczej niezbędnych do rozwiązania określonego zadania jest na ogół trudne. Znajomość takich ograniczeń pozwala nam jednak porównywać między sobą zadania obliczeniowe pod względem stopnia ich złożoności, niezależnie od tego, że specyfika tych zadań może być bardzo różna. Na wspólnej skali "obliczeniowej trudności" możemy uszeregować w ten sposób zadania o tak różnej naturze, jak np. wspomniane sortowanie, rozwiązanie układu równań, wyznaczanie podzielników zadanej liczby, czy też poszukiwanie zamkniętej marszruty konika szachowego odwiedzającej każde pole szachownicy $n \times n$ dokładnie jeden raz.

Wspomniana przed chwilą klasyfikacja zadań obliczeniowych uzyskuje rygorystyczną i jednolitą reprezentację na gruncie teorii automatów i języków formalnych. Języki rozumiane są jako zbiory słów, czyli napisów zbudowanych z liter pewnego alfabetu. Z każdym językiem związany jest unikalny problem obliczeniowy, polegający na rozstrzygnięciu czy dane słowo jest elementem tego języka, czy też nie. Na przykład, jeśli język określony jest jako zbiór ciągów zawierających p jedynek, gdzie p jest liczbą pierwszą,

$$L \;=\; \{11,\, 111,\, 11111,\, 1111111,\, 11111111111,\, \ldots\}\,,$$

wówczas problem obliczeniowy reprezentowany przez ten język polega na stwierdzeniu czy długość danego słowa $11\ldots1$ nie posiada nietrywialnych podzielników. Skojarzony z językiem automat (algorytm) czyta ciąg wejściowy i po osiągnięciu jego końca sygnalizuje odpowiedź "tak" lub "nie". Rozstrzygnięcie czy słowo $\alpha \in L$ pod względem trudności jest więc równoważne problemowi badania czy dana liczba jest pierwsza.

Jak jednak reprezentować zadania polegające na wyznaczeniu wartości pewnej funkcji? Weźmy przykładowy problem znajdowania największego wspólnego podzielnika dwóch liczb, $k = \mathrm{NWP}(m, n)$. Definiujemy język

$$L = \{a^m b^n c^k \;:\; k = \mathrm{NWP}(m, n)\}\,,$$

gdzie przez a^m oznaczyliśmy skrótowo ciąg m liter a. Pytanie czy $a^m b^n c^k \in L$ rozstrzyga się przez sprawdzenie równości $k = \mathrm{NWP}(m, n)$. Ściśle rzecz biorąc zadanie

[1] Istnieją algorytmy sortowania nie wykonujące bezpośrednich porównań, które realizują zadanie w około n krokach, jednak nie są one w pełni uniwersalne, korzystają bowiem z pewnej dodatkowej wiedzy o rozkładzie i zakresie zmienności porządkowanych danych.

polegające na *sprawdzeniu* czy k jest największym wspólnym podzielnikiem liczb m i n jest nieco inne niż *obliczenie* takiego k. Jednak można udowodnić, że obydwa zadania posiadają dokładnie taki sam stopień trudności, to jest do ich rozwiązania trzeba zaangażować jednakowe zasoby obliczeniowe. W ten sposób język L reprezentuje problem wyznaczania NWP.

W kolejnych rozdziałach niniejszego skryptu zajmiemy się określeniem i badaniem różnych klas języków o coraz to wyższej złożoności. Języki tej samej klasy reprezentują wówczas zadania obliczeniowe o podobnym stopniu trudności. W ostatnim rozdziale będziemy w stanie wyznaczyć także granicę między problemami algorytmicznie rozwiązalnymi, a zadaniami nierozstrzygalnymi, które nie posiadają ogólnych algorytmicznych metod rozwiązania.

II.2 Alfabet i język nad alfabetem

Rozpocznijmy od wprowadzenia kilku podstawowych definicji.

DEFINICJA 2.1 *Alfabetem nazywamy dowolny lecz skończony zbiór. Elementy tego zbioru określamy mianem liter lub symboli. Słowa nad danym alfabetem są skończonymi ciągami liter.*

Przykłady alfabetów to np. $A = \{a, b, c\}$ lub $B = \{0, 1\}$. Przykładami słów są w szczególności $\alpha = aabac$ lub $\beta = 101010$. Z formalnego punktu widzenia słowa utożsamiać można z elementami odpowiednich potęg kartezjańskich alfabetu, np.

$$aabac \in A^5 \qquad \text{oraz} \qquad 101010 \in B^6 \,.$$

Dodatkowo przyjmujemy konwencję, że $A^0 = \{\lambda\}$, gdzie λ oznacza słowo puste, składające się z 0 liter. Wówczas

$$A^* \stackrel{\text{df}}{=} \bigcup_{n=0}^{\infty} A^n$$

jest zbiorem wszystkich słów nad alfabetem A. Zwróćmy uwagę, że jeśli $\alpha \in A^*$, oznacza to, że istnieje liczba n taka, że $\alpha \in A^n$, a więc A^* nie zawiera nieskończonych ciągów liter. Istnieje naturalny sposób uporządkowania słów w A^* pod warunkiem, że ustalimy porządek liter w alfabecie A. Wówczas porządek w zbiorze A^* odpowiada intuicyjnie rozumianemu uporządkowaniu alfabetycznemu słów. W matematyce uporządkowanie tego typu nazywamy leksykograficznym.

DEFINICJA 2.2 *Porządkiem leksykograficznym na A^* nazywamy relację $\alpha < \beta$ dla $\alpha = a_1 a_2 \ldots a_m$, $\beta = b_1 b_2 \ldots b_n$ określoną warunkiem: istnieje $1 \leqslant k \leqslant m$ takie, że $a_i = b_i$ dla $i = 1, \ldots, k-1$ oraz albo $a_k < b_k$ albo $k = m < n$.*

Konkatenacja jest operacją sklejania słów w zbiorze A^*, np. jeśli $\alpha = abc$, $\beta = bbaa$, wówczas ich konkatenacja $\alpha\beta = abcbbaa$, podobnie jak $\beta\alpha = bbaaabc$. Słowo puste λ pełni rolę elementu neutralnego konkatenacji (przez analogię do mnożenia liczb zwanego też "jednością"): dla dowolnego słowa α mamy

$$\alpha\lambda \;=\; \lambda\alpha \;=\; \alpha \,.$$

Piszemy także $\alpha^n = \alpha\alpha\ldots\alpha$ (n-krotnie) oraz $\alpha^0 = \lambda$. Zauważmy, że konkatenacja jest działaniem łącznym w A^*: $(\alpha\beta)\gamma = \alpha(\beta\gamma)$. Dlatego A^* jest względem konkatenacji *półgrupą* z jednością, czyli tzw. *monoidem*.

W dalszej części skryptu używać będziemy często następujących oznaczeń:

$$|\alpha| \;=\; \text{długość słowa } \alpha$$

oraz

$$|\alpha|_a \;=\; \text{liczba liter } a \text{ w słowie } \alpha\,.$$

DEFINICJA 2.3 *Językiem L nad alfabetem A nazywamy dowolny podzbiór $L \subseteq A^*$.*

Zauważmy, że o ile A^* jest zbiorem przeliczalnym, liczba różnych języków (za więc elementów zbioru potęgowego $\mathcal{P}(A^*)$) jest nieprzeliczalna. Przejdźmy do omówienia kilku przykładów języków.

PRZYKŁAD 2.1 Przyjmijmy, że $A = \{a, b\}$.

$$\begin{aligned}
L_1 &= \{ab^n :\ n \geqslant 0\} = \{a, ab, abb, abbb, \ldots\}\\
L_2 &= \{a^n b^n :\ n \geqslant 0\} = \{\lambda, ab, aabb, aaabbb, \ldots\}\\
L_3 &= \{\alpha \in A^* :\ |\alpha|_a = |\alpha|_b\} = \{\lambda, ab, ba, aabb, abab, abba, baab, baba, bbaa, \ldots\}\\
L_4 &= \{\alpha\alpha :\ \alpha \in A^*\} = \{\lambda, aa, bb, aaaa, abab, baba, bbbb, \ldots\}\\
L_5 &= \{a^n b^n a^n :\ n \geqslant 0\} = \{\lambda, aba, aabbaa, \ldots\}
\end{aligned}$$

Załóżmy, że automat, a więc np. program komputerowy, ma rozstrzygnąć czy dane słowo $\alpha \in A^*$ jest elementem języka L_i, czy też nie. Charakterystyczne dla automatu jest to, że będzie on analizował α litera po literze, od lewej do prawej strony, widząc w danym momencie tylko jedną — kolejną literę. To tak, jak gdybyśmy oglądali słowa α przez kartkę papieru z wyciętym otworem, odsłaniając w danym momencie tylko pojedynczą literę, jednocześnie przysłaniając pozostałe. Zupełnie inaczej do tego zadania podchodzi nasz mózg, oglądając słowa w całości i analizując np. powtórzenia dostrzeganych w nich wzorców.

Zwróćmy w tym kontekście uwagę na istotne różnice między językami np. L_1 a L_2 lub L_5. O ile w przypadku języka L_1, gdy oglądamy słowa α litera po literze, wystarczy zapamiętać fakt wystąpienia początkowej litery a, a następnie sprawdzić czy kolejne odsłaniane litery to wyłącznie b, o tyle w przypadku L_2 konieczne jest zapamiętanie ile liter a widzieliśmy, by następnie sprawdzić czy liczba liter b jest zgodna. Ponieważ n może być dowolnie duże, musimy tym samym dysponować nieograniczoną pamięcią na zapamiętanie tej liczby. W przypadku języka L_3 automat z prostym licznikiem mógłby dodawać do niego 1 przy napotkaniu kolejnej litery a lub odejmować 1 gdy kolejną literą jest b. Osiągnięcie 0 jako końcowego stanu licznika byłoby równoznaczne z zaakceptowaniem słowa α jako elementu L_3.

Zwykły licznik nie wystarczy do badania czy $\alpha \in L_4$. Słowa L_4 składają się zawsze z dwóch identycznych części, które należy w jakiś sposób porównać, jeśli jednak analizujemy je litera po literze, głównym problemem staje się poprawne zidentyfikowanie punktu podziału słowa na te 2 części. O ile automatom analizującym języki

1–3 wystarczy jednokrotny skan α od lewej do prawej i jednoczesne operacje na liczniku, tym razem konieczne jest wielokrotne przeglądanie słowa α lub równoważnie, możliwość cofania się czytnika liter w lewo.

Nasze obserwacje sugerują, że w obecnym przykładzie mamy do czynienia z językami, których analiza wymaga coraz to bardziej złożonych zasobów. Choć zapewne w tej chwili nie będzie to oczywiste, pod względem stopnia skomplikowania język L_5 jest bliższy L_4 niż pozornie podobnemu językowi L_2.

PRZYKŁAD 2.2 Język może być zbiorem skończonym, np. $L = \{aa, bb, abb, baa\}$. Należy odróżniać język jednoelementowy $L = \{\lambda\}$ od języka pustego $L = \emptyset$.

PRZYKŁAD 2.3 Oto kilka innych przykładowych języków.

$$\begin{aligned}
L_1 &= \{a^{n^2} : n \geqslant 0\} = \{\lambda, a, aaaa, aaaaaaaaa, \ldots\} \\
L_2 &= \{a^{n!} : n > 0\} = \{a, aa, aaaaaa, \ldots\} \\
L_3 &= \{a^p : p \text{ jest liczbą pierwszą}\} = \{aa, aaa, aaaaa, \ldots\} \\
L_4 &= \{a^x b^y c^z : x, y, z \in \mathbb{N} \quad \text{i} \quad x^2 + y^2 = z^2\} = \{a^3b^4c^5, a^5b^{12}c^{13}, a^6b^8c^{10}, \ldots\} \\
L_5 &= \{a^x b^y c^z a^n : x, y, z, n \in \mathbb{Z},\ n \geqslant 3 \quad \text{i} \quad x^n + y^n = z^n\}
\end{aligned}$$

Przykłady te ilustrują, że w postaci języka zakodować można rozmaite procedury obliczeniowe o dowolnym stopniu skomplikowania. To czy język L_5 jest pusty czy też nie, zależy od prawdziwości Wielkiego Twierdzenia Fermata, fascynującej zagadki matematycznej, która czekała na rozwiązanie ponad 350 lat: od roku 1637, gdy została sformułowana przez francuskiego matematyka-samouka Pierre'a de Fermat (był on z wykształcenia prawnikiem), do roku 1994, gdy angielski matematyk Andrew Wiles podał jej ostateczne rozwiązanie. Tak więc od około 20 lat wiemy, że $L_5 = \emptyset$.

II.3 Operacje na językach

Jako zbiory, języki podlegają operacjom mnogościowym: sumom, przekrojom, dopełnieniom, różnicom i różnicom symetrycznym. Powstają w ten sposób nowe języki.

Zdefiniowaną w poprzednim rozdziale operację konkatenacji słów można także określić na ich zbiorach:

$$XY = \{\alpha\beta : \alpha \in X \quad \text{i} \quad \beta \in Y\}$$

oraz $X^n = XX \ldots X = \{\alpha_1 \ldots \alpha_n : \alpha_i \in X\}$. Symbol X^* oznacza tym razem tzw. *domknięcie konkatenacyjne* zbioru X,

$$X^* = \bigcup_{n=0}^{\infty} X^n ,$$

a więc zbiór napisów, które można utworzyć sklejając ze sobą skończenie wiele słów (oraz ich kopii) z X. Idąc tym tropem możemy określić operację konkatenacji języków, $L = L_1L_2$ i domknięcie konkatenacyjne języka L^*. Wróćmy na chwilę do języków z Przykładu 2.1. Mamy np.

$$L_1L_2 = \{ab^m a^n b^n : m, n \geqslant 0\} .$$

Czytelnik zechce się przekonać, że o ile $L_4 \neq L_4L_4$, podobnie jak $L_4 \neq A^*A^* = A^*$, to np. $L_3 = L_3L_3$.

Operacje "dzielenia" słów są w pewnym sensie odwrotne do konkatenacji. Tym razem chodzi o usunięcie pewnego podsłowa z $\alpha \in A^*$, przeważnie jego końcowego lub początkowego fragmentu. *Prawostronnym ilorazem* języka L_1 przez L_2 nazywamy zbiór słów

$$L_1/L_2 \;=\; \{\alpha \in A^* :\; \alpha\beta \in L_1 \quad \text{dla pewnego} \quad \beta \in L_2\}\,.$$

Np. jeśli ponownie wziąć L_1 i L_2 z Przykładu 2.1 otrzymamy

$$L_1/L_2 \;=\; \{\lambda\} \qquad \text{oraz} \qquad L_2/L_1 \;=\; \{a^n :\; n \geqslant 0\}\,.$$

Jeśli zaś $L = \{b^n :\; n \geqslant 0\}$, wówczas

$$L_2/L \;=\; \{a^n b^m :\; n \geqslant m \geqslant 0\}\,.$$

Podobnie definiujemy *iloraz lewostronny* języka L_1 przez L_2

$$L_2\backslash L_1 \;=\; \{\beta \in A^* :\; \alpha\beta \in L_1 \quad \text{dla pewnego} \quad \alpha \in L_2\}\,.$$

Proponujemy czytelnikowi samodzielne wyznaczenie ilorazów $L_1\backslash L_2$ i $L_2\backslash L_1$ dla L_1, L_2 z Przykładu 2.1.

Jeśli w roli dzielnika L_2 w powyższych definicjach użyjemy języka pełnego A^*, otrzymamy operacje Head oraz Tail, a więc "głowę" i "ogon" dla danego języka L. Są to oczywiście zbiory dopuszczalnych początków i odp. zakończeń słów w L:

$$\mathrm{Head}(L) \;=\; \{\alpha \in A^* :\; \alpha\beta \in L \quad \text{dla pewnego} \quad \beta \in A^*\}$$

oraz

$$\mathrm{Tail}(L) \;=\; \{\beta \in A^* :\; \alpha\beta \in L \quad \text{dla pewnego} \quad \alpha \in A^*\}\,.$$

Jeśli zaś dzielnik L_2 jest językiem jednoelementowym, $L_2 = \{\beta\}$, otrzymujemy tzw. *prawo-* i *lewostronne pochodne* L w punkcie β,

$$\frac{dL}{d\beta^+} \;=\; \{\alpha \in A^* :\; \alpha\beta \in L\} \qquad \text{oraz} \qquad \frac{dL}{d\beta^-} \;=\; \{\alpha \in A^* :\; \beta\alpha \in L\}\,. \tag{2.1}$$

W teorii języków formalnych rozważa się ponadto wiele innych, specyficznych typów operacji nad słowami, które rozciąga się następnie na języki. Wymienimy tylko trzy najważniejsze. Jedną z nich jest tzw. *lustrzane odbicie* $\overleftarrow{L}$. Dla słów odbicie definiuje się następująco:

$$\begin{aligned} \overleftarrow{\lambda} &= \lambda \\ \overleftarrow{\alpha a} &= a\overleftarrow{\alpha} \qquad \text{dla} \quad a \in A, \quad \alpha \in A^*\,. \end{aligned}$$

Polega więc ono na odwróceniu porządku liter w słowie, na którym działa. Wówczas dla języka L mamy:

$$\overleftarrow{L} \;=\; \{\overleftarrow{\alpha} :\; \alpha \in L\}\,.$$

Drugą ważną operacją, którą opiszemy obecnie jest tzw. *homomorficzny obraz języka*. Jeśli A oraz B są alfabetami, wówczas funkcja $h : A \to B^*$ definiuje homomorfizm konkatenacyjny h^* między zbiorami A^* i B^*. Jeśli bowiem $\alpha = a_1 a_2 \ldots a_k \in A^*$, wówczas

$$h^*(\alpha) \stackrel{\text{df}}{=} h(a_1)h(a_2)\ldots h(a_k)$$

jest słowem nad B. Operacje takie znamy z codziennej praktyki: A może być zbiorem znaków drukarskich, a $B = \{0, 1\}$. Przykładem funkcji h jest wówczas kod ASCII, przyporządkowujący poszczególnym znakom 8-bitowe kody. Działanie homomorfizmu polega wówczas na zamianie tekstu na odpowiednią reprezentacje binarną. Jeśli więc L jest językiem nad alfabetem A oraz $h : A \to B^*$, homomorfizm h^* określa obraz L w zbiorze B^*, a więc język nad B,

$$h^*(L) \;=\; \{h^*(\alpha) :\; \alpha \in L\} \;\subset\; B^* \,.$$

Na ogół opuszczamy gwiazdkę przy symbolu homomorfizmu, pisząc po prostu $h(L)$, ponieważ nie prowadzi to do niejasności.

W literaturze spotkać można także tzw. operację podstawienia, która jest nieco ogólniejsza niż homomorfizm. Niech L będzie językiem nad alfabetem A. Ponadto niech dla każdego $a \in A$ określony będzie pewien język L_a nad alfabetem B_a (w ogólności zarówno alfabety B_a jak i języki L_a mogą być zupełnie różne). Tak więc mamy tu pewną funkcję $\chi : A \to \mathcal{L}$ o wartościach w klasie $\mathcal{L}$ zawierającej wszystkie języki L_a. Przez analogię z homomorfizmem napiszemy

$$\chi^*(L) \;=\; \bigcup_{\alpha\in L} \chi^*(\alpha) \;=\; \bigcup_{\alpha\in L} \Big\{\beta_1\beta_2\ldots\beta_k :\; \beta_i \in L_{a_i},\;\; \text{gdzie } a_1 a_2 \ldots a_k = \alpha\Big\}. \quad (2.2)$$

$\chi^*(L)$ jest więc sumą rozmaitych konkatenacji języków L_a tworzonych wg. wzorców $\alpha \in L$. Jak poprzednio, dla wygody opuszczamy często gwiazdkę, pisząc po prostu $\chi(L)$.

Zauważmy, że operacja podstawienia uogólnia wiele innych prostszych działań na językach. Gdy np. każdy z języków L_a jest jednoelementowy, $L_a = \{\beta_a\}$, nad wspólnym alfabetem B, podstawienie staje się zwykłym homomorfizmem. Gdy z kolei $L = \{ab\}$ oraz $\chi(a) = L_1$ i $\chi(b) = L_2$, podstawienie realizuje konkatenację L_1L_2, natomiast gdy $L = \{a, b\}$, podstawienie staje się sumą $L_1 \cup L_2$. W końcu dla $L = \{a\}^*$ obraz $\chi(L)$ jest równy L_1^*.

II.4 Metody definiowania języków

Wyrażenia regularne

Użytkownicy systemu Unix korzystają często z tzw. *wyrażeń regularnych* do systematycznego przeszukiwania dużych plików tekstowych w celu znalezienia wierszy zawierających określone słowa. Np. polecenie

```
grep '^From: .*umk\.pl' mybox
```

znajduje wszystkie wiersze pliku `mybox` rozpoczynające się słowem `'From: '`, po którym następuje dowolny ciąg znaków zakończony literami `'umk.pl'`. Wyrażenia

regularne są również bardzo przydatne podczas edycji plików tekstowych, gdy np. zależy nam za szybkiej, systematycznej zamianie pewnej grupy napisów innymi napisami w obrębie całego tekstu. Wyrażenia te pełnią rolę *wzorców* dopasowywanych do rzeczywistych ciągów znaków w przetwarzanym tekście. Używane są także systemach programowania np. w `C++` lub w Javie, podobnie jak wyżej do określania wzorców łańcuchów znakowych.

Istnieje wiele standardów określających składnię wyrażeń regularnych. Jedną z bardziej rozpowszechnionych postaci jest ta opisana w standardzie POSIX.[2] I tak np. wyrażenie

```
[ABC]xxx[0-9]+\.[^efg]
```

określa klasę napisów, które

— rozpoczynają się literą `'A'`, `'B'` lub `C`

— kolejne 3 litery to `'xxx'`

— dalej następuje niepusty ciąg cyfr dowolnej długości zakończony kropką

— po kropce nie występuje żadna z liter `'e'`, `'f'` ani `'g'`.

Nie będziemy w tym miejscu dłużej skupiać się na składni wyrażeń regularnych tego lub innego standardu, wystarczy jeśli zauważymy, że wyrażenia takie mogą być wygodną formą określania języków. Jeśli bowiem ω jest takim wyrażeniem, napiszemy

$$L(\omega) \;=\; \{\alpha \in A^* :\; \alpha \text{ pasuje do wzorca } \omega\}\,.$$

W istocie określimy niebawem wyrażenia regularne jako jedną z alternatywnych metod definicji pewnej klasy języków formalnych.

Gramatyki

Gramatyki stanowią poręczną formę definiowania poprawnych strukturalnie elementów danego języka. Choć pierwotnie jest to pojęcie z zakresu lingwistyki zajmującej się badaniem i opisem struktury języków naturalnych, jednak okazało się ono niezwykle przydatne w informatyce do definiowania składni języków programowania. Strukturalny opis gramatyczny języka programowania jest bardzo pomocny przy projektowaniu efektywnych algorytmów kompilacji, wykrywających w szerokim zakresie ewentualne błędy składniowe popełniane przez programistów.

Gramatyka zawsze składa się ze *skończonego* zbioru reguł, które — stosowane iteracyjnie — generują klasę poprawnych formuł danego języka. W opisie języków programowania stosowana jest często gramatyczna notacja BNF (Backus-Naur form), której przykład podajemy niżej.

Przykład 2.4 Przytoczymy fragment opisu BNF składni wyrażeń arytmetycznych. Składnia ta obowiązuje w większości języków programowania. W notacji Backusa-Naura używane są następujące symbole pomocnicze: nawiasy metajęzykowe “$\langle \ldots \rangle$”, symbol definicji “$::=$” oraz symbol alternatywy “$|$”. Wszystkie inne znaki oznaczają same siebie.

[2] Portable Operating System Interface for Unix

$$
\begin{array}{rcl}
\langle\,\text{wyrażenie}\,\rangle & ::= & \langle\,\text{wyrażenie1}\,\rangle \mid \langle\,\text{add}\,\rangle\langle\,\text{wyrażenie1}\,\rangle \\
\langle\,\text{wyrażenie1}\,\rangle & ::= & \langle\,\text{składnik}\,\rangle \mid \langle\,\text{składnik}\,\rangle\langle\,\text{add}\,\rangle\langle\,\text{wyrażenie1}\,\rangle \\
\langle\,\text{składnik}\,\rangle & ::= & \langle\,\text{czynnik}\,\rangle \mid \langle\,\text{czynnik}\,\rangle\langle\,\text{mul}\,\rangle\langle\,\text{składnik}\,\rangle \\
\langle\,\text{czynnik}\,\rangle & ::= & \langle\,\text{stała}\,\rangle \mid \langle\,\text{zmienna}\,\rangle \mid \Big(\langle\,\text{wyrażenie}\,\rangle\Big) \\
\langle\,\text{add}\,\rangle & ::= & + \mid - \\
\langle\,\text{mul}\,\rangle & ::= & * \mid /
\end{array}
$$

Lewa strona wraz z każdą z alternatywnych postaci po prawej stronie stanowi tu osobną regułę gramatyczną zwaną też *produkcją*. Symbol " | " pozwala na bardziej zwięzłe zapisanie kilku produkcji na raz w jednym wierszu. *Wyprowadzenie* konkretnego wyrażenia arytmetycznego z reguł składniowych polega na stopniowym zastępowaniu obiektów metajęzykowych innymi, zgodnie z zasadą, że lewa strona produkcji zamieniana jest przez formę po jej prawej stronie. I tak np.

$$
\begin{array}{rcl}
\langle\,\text{wyrażenie}\,\rangle & \to & \langle\,\text{wyrażenie1}\,\rangle \to \langle\,\text{składnik}\,\rangle \to \langle\,\text{czynnik}\,\rangle\langle\,\text{mul}\,\rangle\langle\,\text{składnik}\,\rangle \\
& \to & \langle\,\text{zmienna}\,\rangle\langle\,\text{mul}\,\rangle\langle\,\text{składnik}\,\rangle \to a * \langle\,\text{składnik}\,\rangle \\
& \to & a * \langle\,\text{czynnik}\,\rangle \to a * \Big(\langle\,\text{wyrażenie}\,\rangle\Big) \to a * \Big(\langle\,\text{wyrażenie1}\,\rangle\Big) \\
& \to & a * \Big(\langle\,\text{składnik}\,\rangle\langle\,\text{add}\,\rangle\langle\,\text{wyrażenie1}\,\rangle\Big) \\
& \to & a * \Big(\langle\,\text{czynnik}\,\rangle\langle\,\text{add}\,\rangle\langle\,\text{składnik}\,\rangle\Big) \\
& \to & a * \Big(\langle\,\text{zmienna}\,\rangle\langle\,\text{add}\,\rangle\langle\,\text{składnik}\,\rangle\Big) \to a * \Big(b + \langle\,\text{składnik}\,\rangle\Big) \\
& \to & a * \Big(b + \langle\,\text{czynnik}\,\rangle\Big) \to a * \Big(b + \langle\,\text{stała}\,\rangle\Big) \to a * \Big(b + 1\Big)
\end{array}
$$

Dla kompletności opisu należałoby podać jeszcze precyzyjne określenia klas metajęzykowych $\langle\,\text{stała}\,\rangle$ i $\langle\,\text{zmienna}\,\rangle$. Pominiemy to jednak w tym miejscu z uwagi na oczywistość tych definicji.

W przypadku gramatyk języków formalnych stosujemy bardziej zwięzłą notację, rozróżniając zawczasu, które symbole pełnią rolę liter *terminalnych*, tj. elementów alfabetu definiowanego języka, a które pełnią rolę pomocniczą. Te ostatnie nazywamy symbolami *nieterminalnymi* lub krótko *nieterminalami* gramatyki. Zwykle przyjmuje się konwencję, że terminale oznaczanie małymi literami, a nieterminale dużymi. Strzałka "$\to$" separuje lewe i prawe strony produkcji, a symbol " | " jak poprzednio pozwala zwięźle opisywać alternatywne postaci prawych stron produkcji dla tej samej lewej strony. Końcowe (terminalne) słowa języka generowanego przez gramatykę nie mogą zawierać już żadnych symboli nieterminalnych. Kolejne etapy wyprowadzenia separujemy znakiem "$\Rightarrow$" dla odróżnienia od zwykłej strzałki używanej w produkcjach.

Przykład 2.5 Gramatyka z produkcjami

$$S \;\to\; \lambda \mid aSb$$

generuje język $L_2 = \{a^n b^n : \; n \geqslant 0\}$ z Przykładu 2.1. Istotnie, iterowane użycie drugiej produkcji wyprowadza

$$S \;\Rightarrow\; aSb \;\Rightarrow\; aaSbb \;\Rightarrow\; \ldots \;\Rightarrow a^n S b^n\,,$$

natomiast produkcja pusta $S \to \lambda$ usuwa ostatecznie symbol nieterminalny S.

Z kolei język L_3 z tegoż przykładu opisuje gramatyka G

$$S \;\to\; \lambda \mid aSb \mid bSa \mid SS\,.$$

Rzeczywiście, każde słowo terminalne generowane przez reguły tej gramatyki zawiera równe liczby liter a i b, bo prawe strony wszystkich produkcji mają te własność. Pozostaje jedynie przekonać się, że każde słowo $\alpha \in \{a,b\}^*$ o jednakowej liczbie liter a i b może być wyprowadzone w tej gramatyce. Użyjmy indukcji względem długości słowa $|\alpha| = 2n$. Gdy $n = 1$ $\alpha = ab$ lub $\alpha = ba$, a każde z nich posiada wyprowadzenie w G. Załóżmy więc, że wszystkie słowa długości nie większej niż $2n$ o równej liczbie a i b dają się wygenerować w G i rozważmy takie α, dla którego $|\alpha| = 2n+2$. Jeśli jego pierwszą literą jest a a ostatnią b (lub odwrotnie), wówczas $\alpha = a\beta b$, $|\beta| = 2n$, $|\beta|_a = |\beta|_b$, a więc zgodnie z założeniem indukcyjnym β może być wyprowadzone z symbolu S w G. Mamy więc

$$S \;\Rightarrow\; aSb \;\Rightarrow\; \ldots \;\Rightarrow a\beta b \;=\; \alpha\,.$$

Jeśli zaś pierwszą i ostatnią literą w α jest a (dla b argument jest analogiczny), oznaczmy kolejne litery $\alpha = x_1x_2\ldots x_{2n-1}x_{2n}$. Wówczas w podsłowie $x_1x_2\ldots x_{2n-1}$ o 1 przeważa litera b. Określmy funkcję

$$\Delta(k) \;=\; \text{liczba liter } a \;-\; \text{liczba liter } b \quad \text{w podsłowie } x_1x_2\ldots x_k\,. \tag{2.3}$$

Mamy więc $\Delta(1) = 1$ oraz $\Delta(2n-1) = -1$. Zatem istnieje co najmniej jeden taki indeks $1 < k < 2n-1$, dla którego $\Delta(k) = 0$. Jeśli podzielimy słowo α w tym punkcie oznaczając $\alpha_1 = x_1\ldots x_k$ oraz $\alpha_2 = x_{k+1}\ldots x_{2n+2}$, każde z tych podsłów zawiera po równo liter a i b i każde z nich jest nie dłuższe niż $2n$. A więc można obydwa te słowa wyprowadzić z symbolu nieterminalnego S. Ostatecznie

$$S \;\Rightarrow\; SS \;\Rightarrow\; \ldots \;\Rightarrow\; \alpha_1\alpha_2 \;=\; \alpha\,.$$

PRZYKŁAD 2.6 Gramatyka generująca język $L_5 = \{a^nb^na^n :\; n \geqslant 0\}$ jest bardziej skomplikowana:

$$\begin{aligned} S &\to \lambda \mid PX \\ P &\to aAb \mid aPAb \\ Ab &\to bA \\ AX &\to a \\ Aa &\to aa \end{aligned}$$

Zwróćmy uwagę, że o ile we wcześniejszych przykładach lewe strony produkcji stanowił pojedynczy symbol nieterminalny, obecnie mogą to być grupy liter. Polecamy czytelnikowi wyprowadzenie słów *aba*, *aabbaa*, oraz *aaabbbaaa* jako samodzielne ćwiczenie.

Podamy na koniec formalne definicje gramatyki i generowanego przez nią języka w ujęciu Chomsky'ego.

DEFINICJA 2.4 *Gramatyką w ogólnej, nieograniczonej postaci nazywamy czwórkę obiektów*

$$G \;=\; (A, V, S, \Pi)\,,$$

gdzie:

- A *jest alfabetem definiowanego przez* G *języka zwanym też alfabetem terminalnym,*
- V *jest alfabetem symboli pomocniczych, zwanych też symbolami nieterminalnymi, przy czym* $A \cap V = \emptyset$*,*
- S *jest wyróżnionym nieterminalnym symbolem startowym*
- Π *jest skończonym zbiorem reguł, zwanych produkcjami gramatyki, postaci*

$$\gamma \;\to\; \eta\,, \qquad \textit{gdzie} \quad \gamma, \eta \in (A \cup V)^*\,,$$

przy czym γ *zawiera obowiązkowo co najmniej jeden symbol nieterminalny.*

Operacyjny sens pojęcia gramatyki zilustrowaliśmy już w Przykładach 2.4–2.6. Bardziej precyzyjnie, gramatyka G indukuje tzw. *relację wyprowadzenia* na zbiorze napisów $(A \cup V)^*$:

$$\alpha \Rightarrow_G \beta \qquad \text{jeśli} \qquad \alpha = \alpha_1\gamma\alpha_2\,, \quad \beta = \alpha_1\eta\alpha_2 \quad \text{i} \quad \gamma \to \eta \in \Pi\,. \tag{2.4}$$

Innymi słowy, $\alpha \Rightarrow_G \beta$, jeśli β powstaje z α przez zastąpienie w niej pewnego fragmentu identycznego z lewą stroną jednej z produkcji przez prawą stronę tej produkcji. Zauważmy, że w definicji produkcji wymagamy, aby jej lewa strona zawierała co najmniej jeden symbol nieterminalny. Chodzi o to, by łańcuchy wyprowadzeń kończyły się nieodwołalnie, bez możliwości kontynuacji na napisach terminalnych $\beta \in A^*$.

By zwięźle zapisywać ciągi wyprowadzeń indukowane przez gramatyki, bez konieczności pedantycznego wypisywania wszystkich bezpośrednich kroków, wprowadzimy tranzytywne domknięcie relacji "$\Rightarrow_G$": $\alpha \Rightarrow_G^* \beta$ jeśli $\alpha \Rightarrow_G \beta$ lub istnieją napisy $\alpha_1, \ldots \alpha_k$, dla których

$$\alpha \Rightarrow_G \alpha_1 \Rightarrow_G \cdots \Rightarrow_G \alpha_k \Rightarrow_G \beta\,.$$

Z kolei, przy pomocy relacji $\Rightarrow_G^*$ zdefiniujemy język opisywany przez gramatykę.

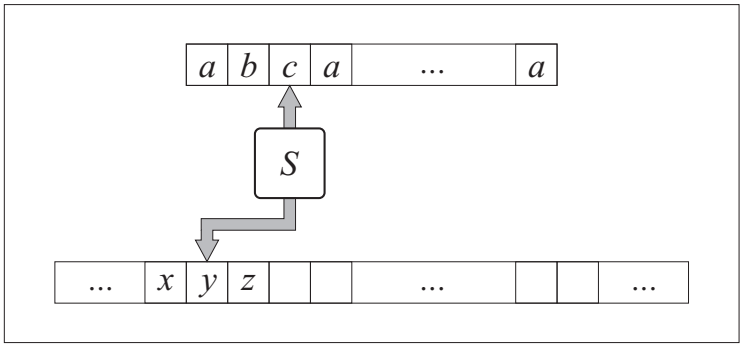
DEFINICJA 2.5 *Językiem generowanym przez gramatykę* $G = (A, V, S, \Pi)$ *nazywamy zbiór*

$$L(G) \;=\; \{\alpha \in A^* :\; S \Rightarrow_G^* \alpha\}\,.$$

Słowa pośrednie α_i w wyprowadzeniach zaczynających się od S, zawierające wciąż symbole nieterminalne, nazywamy *formami zdaniowymi* gramatyki G.

Automaty

Automaty stanowią uproszczone, abstrakcyjne modele cyfrowych komputerów. Redukują ich operacje do minimalnego zbioru elementarnych "kroków" i definiują jednocześnie klasę problemów obliczeniowych, które można rozwiązać bazując wyłącznie



Rys. 2.1: Schemat automatu.

na tych elementarnych mikrooperacjach. W kontekście teorii języków formalnych automaty stanowią reprezentację języków $L \subset A^*$: analizują poddawane im słowa $\alpha \in A^*$ i w skończonej liczbie kroków rozstrzygają czy $\alpha \in L$, czy też $\alpha \notin L$.

Ogólny schemat automatu przedstawiono na Rys. 2.1. Dane wejściowe (analizowane słowo $\alpha \in A$) umieszczane są na górnej taśmie. Dolna taśma, jeśli występuje, służy jako pamięć robocza i posiada swój własny alfabet. Jednostka sterująca wyposażona jest w głowice pozwalające skanować pojedyncze znaki na obu taśmach. W każdej chwili automat znajduje się w jednym ze swoich stanów wewnętrznych. Zbiór tych stanów, podobnie jak alfabety, jest *skończony*. Działanie automatu opisane jest przez skończony zbiór ruchów: zależnie od aktualnego stanu i skanowanych na obu taśmach liter, automat wybiera nowy stan, przesuwa górną głowicę o 1 pozycję w prawo, opcjonalnie zapisuje na dolnej taśmie nowy symbol w miejscu skanowanej poprzednio litery, przesuwając przy tym dolną głowicę o jedną pozycję w lewo lub w prawo. Działanie rozpoczyna się zawsze w wyróżnionym stanie startowym S_0 z górną głowicą ustawioną na 1 literze wejściowej. Jeśli po przeskanowaniu ostatniej litery z górnej taśmy automat znajdzie się w jednym z tzw. stanów *finalnych*, słowo α zostaje zaakceptowane jako element danego języka, a każdym innym wypadku — odrzucone. Końcowa zawartość taśmy roboczej jest na ogół nieistotna.

Jeśli alfabety wejściowy i roboczy oznaczymy odpowiednio przez A i B, a zbiór stanów automatu przez Σ, wówczas ruchy automatu opisać można jako *funkcję przejścia*

$$\pi : \Sigma \times A \times B \longrightarrow \Sigma \times B \times \{L, R\}\,.$$

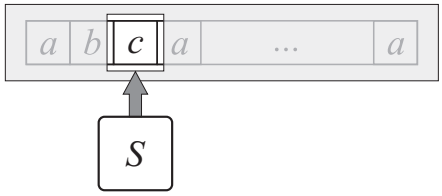
Np. $\pi(S, a, x) = (S', y, R)$ oznacza, że po przeczytaniu litery a z górnej taśmy w stanie S, gdy na taśmie roboczej skanowany jest element x, automat przechodzi do stanu S', zapisuje y w miejsce x oraz przesuwa dolną głowicę o 1 pozycję w prawo. Ruch górnej głowicy czytającej dane jest przy tym automatyczny.

Na ogół π jest odwzorowaniem częściowym, to znaczy, że nie musi być ono określone dla wszystkich konfiguracji (stan, litera, litera). W przypadku przejścia do takiej konfiguracji automat zatrzyma się nie mogąc wykonać kolejnego ruchu. Jest to równoważne z odrzuceniem słowa wejściowego α, bo górna głowica z pewnością nie osiągnęła jeszcze jego końca. Podkreślmy jeszcze raz, że przeskanowanie ostatniej litery jest warunkiem koniecznym zaakceptowania danego słowa.

Jeśli zamiast funkcji π automat opisany jest przez *relację* przejścia, mówimy że jego działanie jest *niedeterministyczne*. Wówczas "wartość" $\pi(S, a, x)$ nie jest okre-

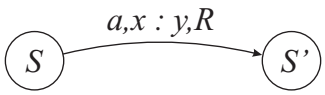
ślona jednoznacznie — jest to zbiór możliwych ruchów automatu i rozumie się, że wybór kolejnej konfiguracji jest niedeterministyczny. Mówimy w tej sytuacji, że słowo α jest akceptowane przez niedeterministyczny automat, jeśli *istnieje* sekwencja jego ruchów prowadząca do konfiguracji końcowej w jednym ze stanów finalnych, z głowicą górną na końcu taśmy wejściowej.

Podkreślmy jedną bardzo istotną cechę działania automatu: w danej chwili ma on dostęp tylko do *pojedynczej* litery na każdej z używanych taśm i aktualnego swojego stanu. Relacja przejścia $\pi(S, a, x)$ jest tak właśnie określona. Mamy tendencję zapominać o tym, gdy "ludzkim okiem" patrzymy na słowa danego języka, np. *aaabbb*. Nasz mózg jest w stanie ogarniać i analizować jednocześnie całe słowa lub przynajmniej duże ich fragmenty, rozpoznając z łatwością rządzący nimi wzorzec. Bardziej adekwatna byłaby więc wizualizacja działania automatu, w której jego taśmy wyposażone są w przysłony ukazujące tylko pojedyncze litery:



Wówczas można lepiej wyobrazić sobie na czym polega trudność w rozpoznaniu słów języka $a^n b^n$ w stosunku np. do wzorca $a^n b^m$, gdzie n i m są niezależne. O ile w tym drugim wypadku wystarczy przełączyć odpowiednio stan automatu po napotkaniu pierwszej litery b, by odrzucić te słowa, w których po b pojawiają się jeszcze litery a (taśma robocza jest do tego niepotrzebna), o tyle w pierwszym trzeba przechować informację o liczbie napotkanych liter a, by zweryfikować jej zgodność z liczbą liter b. Ponieważ jednak liczba stanów automatu jest skończona i określona z góry, zatem przy pomocy samych stanów nie można zapamiętać dowolnie dużego n, trzeba posłużyć się taśmą roboczą jako pamięcią. Zachęcamy czytelnika do wykonania prostych eksperymentów z oglądaniem napisu litera po literze przez otwór wycięty w zasłaniającej go kartce papieru. Zmusimy wówczas nasz mózg do trybu pracy właściwego automatom.

Istnieje także wygodna reprezentacja automatów przy pomocy grafów. Grafem $G(M)$ automatu M nazywamy graf skierowany, którego wierzchołki identyfikujemy ze stanami automatu, natomiast krawędzie odpowiadają jego ruchom, przy czym etykiety przypisane krawędziom jednoznacznie identyfikują te ruchy. Np. przejściu $\pi(S, a, x) = (S', y, R)$ odpowiada etykietowana krawędź



Każda etykieta krawędzi $(a, x : y, R)$ składa się z części wejściowej (a, x) i wyjściowej (y, R). W przypadku automatu niedeterministycznego w grafie pojawią się

wielokrotne krawędzie wychodzące z jednego wierzchołka o identycznych etykietach wejściowych. Każda z nich reprezentuje alternatywny ruch automatu w danej konfiguracji.

II.5 Klasyfikacja języków formalnych

Zarówno automaty jak i gramatyki stanowią dogodny punkt wyjścia do zdefiniowania pewnych specyficznych klas języków formalnych. Nakładając ograniczenia na postać produkcji gramatyk lub na sposób działania automatów zawężamy klasy języków, które mogą być przez nie generowane bądź akceptowane. W przypadku automatów ograniczenia dotyczą sposobu użycia pamięci roboczej.

Najprostszą formą automatów są tzw. *automaty skończone* pozbawione w ogóle pamięci roboczej. Jedyną formą pamięci są w ich przypadku stany, które, w ograniczonym zakresie, mogą przechowywać informację o wcześniej przeczytanych literach napisu wejściowego. Ograniczenie to wynika z faktu, że liczba stanów automatu jest skończona i dana raz na zawsze. Jeśli więc napis wejściowy jest dostatecznie długi, automat nie może przechować informacji o wszystkich jego literach. Przykładem języka, który może być poprawnie rozpoznany przez automat skończony jest język L_1 z Przykładu 2.1, natomiast język L_2 wykracza poza tę klasę. Jak wspomnieliśmy wcześniej, automat rozpoznający ten język musi przechować w pamięci informację o liczbie przeczytanych liter a, by następnie sprawdzić jej zgodność z liczbą liter b. Wystarczy więc wybrać słowo $a^n b^n$ o n większym niż liczba stanów skończonego automatu, by wyczerpać jego zasoby pamięciowe. Innymi słowy dla dostatecznie dużych n automat tej klasy nie będzie w stanie rozpoznać różnicy między słowami np. $a^n b^n$ oraz $a^n b^{n-1}$.

Automaty ze stosem są kolejną kategorią maszyn, których możliwości rozpoznawania języków są większe niż w przypadku automatów skończonych. Posiadają one potencjalnie nieskończoną pamięć roboczą, która jednak dostępna jest wyłącznie jako stos: zapis i odczyt odbywają się na szczycie stosu. Dostęp do informacji zapisanej wcześniej, a więc głębiej w stosie, wymaga bezpowrotnego usunięcia ze stosu danych zapisanych później. Okazuje się, że o ile taki automat jest w stanie poprawnie rozpoznać wszystkie słowa języka L_2, nie wykona podobnego zadania dla języka $L_5 = \{a^n b^n a^n : \ n \geqslant 0\}$. Chodzi o to, że podczas skanowania liter b zawartość stosu przechowująca informację o liczbie początkowych liter a zostanie odczytana i zabraknie jej dla porównania liczby końcowych a.

Jeszcze bardziej ogólną postacią automatów są tzw. maszyny *liniowo ograniczone*. W tym przypadku pamięć robocza może być zapisywana i odczytywana w dowolnej kolejności, jednak jej dostępny obszar ograniczony jest przez długość napisu wejściowego. Zarówno wspomniany przed chwilą język L_5, jak i wszelkie języki podobne do L_1–L_3 z Przykładu 2.3 mogą być rozpoznawane przez automaty liniowo ograniczone.

Automaty bez ograniczeń co do sposobu wykorzystania taśmy roboczej stanowią klasę tzw. *maszyn Turinga*. W dalszych rozdziałach opiszemy je szczegółowo przy pomocy nieco prostszego, lecz całkowicie równoważnego modelu. Maszyny Turinga, zgodnie z tzw. *hipotezą Churcha*, są najbardziej ogólnymi modelami "efektywnie wykonalnych procesów obliczeniowych" czyli algorytmów lub równoważnie — "funkcji

obliczalnych". Klasę języków formalnych odpowiadającą maszynom Turinga stanowią tzw. języki *rekurencyjnie przeliczalne*, o których można powiedzieć jedynie to, że dla każdego z nich istnieje algorytm (a więc maszyna Turinga) generujący sekwencyjnie wszystkie jego słowa. Jest to możliwie najsłabsza, lecz wciąż *konstruktywna* charakteryzacja języka jako zbioru określonych ciągów liter: wzorce pozwalające generować wszystkie te ciągi można zapisać za pomocą skończonego zbioru reguł gramatycznych lub kroków maszyny Turinga. Dzięki iteracji tych reguł otrzymujemy kolejne słowa języka.

Jak wspomnieliśmy poprzednio, rodzina wszystkich języków nad danym alfabetem jest zbiorem potęgowym $\mathcal{P}(A^*)$, a zatem — wobec przeliczalności A^* — nieprzeliczalnym. Przekonamy się później, że rodzina wszystkich maszyn Turinga jest zbiorem przeliczalnym. Stąd wniosek, że znakomita większość języków $L \subset A^*$ nie posiada w ogóle generujących je algorytmów, a więc z pozoru proste pytanie czy dane słowo $\alpha \in L$ na ogół nie może być rozstrzygnięte algorytmicznie.

Opisaliśmy tu szkicowo 4 klasy języków formalnych, odpowiadające hierarchii automatów: od najprostszych automatów skończonych do najbardziej ogólnych maszyn Turinga. Klasy te zostały scharakteryzowane przez Noama Chomsky'ego przez wprowadzenie klasyfikacji generujących je gramatyk. W literaturze klasy języków formalnych Chomsky'ego oznaczane są symbolami $\mathcal{L}_i$, $i = 3, 2, 1, 0$ i nazywane odpowiednio językami *regularnymi*, *bezkontekstowymi*, *kontekstowymi* i *rekursywnie przeliczalnymi*. Nazwy te pochodzą od typów generujących je gramatyk. Podstawowym rezultatem teorii języków formalnych jest hierarchizacja

$$\mathcal{L}_3 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_0 .$$

W dalszej części skryptu zajmiemy się równoważną charakteryzacją poszczególnych klas języków formalnych i szczegółowym omówieniem ich własności.

II.6 Zadania do Rozdziału II

Zadanie 1

Dla języków L_i zdefiniowanych w Przykładzie 2.1, wyznaczyć:

a) L_1^c i L_2^c

b) L_1^*, L_2^* i L_3^*

c) $L_3 \cap L_4$

d) L_3/L_2, $L_1 \backslash L_2$, $L_2 \backslash L_4$ i $L_3 \backslash L_5$

e) pochodne lewo- i prawostronne $\frac{dL_i}{d\alpha}$ dla $i = 1, \ldots, 5$, przyjmując $\alpha = a^k$, $a^k b^l$, b^k, $b^k a^l$, $k, l \geqslant 0$

f) Head(L_i) oraz Tail(L_i) dla $i = 1 \ldots, 5$

g) $h(L_i)$ dla $i = 1, 2, 5$, gdzie h jest homomorfizmem takim, że $h(a) = 010$, $h(b) = 101$.

Zadanie 2

Wykazać prawdziwość lub fałszywość następujących tożsamości dla języków:

a) $(L_1 \cup L_2)^* = L_1^* \cup L_2^*$
b) $(L_1 \cap L_2)^* = L_1^* \cap L_2^*$
c) $(L_1 L_2)^* = L_1^* L_2^*$
d) $(L^*)^* = L^*$
e) $\overleftarrow{(L_1 \cup L_2)} = \overleftarrow{L_1} \cup \overleftarrow{L_2}$
f) $\overleftarrow{(L_1 \cap L_2)} = \overleftarrow{L_1} \cap \overleftarrow{L_2}$
g) $\overleftarrow{L_1 L_2} = \overleftarrow{L_2}\,\overleftarrow{L_1}$
h) $\overleftarrow{L^*} = \left(\overleftarrow{L}\right)^*$
i) $(L^c)^* = (L^*)^c$
j) $\overleftarrow{(L^c)} = \left(\overleftarrow{L}\right)^c$
k) $h(L_1 \cup L_2) = h(L_1) \cup h(L_2)$, gdzie h jest homomorfizmem
l) $h(L_1 \cap L_2) = h(L_1) \cap h(L_2)$
m) $h(L_1 L_2) = h(L_1) h(L_2)$
n) $h(L^*) = \Big(h(L)\Big)^*$
o) $h(L^c) = \Big(h(L)\Big)^c$
p) $h\Big(\overleftarrow{L}\Big) = \overleftarrow{(h(L))}$.

Zadanie 3

Zakładając, że języki L_1 i L_2 są generowane odpowiednio przez gramatyki $G_i = (A, V_i, S_i, \Pi_i)$, $i = 1, 2$, gdzie dla uproszczenia przyjmujemy, że $V_1 \cap V_2 = \emptyset$, podać przepis na utworzenie gramatyki G generującej język

a) $L = L_1 \cup L_2$
b) $L = L_1 L_2$
c) $L = L_1^*$
d) $L = \overleftarrow{L_1}$
e) $L = h(L_1)$ dla zadanego homomorfizmu h.

Zadanie 4

Znaleźć gramatyki dla następujących języków nad alfabetem $A = \{a, b\}$:

a) $L = \{a^m b^n : \ m, n \geqslant 1\}$
b) $L = \{a^n b^{n+2} : \ n \geqslant 0\}$
c) $L = \{a^m b^{m+n} a^n : \ m, n \geqslant 0\}$
d) $L = \{\omega : \ |\omega|_a > |\omega|_b\}$
e) $L = \{\omega : \ |\omega|_a = 2|\omega|_b\}$

Rozdział III

Języki regularne

Rozdział niniejszy poświęcony jest charakteryzacji języków regularnych, strukturalnie najprostszych klas napisów. Z algorytmicznego punktu widzenia języki regularne są również najprostsze — dla dowolnego słowa $\alpha \in A^*$ rozstrzygnięcie czy α jest czy też nie jest elementem języka regularnego L wymaga wykonania nie więcej niż $|\alpha|$ operacji. Innymi słowy koszt obliczeniowy jest w tym przypadku minimalny, to jest dokładnie taki, jak koszt wczytania słowa α litera po literze. Złożoność rozpoznawania słów α języka L jest więc klasy $O(n)$, gdzie $n = |\alpha|$.

Wymagania pamięciowe w rozpoznawaniu języków regularnych są również minimalne: liczba wykorzystywanych "komórek" lub — w terminologii automatowej — "stanów" pamięci jest skończona, ustalona dla danego języka i *niezależna* od długości analizowanego słowa. Mówimy, że złożoność pamięciowa rozpoznawania języków regularnych jest klasy $O(1)$.

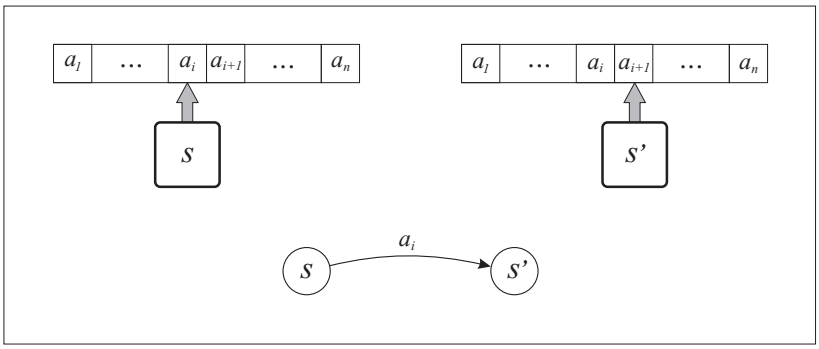
Języki regularne można opisać trzema równoważnymi metodami: przez automaty skończone, przy pomocy wyrażeń regularnych oraz przy użyciu formalnych gramatyk specjalnego, prostego typu.

III.1 Automaty skończone

Pierwszy sposób charakteryzacji języków regularnych polega na ich opisie przez skończone automaty. Automaty te nie posiadają taśmy roboczej, a jedynie wejściową, którą mogą pasywnie odczytywać bez możliwości cofania głowicy. Stąd też jedyna dostępna pamięć reprezentowana jest przez stany automatu. Jak wspomnieliśmy w podrozdziale II.4, liczba stanów automatu jest z definicji skończona, a więc jest ustalona dla języka akceptowanego przez automat, nie ma zaś związku z długością analizowanego napisu wejściowego α.

1 Skończone automaty deterministyczne

Pierwszym typem automatów, którymi zajmiemy się w tym rozdziale są skończone automaty deterministyczne, w skrócie SAD. Są to urządzenia, których zadaniem jest rozstrzygnięcie czy zapisany na taśmie wejściowej napis α jest "poprawny" czy też nie.



Rys. 3.1: Działanie skończonego automatu deterministycznego opisane przez funkcję przejścia $s' = \pi(s, a_i)$.

DEFINICJA 3.1 *Skończony automat deterministyczny dany jest przez pięć obiektów*

$$M \;=\; \Big(A, \Sigma, s_0, S_F, \pi\Big)\,,$$

gdzie:

A	*jest alfabetem*
Σ	*jest zbiorem stanów automatu,* $\sigma = \{s_0, s_1, \ldots, s_q\}$
$s_0 \in \Sigma$	*jest wyróżnionym stanem startowym*
$S_F \subseteq \Sigma$	*jest zbiorem stanów końcowych (akceptujących)*
π	*jest częściową funkcją przejścia,* $\pi : \Sigma \times A \nrightarrow \Sigma$.

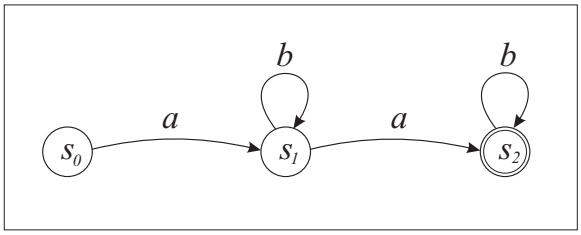
Działanie SAD opiszemy następująco. Na początku automat znajduje się w stanie startowym s_0 z głowicą ustawioną na pierwszej literze $a_1 \in A$ zapisanego na taśmie wejściowej napisu $\alpha = a_1 a_2 \ldots a_n$. Wartością funkcji przejścia $\pi(s_0, a_1)$ jest stan $s' \in \Sigma$, do którego automat przechodzi, przesuwając jednocześnie głowicę o jedną pozycję w prawo, do następnej litery na taśmie a_2. Proces ten powtarza się dalej cyklicznie: obliczany jest kolejny stan automatu $s'' = \pi(s', a_2)$, a głowica przesuwa się do kolejnej litery. W chwili gdy głowica znajdzie się za ostatnią literą a_n, ostatni osiągnięty stan automatu s wskazuje czy słowo α zostało zakwalifikowane jako poprawny element języka L: jeśli $s \in S_F$, wówczas $\alpha \in L$, w przeciwnym wypadku α nie jest poprawnym słowem języka L.

Możliwe jest także zatrzymanie automatu przed osiągnięciem końca napisu α z powodu nieokreślonej wartości funkcji przejścia dla danej konfiguracji "stan–litera" (przypomnijmy, że π jest z definicji określona jedynie *częściowo*). Wówczas słowo α jest odrzucane jako niepoprawne.

Rysunek 3.1 ilustruje poglądowo ruch automatu w sytuacji, gdy $\pi(s, a_i) = s'$, oraz reprezentację grafową takiego przejścia.

Działanie automatu M wygodnie jest opisywać za pomocą generowanej przez niego tzw. relacji przejścia $\Rightarrow_M$, analogicznej do relacji wyprowadzenia w gramatyce (2.4). Określamy ją na zbiorze napisów $(A \cup \Sigma)^*$ następująco:

$$a_1 \ldots a_{i-1} s a_i a_{i+1} \ldots a_n \quad \Rightarrow_M \quad a_1 \ldots a_{i-1} a_i s' a_{i+1} \ldots a_n$$



Rys. 3.2: SAD akceptujący język z Przykładu 3.1.

jeśli $s' = \pi(s, a_i)$. Zauważmy, że pozycja wskaźnika stanu s w napisie określa jednoznacznie aktualną pozycję głowicy automatu: czyta ona literę następującą po s.

Dla poprawnego słowa $\alpha \in L$ mamy więc

$$s_0 a_1 \dots a_n \quad \Rightarrow_M \quad \dots \quad \Rightarrow_M \quad a_1 \dots a_n s\,,$$

przy czym $s \in S_F$. Oznaczając przez $\Rightarrow_M^*$ tranzytywne domknięcie relacji przejścia zapiszemy krótko

$$s_0 a_1 \dots a_n \;\Rightarrow_M^*\; a_1 \dots a_n s\,.$$

W praktyce, gdy nie prowadzi to do niejasności, pomijamy indeks M pisząc po prostu $\Rightarrow$ i odpowiednio $\Rightarrow^*$.

Możemy teraz formalnie opisać język akceptowany przez dany SAD.

DEFINICJA 3.2 *Językiem akceptowanym przez skończony automat deterministyczny* $M = (A, \Sigma, s_0, S_F, \pi)$ *jest zbiór napisów*

$$L(M) \;=\; \left\{\alpha \in A^* :\; s_0\alpha \;\Rightarrow_M^*\; \alpha s, \quad \text{gdzie} \quad s \in S_F\right\}.$$

Języki akceptowane przez skończone automaty deterministyczne noszą nazwę języków regularnych, a ich klasa oznaczana jest symbolem $\mathcal{L}_{\mathrm{Reg}}$ *lub* $\mathcal{L}_3$ *w klasyfikacji Chomsky'ego.*

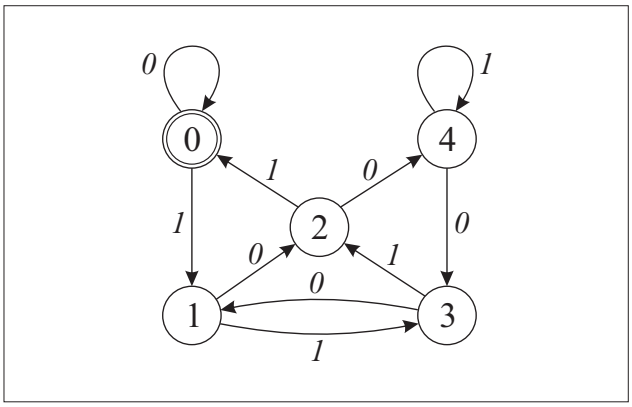
Przykład 3.1 Rys. 3.2 przedstawia SAD rozpoznający język $L = \{ab^m ab^n : m, n \geqslant 0\}$. Zwyczajowo stany końcowe w reprezentacji grafowej automatu oznaczamy podwójną linią.

Przykład 3.2 Zajmiemy się obecnie przykładem języka o bardziej złożonej strukturze. Niech L będzie zbiorem liczb naturalnych podzielnych przez 5 zapisanych w postaci binarnej, a więc

$$L \;=\; \{0, 101, 1010, 1111, 10100, \dots\}\,.$$

Na początek skonstruujemy SAD analizujący ciągi zero-jedynkowe, które traktowane jako liczby w notacji binarnej są podzielne przez 5, a następnie wyeliminujemy pewne jego wady związane z niejednoznacznością takiego zapisu liczb.

Zacznijmy od spostrzeżenia, że jeśli skanujemy ciąg binarny od lewej do prawej strony, pojawienie się 0 jako kolejnej cyfry oznacza, że liczbę utworzoną dotąd z wcześniejszych cyfr należy pomnożyć przez 2, natomiast gdy kolejną skanowaną



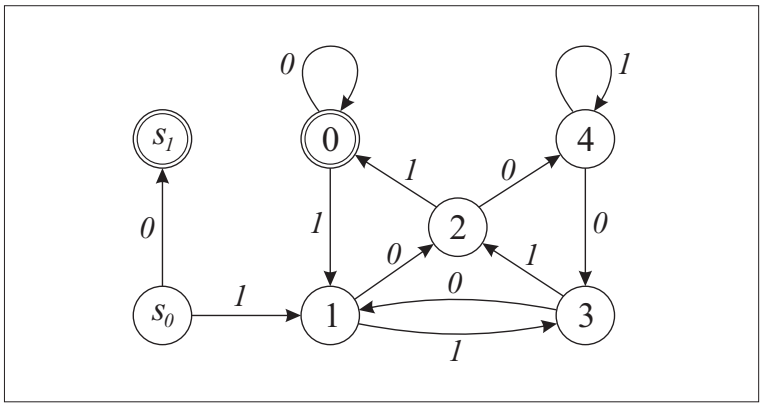
Rys. 3.3: Wstępna postać automatu badającego podzielność liczb w postaci binarnej przez 5.

cyfrą jest 1, dotychczasowy wynik należy pomnożyć przez 2 i zwiększyć o 1. Np. jeśli wczytany dotąd ciąg to 110, a więc liczba 6, pojawienie się następnego zera, 1100, to liczba $6 \times 2 = 12$, natomiast pojawienie się jako następnej cyfry jedynki, 1101, daje liczbę $6 \times 2 + 1 = 13$. Oczywiście przed przeczytaniem pierwszej cyfry domyślną wartością liczbową ciągu (pustego!) jest 0.

Oznaczmy stany automatu liczbami $0, \ldots, 4$ — możliwymi wartościami reszty z dzielenia przez 5. Jeśli więc na pewnym etapie wczytywania ciągu otrzymaliśmy liczbę, która w dzieleniu przez 5 daje resztę r, automat powinien znajdować się w tym momencie w stanie r. Pojawienie się jako kolejnego znaku cyfry 0 podwaja liczbę, a więc podwaja też resztę $r \to 2r \pmod 5$. Np. jeśli wczytana do tej pory wartość to 7 (reszta 2), dodanie kolejnego zera prowadzi do wartości 14 (reszta $2 \times 2 = 4$), podczas gdy np. dla wartości 18 (reszta 3) dodanie kolejnego zera daje liczbę 36 (reszta $3 \times 2 = 6 = 1 \pmod 5$). Na tej samej zasadzie dopisanie kolejnej jedynki zmienia resztę na $r' = 2r + 1 \pmod 5$. W ten sposób opisujemy wszystkie poprawne przejścia między stanami automatu. Rysunek 3.3 przedstawia graf kompletnego SAD. Zwróćmy uwagę, że stanem początkowym, a zarazem końcowym jest "0", ponieważ automat ma akceptować wyłącznie liczby podzielne przez 5.

Pozostaje nam wyeliminować niewielkie wady skonstruowanego automatu. Fakt, że "0" jest stanem początkowym i jednocześnie końcowym powoduje, że automat akceptuje napis pusty λ traktując go jako liczbę 0. Ponadto napisami akceptowanymi jako poprawne są np. 000, 0101, 00001010 itd., a więc liczby binarne ze zbędnymi nieznaczącymi zerami. By wyeliminować pierwszą wadę, należy oddzielić stan początkowy od końcowego. Uzyskamy to dodając nowy stan startowy s_0 oraz przejścia $\pi(s_0, 0) = 0$ i $\pi(s_0, 1) = 1$. Pozbycie się wiodących nieznaczących zer osiągnąć można dodając jeszcze jeden stan automatu s_1, który przechwytuje wyjątek w postaci pojawienia się zera jako pierwszej cyfry, a więc $\pi(s_0, 0,) = s_1$. Jest to stan końcowy, gdyż pojedyncze 0 jest poprawną liczbą podzielną przez 5. Ponieważ jednak w automacie nie dodajemy dalszych przejść ze stanu s_1, żaden inny napis rozpoczynający się cyfrą 0 nie będzie zaakceptowany, p. Rys. 3.4.

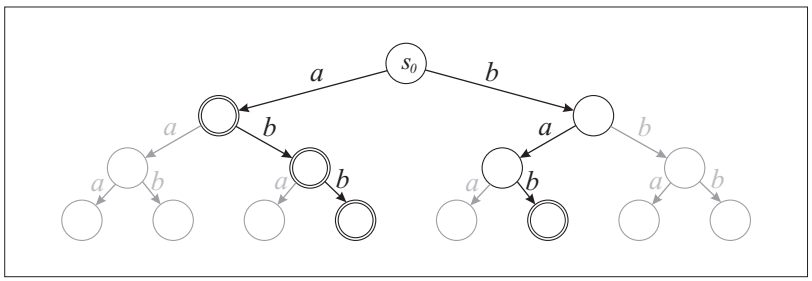
Przekonamy się obecnie, że każdy język skończony jest językiem regularnym. Niech więc $A = \{a_1, a_2, \ldots, a_k\}$ będzie alfabetem oraz niech $L = \{\alpha_1, \ldots, \alpha_r\} \subset A^*$.



Rys. 3.4: Kompletny automat rozpoznający liczby binarne podzielne przez 5.

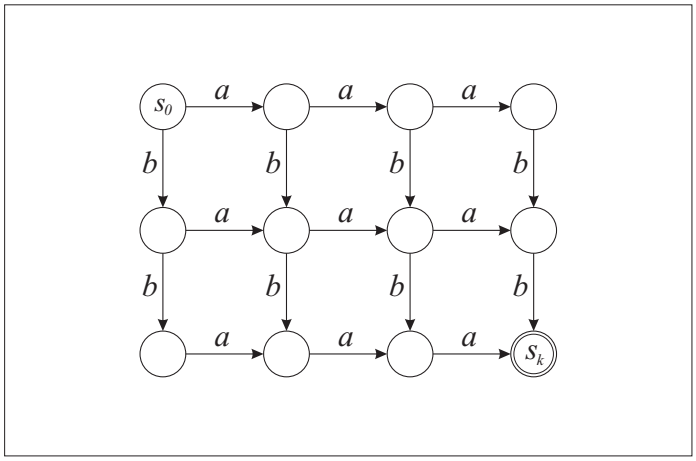
Załóżmy przy tym, że najdłuższe ze słów w L składa się z m liter. Ogólna metoda konstrukcji grafu SAD dla takiego języka polega na zbudowaniu go jako skierowanego drzewa o korzeniu s_0 i m piętrach (korzeń traktujemy jako piętro zerowe). Z każdego wierzchołka wychodzi k krawędzi etykietowanych literami $a_1, \ldots, a_k$ do wierzchołków kolejnego piętra. Węzłom drzewa nadajmy kolejno unikalne etykiety $s_1, s_2, \ldots, s_n$. Następnie dla słowa $\alpha_1 = a_{i_1} a_{i_2} \ldots a_{i_p}$ odnajdujemy w drzewie ścieżkę z korzenia wzdłuż krawędzi z etykietami kolejno a_{i_1}, a_{i_2}, itd. aż do węzła będącego końcem ostatniej krawędzi a_{i_p}. Ponieważ drzewo ma $m \geqslant p$ poziomów, ścieżka taka na pewno w nim istnieje. Oznaczmy ostatni wierzchołek tej ścieżki s_j jako stan końcowy automatu. Podobnie postępujemy z kolejnymi słowami α_i języka L. Można w końcu usunąć z drzewa wszystkie fragmenty ścieżek (krawędzie i węzły) nie prowadzące do stanów końcowych.

Konstrukcja powyższa jest na tyle oczywista, że podarujemy sobie osobny dowód faktu, iż rzeczywiście $L = L(M)$ dla zbudowanego automatu M. Zamiast tego posłużymy się przykładem języka $L = \{a, ab, abb, bab\}$. Graf odpowiedniego automatu skonstruowany wyżej opisaną metoda przedstawiono na poniższym rysunku.



Rys. 3.5: Automat w postaci drzewa dla skończonego języka $L = \{a, ab, abb, bab\}$. Szarym kolorem oznaczyliśmy krawędzie i stany, które mogą być usunięte z ostatecznej wersji grafu.

Zauważmy na koniec, że mogą istnieć także inne, prostsze automaty niż te powstające z ogólnej konstrukcji (tj. zawierające mniejszą liczbę stanów), akceptujące ten sam język.



Rys. 3.6: Automat dla języka z Przykładu 3.3.

PRZYKŁAD 3.3 Jako konkretny przykład rozważmy język L nad alfabetem $A = \{a, b\}$ składający się ze słów zawierających dokładnie trzy litery a i dwie b. Uniwersalna konstrukcja produkowałaby drzewo binarne o 5 piętrach, a więc o liczbie węzłów równej $2^6 - 1 = 63$. Ponieważ liczba słów w L wynosi $\binom{5}{2} = 10$, jedynie 10 węzłów 5 piętra oznaczylibyśmy jako stany końcowe automatu. Alternatywnie ten sam język akceptowany jest przez automat z Rys. 3.6, w którym utożsamiliśmy ze sobą wszystkie uzyskane wyżej stany końcowe, ponieważ kolejność w jakiej pojawiają się litery a i b jest w naszym przykładzie nieistotna.

Implementacja SAD

By zaimplementować skończony automat deterministyczny, wygodnie jest rozszerzyć najpierw funkcję przejścia π do odwzorowania $\hat{\pi}$ określonego na całym zbiorze $\Sigma \times A$, a więc do funkcji totalnej. Można to zawsze osiągnąć przez dodanie do oryginalnego zbioru Σ nowego stanu *pułapkowego* s_p oraz uzupełnienia π według schematu:

— jeśli $\pi(s, a)$ nie jest określone, kładziemy $\hat{\pi}(s, a) = s_p$;

— dodajemy przejścia $\hat{\pi}(s_p, a) = s_p$ dla wszystkich liter $a \in A$.

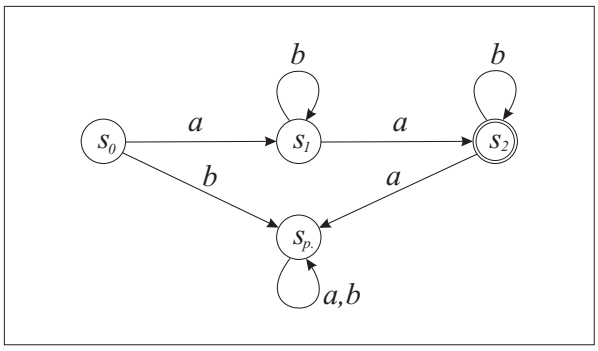
Nowy automat przechodzi więc do stanu pułapkowego (i pozostaje w nim do wyczerpania liter napisu wejściowego) w każdej sytuacji, w której poprzednio dalszy ruch był nieokreślony i stary automat przerywał działanie. Rys. 3.7 przedstawia automat rozważany w Przykładzie 3.1 po dodaniu stanu pułapkowego.

Implementacja SAD wykorzystuje tablicę do przechowania funkcji przejścia:

```
var P: array[stan][litera] of stan;
```

Mamy więc `P[s][a]` = s' jeśli $\pi(s, a) = s'$. Zmienna `s` przechowuje aktualny stan automatu, a kolejne symbole napisu wejściowego odczytywane są z pliku `input`. Sam algorytm jest następujący:

```
s := s0;
while not eof(input) do begin
```



Rys. 3.7: Automat dla języka z Przykładu 3.1 po rozszerzeniu π do totalnej funkcji. s_p jest stanem pułapkowym.

```
        a := getnextchar(input);
        s := P[s][a];
    end;
    if s ∈ S_F then writeln('tak') else writeln('nie');
```

Zauważmy na koniec, że pozostawienie opisu automatu w postaci częściowej funkcji przejścia π prowadziłoby do niewielkiego skomplikowania warunku sterującego pętlą "while" (można np. przyjąć, że nieokreśloność $\pi(s,a)$ kodowana byłaby jako `P[s][a] = -1`) i rozpoznania właściwej przyczyny przerwania tej pętli w warunku "if" po niej następującym.

2 Skończone automaty niedeterministyczne

W wersji niedeterministycznej działanie skończonego automatu jest nieco bardziej skomplikowane, a przede wszystkim mniej intuicyjne. Podstawowa różnica polega na tym, że w danej konfiguracji "stan–litera" może istnieć więcej niż jeden dopuszczalny ruch. Funkcja przejścia przestaje być jednoznaczna i zamienia się w ogólną relację, np. $\pi(s,a) = \{s_1, s_2, s_3\}$, co oznacza, że w stanie s z wejściową literą a możliwy jest ruch do dowolnego ze stanów s_1, s_2 lub s_3. Automat wybiera kolejny stan w sposób przypadkowy, a więc niedeterministycznie.

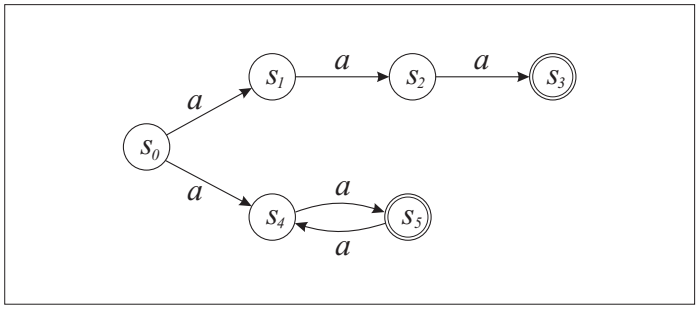
Zasadne jest pytanie w jakim celu wprowadza się takie automaty. Okazuje się, że dla wielu problemów znacznie łatwiej znaleźć można rozwiązanie w postaci niedeterministycznej. Oto prosty przykład takiej sytuacji.

Przykład 3.4 Rozważmy język $L = \{a^n : \ n > 0 \ \text{ i jest parzyste lub } \ n = 3\}$. Obok prostej reguły parzystości, wartość $n = 3$ jest tu wyjątkowa. Rozwiązanie niedeterministyczne przedstawia Rys. 3.8. Istnieje także nieskomplikowane rozwiązanie deterministyczne, jednak omówimy je w następnym podrozdziale.

Przejdźmy z kolei do formalnych definicji.

DEFINICJA 3.3 *Skończony automat niedeterministyczny, w skrócie SAN, zadany jest przez pięć obiektów*

$$M = \left(A, \Sigma, s_0, S_F, \pi\right),$$



Rys. 3.8: Automat niedeterministyczny dla języka z Przykładu 3.4. Źródłem niedeterminizmu są tu 2 alternatywne przejścia ze stanu s_0.

gdzie A, Σ, s_0 i S_F są identyczne z odpowiednimi obiektami w definicji automatu deterministycznego, natomiast π jest relacją przejścia

$$\pi : \Sigma \times (A \cup \{\lambda\}) \to \mathcal{P}(\Sigma)\,.$$

Jak wspomnieliśmy wyżej, relacja przejścia (zapisana wyżej jako funkcja o wartościach w zbiorze potęgowym $\mathcal{P}(\Sigma)$) określa do których stanów alternatywnie przechodzi automat z danej konfiguracji "stan–litera". Dodatkowo automat niedeterministyczny może wykonywać "puste" ruchy polegające na tym, że zmieniany jest jego stan bez badania aktualnej litery i bez ruchu głowicy. Zapisujemy ten rodzaj przejść jako np. $\pi(s, \lambda) = \{s', s''\}$. Symbol λ pojawia się także jako etykieta krawędzi w grafowej reprezentacji SAN.

Zbiór pusty $\emptyset$ jest pełnoprawnym elementem zbioru potęgowego $\mathcal{P}(\Sigma)$, dlatego też zapis $\pi(s, a) = \emptyset$ jest jak najbardziej uprawniony. Oznacza on, że ruch automatu w konfiguracji (s, a) nie jest określony.

Relacja $\Rightarrow_M$ i jej tranzytywne domknięcie $\Rightarrow^*_M$ określone są na $(A \cup \Sigma)^*$ tak samo jak w przypadku automatów deterministycznych. Dodatkowo dla pustych przejść $s' \in \pi(s, \lambda)$ mamy

$$a_1 \ldots a_{i-1} s a_i \ldots a_n \quad \Rightarrow_M \quad a_1 \ldots a_{i-1} s' a_i \ldots a_n$$

niezależnie od wartości a_i.

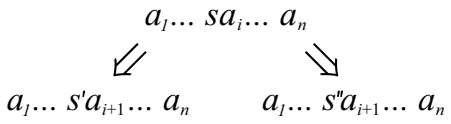
DEFINICJA 3.4 *Językiem $L(M)$ akceptowanym przez skończony automat niedeterministyczny $M = (A, \Sigma, s_0, S_F, \pi)$ jest zbiór napisów określony następująco:*

$$L(M) \;=\; \Big\{\alpha \in A^* :\;\; s_0\alpha \;\Rightarrow^*_M\; \alpha s \quad \textit{dla co najmniej jednego} \;\; s \in S_F\Big\}\,.$$

Tak więc $\alpha \in L(M)$ jeśli istnieje sekwencja legalnych ruchów automatu rozpoczynająca się w stanie s_0 i osiągająca jeden ze stanów końcowych $s \in S_F$ po przeczytaniu ostatniej litery α.

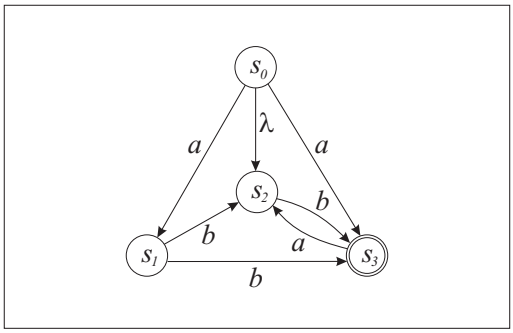
Posługując się relacją przejścia $\Rightarrow$ dla danego napisu wejściowego, wszystkie warianty działania automatu niedeterministycznego można zobrazować w postaci drzewa, którego wierzchołki reprezentują konfiguracje $a_1 \ldots s a_i \ldots a_n$, a krawędzie odpowiednie alternatywne przejścia. Jeśli np. $\pi(s, a_i) = \{s', s''\}$, wówczas w strukturze

drzewa zobaczymy



Wśród liści tego drzewa (a więc konfiguracji, z których nie można wykonać następnego ruchu) znajdują się konfiguracje akceptujące $a_1 \ldots a_n s$, gdzie $s \in S_F$. Historia ruchów automatu niedeterministycznego odpowiada pojedynczej ścieżce w takim drzewie, od korzenia do któregoś z liści. W tym obrazie, słowo α jest elementem języka $L(M)$ jeśli w drzewie przejść o korzeniu $s_0\alpha$ istnieje przynajmniej jedna ścieżka kończąca się konfiguracją akceptującą αs, $s \in S_F$.

Można łatwo wyobrazić sobie *deterministyczny* proces realizujący to samo zadanie co automat niedeterministyczny na podstawie jego relacji przejścia. Dla zadanego napisu wejściowego α proces taki musiałby prowadzić ekstensywne przeszukiwanie drzewa przejść (np. przeszukiwanie w głąb z powrotami) aż do znalezienia konfiguracji akceptującej w jednym z liści. Gdyby takiej konfiguracji nie znaleziono, proces odrzuciłby słowo α jako nienależące do $L(M)$, zauważmy jednak, że byłoby to możliwe dopiero po kompletnym przeszukaniu całego drzewa. Obserwacja ta podpowiada, że automat niedeterministyczny może być zastąpiony równoważnym[1] automatem deterministycznym, jednak za cenę pewnego skomplikowania jego struktury. Ogólną konstrukcję równoważnych automatów deterministycznych omówimy w następnym rozdziale.



Rys. 3.9: Automat niedeterministyczny z Przykładu 3.5.

Przykład 3.5 Przeanalizujemy działanie automatu M z Rys. 3.9. Przejście w pierwszym kroku do stanu s_3 prowadzi do akceptacji wyłącznie napisów postaci $a(ab)^n$ dla dowolnego $n \geqslant 0$, gdzie zapis $(ab)^n$ oznacza n powtórzeń napisu ab, a więc $\lambda, ab, abab, \ldots$. Jeśli w pierwszym kroku wykonamy puste przejście do stanu s_2, jedyne słowa akceptowane w dalszym przebiegu będą miały postać $b(ab)^n$ dla $n \geqslant 0$. W końcu początkowe przejście z s_0 do s_1 prowadzi do akceptacji przez automat słów postaci $ab(ab)^n$ oraz $abb(ab)^n$, $n \geqslant 0$. Język $L(M)$ można zwięźle opisać jako

[1] Równoważność automatów M i M' rozumiemy tu jako identyczność akceptowanych przez nie języków, $L(M) = L(M')$.

konkatenację L_1L_2 prostych języków $L_1 = \{a, b, ab, abb\}$ oraz $L_2 = \{(ab)^n : n \geqslant 0\}$. Proponujemy czytelnikowi samodzielne narysowanie odpowiednich, kompletnych drzew przejść automatu dla napisów wejściowych *abbab* oraz *ababb*.

3 Równoważność skończonych automatów deterministycznych i niedeterministycznych

Przekonamy się obecnie, że skończone automaty deterministyczne i niedeterministyczne opisują w istocie tę samą klasę języków. Zauważmy, że każdy automat deterministyczny jest specjalnym przypadkiem automatu niedeterministycznego, ponieważ funkcja przejścia π, częściowa bądź nie, jest po prostu specjalnym przypadkiem relacji przejścia. Automaty niedeterministyczne są więc ogólniejszą konstrukcją i dla udowodnienia równoważności obu ich typów należy podać metodę przekształcenia automatu niedeterministycznego w deterministyczny, w taki jednak sposób, aby nie zmienić akceptowanego języka.

TWIERDZENIE 3.1 *Dla dowolnego skończonego automatu niedeterministycznego M istnieje równoważny mu automat deterministyczny $\hat{M}$ akceptujący ten sam język, $L(M) = L(\hat{M})$.*

Dowód. Niech $M = (A, \Sigma, s_0, S_F, \pi)$ będzie danym automatem niedeterministycznym. Relacja π może być opisana jako odwzorowanie o wartościach w $\mathcal{P}(\Sigma)$, a więc będących zbiorami stanów,

$$\pi : \Sigma \times (A \cup \{\lambda\}) \to \mathcal{P}(\Sigma)\,.$$

Jest to punkt wyjścia do wykorzystania ogólnej konstrukcji opisanej we wzorze (1.2) w celu przekształcenia π w odpowiednią funkcję przejścia. Zbudujemy automat deterministyczny z tym samym alfabetem A

$$\hat{M} = \left(A, \hat{\Sigma}, \hat{s}_0, \hat{S}_F, \hat{\pi}\right),$$

w którym nowym zbiorem stanów jest zbiór potęgowy starego Σ, $\hat{\Sigma} = \mathcal{P}(\Sigma)$, stanem początkowym $\hat{s}_0$ jest zbiór składający się z poprzedniego stanu startowego s_0 i dodatkowo wszystkich tych stanów s, które są osiągalne z s_0 za pomocą jednego lub więcej λ-przejść. Zbiór $\hat{S}_F$ składa się z tych elementów $\hat{s} \in \hat{\Sigma}$, które zawierają przynajmniej jeden stary stan finalny, a więc dla których zachodzi $\hat{s} \cap S_F \neq \emptyset$. Nową funkcję przejścia

$$\hat{\pi} : \hat{\Sigma} \times A \to \hat{\Sigma}$$

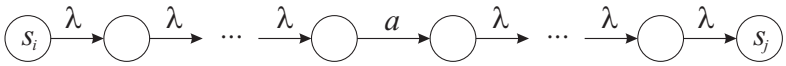
określamy następująco:

$$\hat{\pi}(\hat{s}, a) \;=\; \bigcup_{s_i \in \hat{s}} \left\{ s_j \in \Sigma :\; s_i a \;\Rightarrow_M^* \; a s_j \right\}. \tag{3.1}$$

Wyjaśnijmy powyższą notację. Zapis $s_i a \;\Rightarrow_M^*\; a s_j$ oznacza, że automat M może przenieść się w jednym lub kilku krokach ze stanu s_i do s_j, ponieważ jednak towarzyszy temu wczytanie tylko *jednej* litery a, dodatkowe przejścia mogą być co najwyżej puste,

$$s_i a \Rightarrow s_{i_1} a \Rightarrow \ldots \Rightarrow s_{i_k} a \Rightarrow a s_{i_{k+1}} \ldots \Rightarrow s_{i_r} a \Rightarrow s_j a\,. \tag{3.2}$$

W grafie automatu istnieje zatem ścieżka



Ponieważ argumentami $\hat{\pi}$ są zbiory $\hat{s}$ stanów automatu M, dla prawidłowego określenia jej wartości używamy w (3.1) sumy mnogościowej po wszystkich $s_i \in \hat{s}$. W ten sposób $\hat{\pi}$ jest swego rodzaju kolektywnym opisem wielu alternatywnych przejść automatu niedeterministycznego. Wartość $\hat{\pi}(\hat{s}, a)$ jest więc zbiorem wszystkich tych stanów pierwotnego automatu M, do których można przejść z co najmniej jednego stanu $s_i \in \hat{s}$, skanując przy tym literę a z możliwym wykorzystaniem λ-przejść. Wartości $\hat{\pi}$ określone są jednoznacznie, a więc jest to funkcja i w konsekwencji $\hat{M}$ jest deterministyczny. Ponadto możliwe jest, że $\hat{\pi}(\hat{s}, a) = \emptyset$, gdy w M nie istnieją przejścia z żadnego ze stanów $s_i \in \hat{s}$ dla litery a. Zauważmy przy tym, że $\emptyset \in \hat{\Sigma}$ jest legalnym stanem automatu $\hat{M}$, dla którego mamy $\hat{\pi}(\emptyset, a) = \emptyset$ dla wszystkich $a \in A$. Pełni więc on rolę stanu pułapkowego w $\hat{M}$. Z konstrukcji $\hat{\pi}$ widać więc, że jest ona totalną funkcją określoną na całej dziedzinie $\hat{\Sigma} \times A$.

Pokażemy na koniec, że $L(M) = L(\hat{M})$. Niech $\alpha = a_1 \dots a_n \in L(M)$, czyli $s_0\alpha \Rightarrow_M^* \alpha s_k$, dla pewnego stanu końcowego $s_k \in S_F$. Bardziej szczegółowo

$$s_0 a_1 \dots a_n \;\Rightarrow_M^*\; a_1 s_{i_1} a_2 \dots a_n \;\Rightarrow_M^*\; \dots \;\Rightarrow_M^*\; a_1 \dots a_{n-1} s_{i_{n-1}} a_n \;\Rightarrow_M^*\; a_1 \dots a_n s_k \,,$$

przy czym każde z wypisanych tu przejść $\Rightarrow_M^*$ może wzorem (3.2) zawierać puste ruchy automatu. Stąd zgodnie z (3.1) mamy

$$\begin{aligned}
\hat{s}_1 &= \hat{\pi}(\hat{s}_0, a_1) \ni s_{i_1}\,, && \text{skąd}\\
\hat{s}_2 &= \hat{\pi}(\hat{s}_1, a_2) \ni s_{i_2}\,, && \text{skąd}\\
&\vdots\\
\hat{s}_{n-1} &= \hat{\pi}(\hat{s}_{n-2}, a_{n-1}) \ni s_{i_{n-1}}\,, && \text{skąd}\\
\hat{s}_n &= \hat{\pi}(\hat{s}_{n-1}, a_n) \ni s_k\,,
\end{aligned}$$

a zatem $\hat{s}_n \cap S_F \neq \emptyset$, co oznacza, że $\alpha \in L(\hat{M})$.

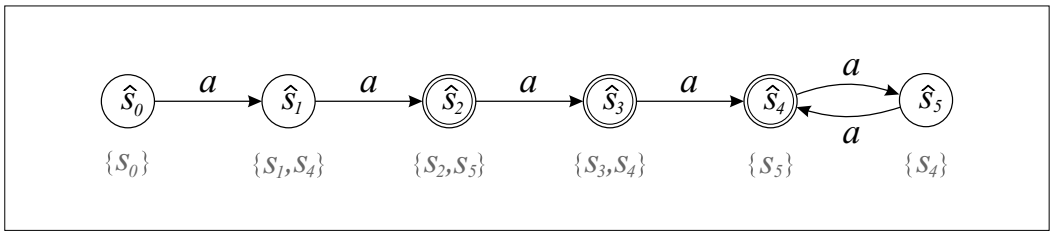
Weźmy z kolei dowolne $\alpha \in L(\hat{M})$ i niech $\hat{s}_0\alpha \Rightarrow_{\hat{M}}^* \alpha\hat{s}_n$, gdzie $\hat{s}_n \in \hat{S}_F$. Załóżmy nie wprost, że $\alpha \notin L(M)$, a więc że nie istnieje $s_k \in S_F$, dla którego mielibyśmy $s_0\alpha \Rightarrow_M^* \alpha s_k$. W każdym możliwym przypadku oznacza to, że $\hat{s}_n \cap S_F = \emptyset$. Bowiem jeśli automat niedeterministyczny M w każdym wariancie swoich możliwych ruchów zatrzymuje się przed dotarciem do ostatniej litery w α, wówczas musiałoby być $\hat{s}_n = \emptyset$. Jeśli zaś istnieje sekwencja przejść w M osiągająca ostatnią literę α, jej ostatni stan nie może, zgodnie z naszym założeniem, być stanem końcowym. Wtedy jednak musiałoby zachodzić $\hat{s}_n \cap S_F = \emptyset$. W obydwu wypadkach mielibyśmy $\hat{s}_n \notin \hat{S}_F$. Wobec otrzymanej sprzeczności $\alpha \in L(M)$, co kończy nasz dowód. □

Przykład 3.6 Zbudujemy SAD równoważny automatowi z Przykładu 3.4. Dla prostych automatów najłatwiej przeprowadzić tę konstrukcję na podstawie rysunku,

p. Rys. 3.8. Standardowo zaczynamy od $\hat{s}_0 = \{s_0\}$. Wyznaczamy stan $\hat{s}_1$ nowego automatu, do którego przechodzimy skanując literę a. Z rysunku odczytujemy, że $\hat{s}_1 = \{s_1, s_4\}$. Podobnie wyznaczamy $\hat{s}_2$ jako $\hat{\pi}(\hat{s}_1, a)$, a więc jako zbiór tych stanów pierwotnego automatu, do których można dotrzeć z s_1 lub s_4 z kolejną literą a na wejściu. Mamy więc $\hat{s}_2 = \{s_2, s_5\}$. Podobnie

$$\begin{aligned} \hat{s}_3 &= \hat{\pi}(\hat{s}_2, a) = \{s_3, s_4\} \\ \hat{s}_4 &= \hat{\pi}(\hat{s}_3, a) = \{s_5\} \\ \hat{s}_5 &= \hat{\pi}(\hat{s}_4, a) = \{s_4\}\,. \end{aligned}$$

Zauważmy na koniec tego procesu, że $\hat{\pi}(\hat{s}_5, a) = \{s_5\} = \hat{s}_4$. Mamy zatem



gdzie dla czytelności naszej konstrukcji pod nowymi stanami podpisaliśmy odpowiadające im zbiory stanów starego automatu. Stany $\hat{s}_2$, $\hat{s}_3$ i $\hat{s}_4$ są końcowe ponieważ — jako zbiory — zawierają co najmniej jeden ze starych stanów finalnych s_3 lub s_5.

PRZYKŁAD 3.7 Przeprowadzimy obecnie podobną konstrukcję dla SAN z Przykładu 3.5. Z Rys. 3.9 otrzymujemy $\hat{\pi}\Big(\{s_0\}, a\Big) = \{s_1, s_3\}$ oraz, wykorzystując puste przejście z s_0 do s_2, $\hat{\pi}\Big(\{s_0\}, b\Big) = \{s_3\}$. Dalej kolejno obliczamy

$$\begin{aligned} &\hat{\pi}\Big(\{s_1, s_3\}, a\Big) = \{s_2\} \quad \text{oraz} \quad \hat{\pi}\Big(\{s_1, s_3\}, b\Big) = \{s_2, s_3\}\,, \\ &\hat{\pi}\Big(\{s_2, s_3\}, a\Big) = \{s_2\} \quad \text{oraz} \quad \hat{\pi}\Big(\{s_2, s_3\}, b\Big) = \{s_3\}\,, \\ &\hat{\pi}\Big(\{s_2\}, a\Big) = \emptyset \quad \text{oraz} \quad \hat{\pi}\Big(\{s_2\}, b\Big) = \{s_3\}\,, \\ &\hat{\pi}\Big(\{s_3\}, a\Big) = \{s_2\} \quad \text{oraz} \quad \hat{\pi}\Big(\{s_3\}, b\Big) = \emptyset\,. \end{aligned}$$

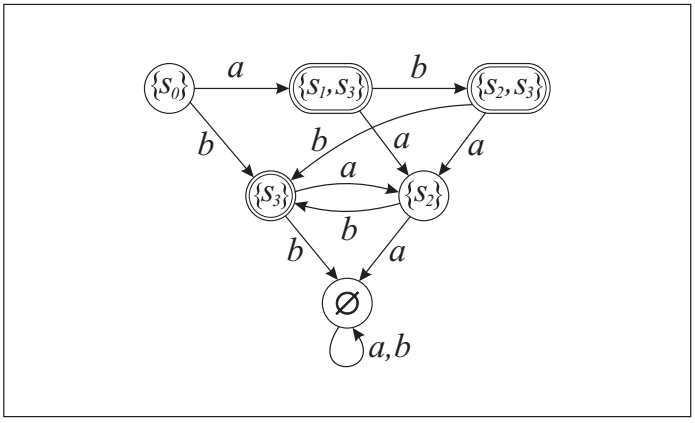
W końcu $\hat{\pi}(\emptyset, a) = \hat{\pi}(\emptyset, b) = \emptyset$, por. Rys. 3.10. Nowe stany końcowe to wszystkie stany zawierające s_3. Czytelnik zechce sprawdzić, że zbudowany wyżej automat akceptuje ten sam język.

Podsumujmy wyniki niniejszego rozdziału w twierdzeniu.

TWIERDZENIE 3.2 *Klasa języków rozpoznawanych prze skończone automaty niedeterministyczne jest identyczna z klasą języków regularnych, a więc rozpoznawanych przez skończone automaty deterministyczne.*

4 Optymalizacja automatów

O ile każdy skończony automat deterministyczny bądź niedeterministyczny akceptuje tylko jeden język, danemu językowi można przyporządkować dowolnie wiele



Rys. 3.10: Automat deterministyczny odpowiadający SAN z Przykładu 3.5.

różnych, rozpoznających go automatów. Najprostszym przykładem jest język pusty, akceptowany przez każdy automat, w którego grafie nie istnieje ścieżka z s_0 do jakiegokolwiek stanu finalnego. Z teoretycznego punktu widzenia nie ma istotnej różnicy między równoważnymi automatami, natomiast postać automatu, a zwłaszcza liczba jego stanów, może mieć praktyczne znaczenie przy poszukiwaniu wydajnej implementacji.

Opiszemy procedurę, która pozwala zredukować pewną liczbę nadmiarowych stanów w automacie deterministycznym.[2] Po pierwsze będą to bezużyteczne stany nieosiągalne ze stanu początkowego s_0. Oczywiście stanów takich raczej nie zobaczymy w przypadku automatów budowanych w świadomy, planowy sposób. Jednak mogą się one pojawić jako efekt uboczny pewnych ogólnych konstrukcji i przekształceń wykonywanych na automatach. Po drugie, bez zmiany akceptowanego przez automat języka można usunąć z niego wszystkie te stany, z których nie da się osiągnąć ani jednego stanu końcowego. Jeśli wymagamy by funkcja przejścia π była totalna, wystarczy — jak to widzieliśmy w podrozdziale o implementacji SAD — na końcu dodać jeden stan pułapkowy i odpowiednio rozszerzyć π.

Można pójść w działaniach optymalizacyjnych jeszcze dalej i wyeliminować stany, których obecność w automacie ma charakter redundantny: ma to miejsce wówczas, gdy inne stany automatu mogą pełnić tę samą rolę przy skanowaniu fragmentów napisów wejściowych. Załóżmy w tym miejscu, że π jest funkcją totalną. Na zbiorze stanów Σ określimy relację, która pozwoli nam stwierdzić czy i które jego elementy pełnią podobną, powielającą się rolę w automacie.

DEFINICJA 3.5 *Mówimy, że dwa stany s, s' automatu $M = (A, \Sigma, s_0, S_F, \pi)$ są nierozróżnialne, jeśli dla dowolnego napisu $\alpha \in A^*$ mamy*

$$s\,\alpha \Rightarrow^* \alpha\, r \qquad \textit{oraz} \qquad s'\,\alpha \Rightarrow^* \alpha\, r'\,,$$

przy czym obydwa stany r i r' jednocześnie należą bądź nie należą do S_F.

[2]Można opisać podobną metodę dla automatów niedeterministycznych, lecz jest ona mniej przejrzysta. Pamiętajmy jednak, że zawsze można w pierwszym kroku zastąpić SAN jego deterministycznym równoważnikiem i dopiero wtedy wykonać procedurę optymalizacyjną w celu wyeliminowania zbędnych stanów.

Innymi słowy nie istnieje napis α, dla którego skanowanie rozpoczęte w stanie s zakończyłoby się w stanie finalnym, podczas gdy podobne skanowanie rozpoczęte w s' doprowadziłoby nas do stanu nie-finalnego lub odwrotnie. Gdy choć jedno takie α istnieje, mówimy, że s i s' są *rozróżnialne*.

Relacja nierozróżnialności stanów jest oczywiście zwrotna i symetryczna. Nietrudno się przekonać, że jest także tranzytywna, a więc jest relacją równoważności na Σ. W konstrukcji optymalnego automatu $\tilde{M} = (A, \tilde{\Sigma}, \tilde{s}_0, \tilde{S}_F, \tilde{\pi})$ akceptującego język identyczny z wyjściowym $L(M)$ używamy klas abstrakcji tej relacji, a więc zbiorów stanów wzajemnie nierozróżnialnych,

$$[s] \;=\; \Big\{ r \in \Sigma : \; r \text{ i } s \text{ są nierozróżnialne} \Big\} .$$

Zbiór klas abstrakcji $[s]$ jest nowym zbiorem stanów $\tilde{\Sigma}$, stanem początkowym $\tilde{s}_0$ jest oczywiście klasa abstrakcji $[s_0]$, natomiast stany końcowe są klasami równoważności starych elementów S_F. Pozostaje określić nową funkcję $\tilde{\pi} : \tilde{\Sigma} \times A \to \tilde{\Sigma}$. Definicja jest niezwykle prosta:

$$\tilde{\pi}([s], a) = [r] \qquad \text{wtedy i tylko wtedy, gdy} \qquad \pi(s, a) = r \, .$$

Dla kompletności opisu tej metody podamy sposób konstrukcji klas równoważności $[s]$. W tym celu usuwamy najpierw z automatu wszystkie stany nieosiągalne z s_0, a następnie stany, z których nie można osiągnąć żadnego stanu końcowego. Uzupełniamy w końcu automat o stan pułapkowy tak, aby π stała się funkcją totalną. Tworzymy następnie tabelę indeksowaną parami stanów — element $T[s_i, s_j]$ opisuje relację między stanami s_i i s_j. Na początek w tabeli T oznaczamy jako *rozróżnialne* wszystkie pary stanów s_i, s_j takie, że $s_i \in S_F$ oraz $s_j \notin S_F$ lub odwrotnie. Dalej powtarzamy cyklicznie następujący krok aż do chwili, gdy nie pojawiają się już nowe pary stanów rozpoznanych jako rozróżnialne:

— Dla wszystkich par stanów s_i, s_j oraz wszystkich $a \in A$ obliczamy $\pi(s_i, a) = r_i$ oraz $\pi(s_j, a) = r_j$. Jeśli r_i oraz r_j oznaczono wcześniej jako rozróżnialne, stany s_i i s_j także oznaczamy jako rozróżnialne.

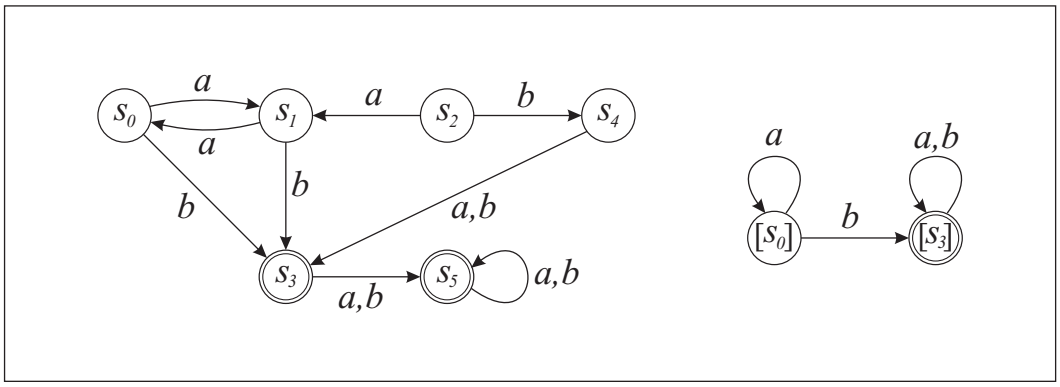
Jest oczywiste, że iteracja ta musi się zakończyć: w każdym obrocie liczba par stanów niesklasyfikowanych dotąd jako rozróżnialne maleje. Zatem albo liczba ta w pewnym momencie osiągnie wartość 0 (t.j. wszystkie stany są rozróżnialne i optymalizacja jest niemożliwa) albo przestanie się zmniejszać i iteracja dobiegnie końca. Z T można wówczas odczytać, które stany pozostają nierozróżnialne i zbudować na tej podstawie poszukiwane klasy abstrakcji.

Zanim przejdziemy do przykładu, sformułujemy bez dowodu twierdzenie gwarantujące poprawność opisanej tu metody redukcji.

TWIERDZENIE 3.3 *Dla danego automatu deterministycznego M procedura redukcji generuje równoważny mu automat deterministyczny $\tilde{M}$, a więc taki, że*

$$L(M) = L(\tilde{M}) \, .$$

Ponadto $\tilde{M}$ jest minimalnym pod względem liczby stanów automatem deterministycznym o tej własności.



Rys. 3.11: Automat z Przykładu 3.8 i jego zoptymalizowana postać.

Przykład 3.8 Procedurze redukcji poddamy obecnie automat z Rys. 3.11. Zauważmy na początek, że stany s_2 i s_4 są nieosiągalne z s_0, można je zatem usunąć. Dla pozostałych stanów funkcja przejścia π jest określona totalnie, nie ma więc potrzeby dodawania nowego stanu pułapkowego. Z kolei budujemy tabelę T, oznaczając w pierwszym kroku symbolem "X" te pary stanów, z których jeden jest finalny, a drugi nie:

	s_0	s_1	s_3	s_5
s_0			X	X
s_1			X	X
s_3	X	X		
s_3	X	X		

W pierwszej iteracji sprawdzimy, czy stany w parach s_0 i s_1 oraz s_3 i s_5 są rozróżnialne. Mamy

$$\pi(s_0, a) = s_1 \qquad \text{oraz} \qquad \pi(s_1, a) = s_0\,,$$
$$\pi(s_0, b) = s_3 \qquad \text{oraz} \qquad \pi(s_1, b) = s_3\,,$$

a więc test ten nie prowadzi do uznania s_0 i s_1 za rozróżnialne. Podobnie obliczamy

$$\pi(s_3, a) = s_5 \qquad \text{oraz} \qquad \pi(s_5, a) = s_5\,,$$
$$\pi(s_3, b) = s_5 \qquad \text{oraz} \qquad \pi(s_5, b) = s_5\,,$$

co również nie pociąga rozróżnialności s_3 i s_5. Ponieważ w tej iteracji nie rozpoznaliśmy żadnych nowych stanów rozróżnialnych, proces kończy się. Z tabeli T możemy odczytać teraz klasy stanów wzajemnie nierozróżnialnych: są nimi s_0 i s_1 oraz s_3 i s_5. Wybierając jako reprezentantów tych klas s_0 i odpowiednio s_3, otrzymujemy jako zbiór stanów zredukowanego automatu $\tilde{\Sigma} = \{[s_0], [s_3]\}$. Obliczamy funkcję przejścia:

— ponieważ $\pi(s_0, a) = s_1$, więc $\tilde{\pi}([s_0], a) = [s_1] = [s_0]$

— na podstawie $\pi(s_0, b) = s_3$, otrzymujemy $\tilde{\pi}([s_0], b) = [s_3]$

— skoro $\pi(s_3, a) = \pi(s_3, b) = s_5$, mamy $\tilde{\pi}([s_3], a) = \tilde{\pi}([s_3], b) = [s_5] = [s_3]$.

Wynik naszej konstrukcji przedstawia Rys. 3.11.

III.2 Wyrażenia regularne

O wyrażeniach regularnych jako technice definiowania pewnych klas języków wspomnieliśmy już w Rozdziale II. Określimy obecnie prosty typ takich wyrażeń, które w bardzo zwięzły sposób charakteryzować będą języki regularne.

1 Wyrażenie regularne i definiowany przez niego język

DEFINICJA 3.6 *Wyrażeniem regularnym nad alfabetem A nazywamy napis zbudowany wg następujących reguł:*

(i) $\emptyset$, λ *oraz każda litera* a *alfabetu* A *są prostymi wyrażeniami regularnymi;*

(ii) Jeśli ρ *oraz* σ *są wyrażeniami regularnymi, są nimi również napisy* $\rho\sigma$, $\rho+\sigma$, ρ^* *oraz* (ρ)*;*

(iii) Jakikolwiek inny napis, którego nie można utworzyć w skończonej liczbie kroków przy pomocy reguł (i) oraz (ii) nie jest wyrażeniem regularnym.

Tak więc proste wyrażenia regularne są zwykle pojedynczymi literami alfabetu. Symbol λ oznacza napis pusty, a więc napis o długości 0, natomiast symbol $\emptyset$ oznacza brak napisu w ogóle. Reguła konkatenacji $\rho\sigma$ pozwala iteracyjnie budować wyrażenia dowolnej (lecz skończonej) długości, np. $\rho = a$, $\sigma = b$, $\eta = \rho\sigma = ab$, $\xi = \sigma\eta = bab$ itp. Stosując konkatenację łącznie z regułami "+" oraz "*", a także używając pomocniczych nawiasów, można budować dowolnie złożone wyrażenia regularne. Oto kilka przykładów:

$$a+ab, \quad (a+bbb)^*bab, \quad (\lambda+a)b^*(ab+abb+\emptyset), \quad 0+1(0+1)^*, \quad a(aa)^*(bb)^*\,.$$

Powiązanie z językami uzyskujemy nadając wyrażeniom regularnym stosowną interpretację w terminach zbiorów "pasujących" do nich napisów.

DEFINICJA 3.7 *Językiem definiowanym przez wyrażenie regularne* ρ*, oznaczanym przez* $L(\rho)$*, nazywamy zbiór napisów powstający wg następujących reguł:*

(i) $L(\emptyset) = \emptyset$

(ii) $L(\lambda) = \{\lambda\}$

(iii) $L(a) = \{a\}$ *dla* $a \in A$

(iv) $L(\rho+\sigma) = L(\rho) \cup L(\sigma)$

(v) $L(\rho\sigma) = L(\rho)\,L(\sigma)$

(vi) $L(\rho^*) = \Big(L(\rho)\Big)^*$

(vii) $L\Big((\rho)\Big) = L(\rho)$.

Zatem znak "+" przekłada się na sumę mnogościową języków odpowiadających połączonym nim podwyrażeniom, konkatenacja wyrażeń odpowiada konkatenacji

odpowiednich języków, a gwiazdka — domknięcie konkatenacyjne języka (por. podrozdział II.3). Np. dla wyrażenia $(a+bbb)^*bab$ z poprzedniego przykładu mamy

$$\begin{aligned} L\Big((a+bbb)^*bab\Big) &= L\Big((a+bbb)^*\Big)\,L(bab) \;=\; \Big(L(a+bbb)\Big)^*\,L(bab) \\ &= \Big(L(a)\cup L(bbb)\Big)^*\,L(bab) \;=\; \{a,bbb\}^*\{bab\} \\ &= \{\lambda,a,bbb,aa,abbb,bbba,bbbbbb,\ldots\}^*\{bab\} \\ &= \{bab,abab,bbbbab,aabab,abbbbab,bbbabab,bbbbbbbab,\ldots\}\,. \end{aligned}$$

Oto kilka prostych do udowodnienia (por. Zadanie 12 na końcu rozdziału) reguł upraszczania wyrażeń regularnych wynikających z odpowiednich tożsamości mnogościowych dla języków:

— dla symbolu $\emptyset$: $\quad \rho+\emptyset=\rho\,,\quad \rho\emptyset=\emptyset\rho=\emptyset\,,\quad \emptyset^*=\lambda\,;$

— dla symbolu λ: $\quad \rho\lambda=\lambda\rho=\rho\,,\quad \lambda^*=\lambda\,;$

— dla gwiazdki: $\quad (\rho^*)^*=\rho^*\,.$

Natomiast nieprawdziwe są następujące równości: $(\rho+\sigma)^*=\rho^*+\sigma^*$ oraz $(\rho\sigma)^*=\rho^*\sigma^*$. W zadaniach na końcu bieżącego rozdziału proponujemy czytelnikowi zweryfikowanie prawdziwości kilku podobnych formuł.

Przykład 3.9 Zadanie polega na zbudowaniu wyrażenia regularnego nad alfabetem $A=\{a,b\}$, generującego język złożony z wszystkich napisów nie zawierających 2 sąsiadujących liter b. Wyrażenie $(a+b)^*$ generuje *wszystkie* napisy nad A (oznacza bowiem to samo co A^*), więc możemy zmodyfikować je tak, aby b występowało zawsze w sąsiedztwie a, np. $(a+ba)^*$. Nie jest to jednak jeszcze kompletne rozwiązanie, bowiem wyklucza napisy kończące się literą b. Zatem należy uzupełnić je następująco:

$$(a+ba)^*(\lambda+b)\,.$$

Nieco inne rozwiązanie tego samego problemu otrzymamy zauważając, że każda litera b w słowie naszego języka będzie otoczona pewną liczbą liter a, a^*ba^*, przy czym ponieważ wzorzec ten może się powtarzać wielokrotnie, po b powinna wystąpić co najmniej jedna litera a. Prowadzi nas to do wyrażenia $(a^*baa^*)^*$. Należałoby jeszcze uwzględnić ciągi zbudowane z samych liter a, a więc otrzymujemy $a^*+(a^*baa^*)^*$ lub równoważnie lecz nieco prościej $(a+a^*baa^*)^*$. W końcu, podobnie jak poprzednio, trzeba zagwarantować jeszcze możliwość zakończenia napisu przez b,

$$(a+a^*baa^*)^*(\lambda+b)\,.$$

Jak widać, ten sam język może być generowany przez różne wyrażenia regularne. Poznamy później sposób na zweryfikowanie czy dane dwa wyrażenia generują identyczny język.

2 Wyrażenia regularne a skończone automaty

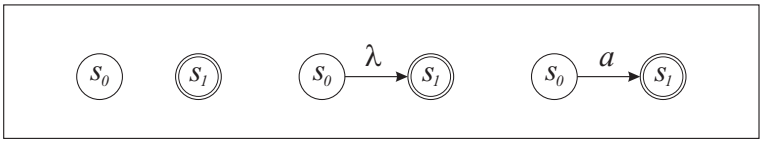
W tym podrozdziale zbadamy relację między językami zadanymi przez wyrażenia regularne a językami określonymi przez skończone automaty. Pokażemy najpierw, że

dla każdego wyrażenia regularnego można zbudować automat niedeterministyczny, akceptujący ten sam język. Analogiczna, bezpośrednia konstrukcja automatu deterministycznego, choć możliwa, okazuje się trudniejsza. Tu między innymi ujawnia się praktyczna przewaga automatów niedeterministycznych nad deterministycznymi, bowiem są one zwykle bardziej poręczne w zadaniach konstrukcyjnych. Ponieważ jednak znamy już metodę konwersji automatu niedeterministycznego w równoważny mu automat deterministyczny, ten ostatni dla zadanego wyrażenia ρ można bez trudu uzyskać dwustopniowo. Drugie twierdzenie, które udowodnimy w dalszej części tego podrozdziału orzeka, że mając dany skończony automat niedeterministyczny, można na jego podstawie w systematyczny sposób zbudować równoważne wyrażenie regularne.

Przejdźmy do pierwszego twierdzenia — jego dowód zawiera ogólną metodę konstrukcji SAN dla zadanego wyrażenia regularnego.

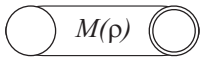
TWIERDZENIE 3.4 *Dla dowolnego wyrażenia regularnego ρ można skonstruować skończony automat niedeterministyczny M, taki że $L(\rho) = L(M)$.*

DOWÓD. Na początek określimy sposób budowy automatów dla prostych wyrażeń regularnych $\emptyset$, λ oraz $a \in A$. Są nimi odpowiednie automaty pokazane na Rys. 3.12.



Rys. 3.12: Automaty odpowiadające prostym wyrażeniom regularnym $\emptyset$, λ i $a \in A$.

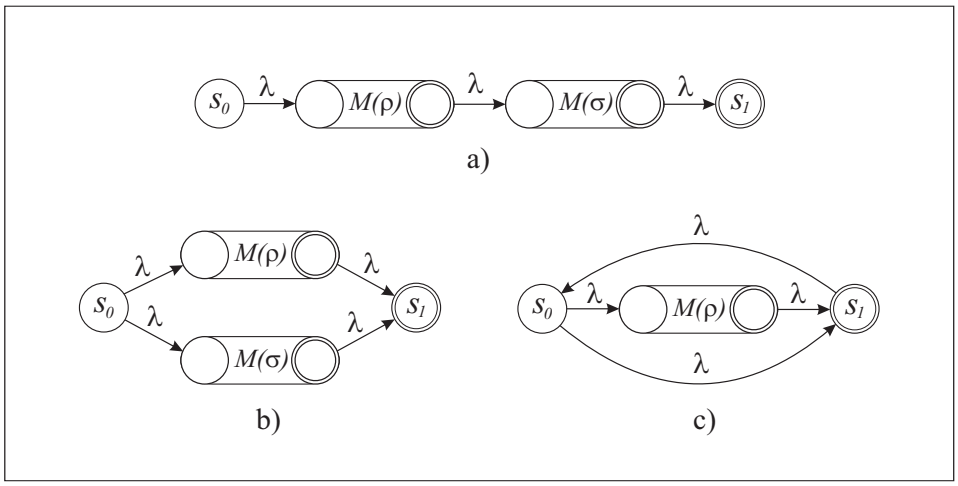
Niech teraz diagram postaci



przedstawia syntetycznie niedeterministyczny automat odpowiadający wyrażeniu ρ. Kółko po lewej stronie reprezentuje jego stan startowy, natomiast po prawej — stan końcowy.[3]

Dla wyrażeń złożonych $\rho\sigma$, $\rho+\sigma$ oraz ρ^* odpowiadające im automaty niedeterministyczne powstają przez dodanie jednego nowego, wspólnego stanu początkowego s_0 oraz nowego, wspólnego stanu końcowego s_f i uzupełnienie połączeń tak jak pokazano na Rys. 3.13. Jest oczywiste, że w każdym z tych przypadków język akceptowany przez dany automat jest identyczny z językiem generowanym przez powiązane wyrażenie regularne.

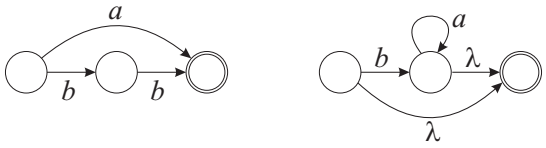
[3] Zauważmy, że o automacie niedeterministycznym możemy bez zmniejszenia ogólności założyć, iż posiada on dokładnie jeden stan końcowy. W innym przypadku należałoby uzupełnić graf automatu o jeden nowy stan, oznaczony teraz jako jedyny stan końcowy i poprowadzić do niego krawędzie z etykietą λ ze wszystkich dotychczasowych stanów końcowych. Oczywiste jest, że takie rozszerzenie nie wpływa na język akceptowany przez ten automat.



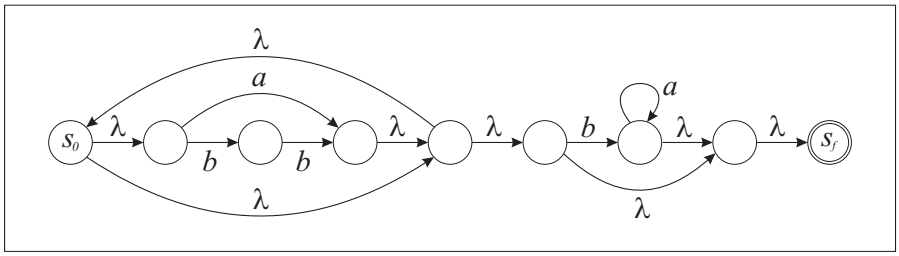
Rys. 3.13: Konstrukcja automatów dla wyrażeń a) $\rho\sigma$, b) $\rho + \sigma$ i c) ρ^*.

Konstrukcję tę możemy oczywiście powtórzyć hierarchicznie dla bardziej złożonych wyrażeń regularnych, podobnie jak systematycznie budujemy języki $L(\rho)$ zgodnie z Definicją 3.7, rozpoczynając od prostych podwyrażeń. □

PRZYKŁAD 3.10 Skonstruujemy automat dla wyrażenia $\rho = (a + bb)^*(ba^* + \lambda)$. Rozpoczynamy od zbudowania prostych automatów dla podwyrażeń $a + bb$ oraz $ba^* + \lambda$:



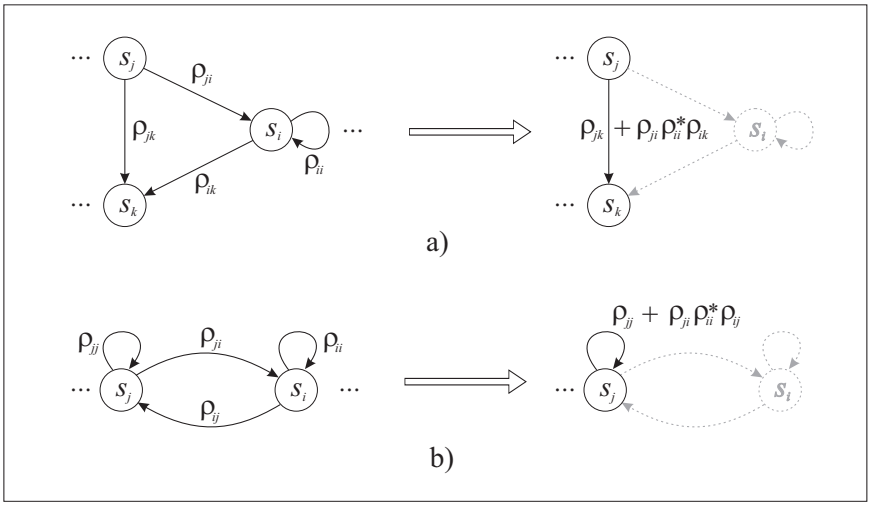
Z tych dwóch elementów tworzymy następnie automat odpowiadający całemu wyrażeniu, por. rysunek poniżej.



Rys. 3.14: Kompletny automat dla wyrażenia regularnego $\rho = (a + bb)^*(ba^* + \lambda)$.

Twierdzenie 3.4 pokazuje, że języki definiowane przez wyrażenia regularne są podklasą języków regularnych. Kolejne twierdzenie formułuje implikację odwrotną: każdy język akceptowany przez SAN może być opisany przez pewne wyrażenie regularne.

TWIERDZENIE 3.5 *Dla każdego skończonego automatu niedeterministycznego M istnieje wyrażenie regularne ρ takie, że $L(M) = L(\rho)$.*



Rys. 3.15: a) Procedura redukcji stanu s_i z uaktualnieniem etykiety ρ_{jk} przy krawędzi między stanami s_j i s_k. b) To samo dla sytuacji, gdy $j = k$.

Zamiast sformalizowanego dowodu powyższego twierdzenia, podamy konstruktywną metodę tworzenia wyrażenia regularnego równoważnego danemu automatowi. Wykorzystamy w tym celu tzw. uogólnione grafy przejścia. Grafy tego typu różnią się od zwykłych grafów SAN jedynie tym, że ich krawędzie są etykietowane dowolnymi wyrażeniami regularnymi, a nie tylko pojedynczymi literami lub symbolem λ. Zauważmy, że każdy graf SAN jest szczególnym przypadkiem uogólnionego grafu przejścia, gdyż etykiety krawędzi są w tym wypadku najprostszymi wyrażeniami regularnymi.

Dla każdej ścieżki z wierzchołka początkowego do końcowego w takim grafie tworzymy wyrażenie regularne konkatenując ze sobą etykiety kolejnych krawędzi. Suma uzyskanych w ten sposób wyrażeń po wszystkich takich ścieżkach w grafie określa odpowiadający mu język.

Główną zaletą uogólnionych grafów przejścia jest fakt, że można je stosunkowo łatwo przekształcać, redukując liczbę ich stanów. Nie naruszamy przy tym opisywanego języka, ponieważ przy eliminowaniu stanów w określony sposób zmieniamy wyrażenia etykietujące krawędzie. Procedurę redukcji można powtarzać aż do pozostawienia w grafie jedynie dwóch stanów: początkowego i końcowego. Z tak uproszczonego grafu łatwo już odczytać poszukiwane wyrażenie regularne.

Na początku zakładamy, że graf wyjściowego automatu niedeterministycznego M zawiera tylko jeden stan końcowy s_f i że jest on różny of s_0. Założenie to można w razie konieczności wypełnić dodając do grafu nowy stan końcowy i odpowiednie λ-krawędzie (por. przypis na str. 58). Traktując ten graf jako uogólniony graf przejścia, zredukujemy kolejno wszystkie stany za wyjątkiem s_0 i s_f.

Niech ρ_{jk} oznacza wyrażenie regularne przypisane krawędzi grafu $s_j \to s_k$. Zgodnie z tą konwencją, ρ_{jj} oznacza etykietę przypisaną pętli przy wierzchołku s_j. Oto jak wygląda redukcja stanu s_i.

(i) Wybieramy parę stanów s_j, s_k w grafie, $j \neq i \neq k$, dla których istnieją obydwie krawędzie $s_j \to s_i$ oraz $s_i \to s_k$.

(ii) Jeśli istnieje krawędź $s_j \to s_k$ z etykietą ρ_{jk}, nadajemy jej nową wartość $\rho_{jk} + \rho_{ji}\rho_{ii}^*\rho_{ik}$ (lub $\rho_{jk} + \rho_{ji}\rho_{ik}$ jeśli przy wierzchołku s_i nie ma pętli). W przypadku gdy nie istnieje krawędź z s_j do s_k, tworzymy ją i nadajemy jej etykietę $\rho_{ji}\rho_{ii}^*\rho_{ik}$ (lub odpowiednio $\rho_{ji}\rho_{ik}$, gdy nie ma pętli przy s_i), por. Rys. 3.15 a.

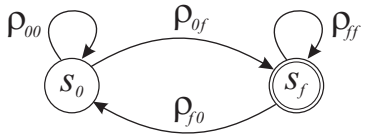
(iii) Powtarzamy punkty (i) oraz (ii) dla wszystkich par stanów s_j, s_k.

Po zakończeniu powyższej iteracji usuwamy stan s_i z grafu wraz ze wszystkimi przyległymi do niego krawędziami. Następnie powtarzamy całą procedurę dla kolejnego s_i, aż do usunięcia z grafu wszystkich stanów za wyjątkiem s_0 i s_f.

Zwróćmy uwagę na jeden istotny aspekt techniczny powyższej procedury, który nie od razu jest widoczny. Ponieważ graf przejścia jest skierowany, czynności (i) oraz (ii) wykonujemy dla każdej z par stanów s_j i s_k *dwukrotnie*: raz od s_j przez s_i do s_k, a drugi raz w odwrotnej kolejności, po zamianie rolami j z k. Za pierwszym razem obliczamy nową etykietę ρ_{jk} krawędzi z s_j do s_k, za drugim zaś etykietę ρ_{kj} *całkiem innej* krawędzi z s_k do s_j. Ponadto (i)–(ii) wykonujemy tak samo w sytuacji gdy $j = k$, zobrazowanej na Rys. 3.15 b, aktualizując etykietę ρ_{jj} pętli przy wierzchołku s_j.

Sens procedury redukcji jest następujący: gdy usuwamy stan s_i z grafu wraz z przyległymi do niego krawędziami, z opisu języka znikają cząstkowe wyrażenia regularne przypisane tym krawędziom. Trzeba więc zastępczo zapamiętać je w etykietach innych krawędzi. Z bezpośrednim przejściem z s_j do s_k związane jest wyrażenie ρ_{jk}, natomiast przejście pośrednie przez stan s_i, czyli $s_j \to s_i \to s_k$, generuje wyrażenie $\rho_{ji}\rho_{ii}^*\rho_{ik}$ (ew. bez ρ_{ii}^* jeśli pętla przy s_i nie występuje). Wystarczy więc etykietę krawędzi $s_j \to s_k$ zmodyfikować tak, by zapamiętywała alternatywę obu tych wyrażeń — "przejście bezpośrednie" lub "przejście pośrednie przez s_i", a zatem wyrażenie $\rho_{jk} + \rho_{ji}\rho_{ii}^*\rho_{ik}$.

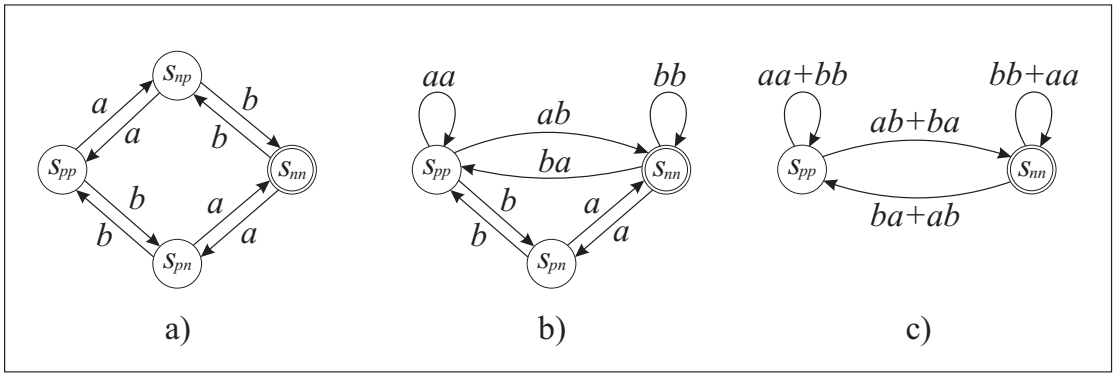
Po zakończeniu procedury eliminacji graf przejścia będzie miał postać:



Stąd już łatwo odczytać globalne wyrażenie regularne dla języka $L(M)$ wyjściowego automatu: rozpoczynając w stanie s_0, możemy najpierw dokonać dowolnie wielu kroków pętlą przy tym stanie, a następnie przejść do s_f. Generuje to wyrażenie $\rho_{00}^*\rho_{0f}$ opisujące już elementy języka $L(M)$, ponieważ osiągnęliśmy stan końcowy. Jednak ścieżkę tę można dalej kontynuować, obracając się pętlą przy s_f lub wracając do stanu s_0, ponownie korzystając 0 lub więcej razy z pętli przy s_0, by w końcu powrócić do s_f. Odpowiadające temu przejściu wyrażenie to $\rho_{ff}^* + \rho_{f0}\rho_{00}^*\rho_{0f}$, a cały ten fragment może być powtórzony dowolnie wiele razy. Reasumując,

$$\rho \;=\; \rho_{00}^*\rho_{0f}\Big(\rho_{ff} + \rho_{f0}\rho_{00}^*\rho_{0f}\Big)^*, \tag{3.3}$$

przy czym dokonaliśmy minimalnego uproszczenia usuwając gwiazdkę znad ρ_{ff} poza nawias. Z konstrukcji wynika oczywiście, że $L(M) = L(\rho)$.



Rys. 3.16: a) Automat deterministyczny akceptujący język z Przykladu 3.11. b) Postać grafu po redukcji stanu s_{np}. c) Postać końcowa, po redukcji stanu s_{pn}.

PRZYKŁAD 3.11 Zbudujemy wyrażenie regularne dla następującego języka nad $A = \{a, b\}$:

$$L = \{\alpha : |\alpha|_a \text{ i } |\alpha|_b \text{ są nieparzyste}\}.$$

Zdarzają się sytuacje, w których wyrażenie można napisać bezpośrednio na podstawie definicji języka, jednak są one raczej rzadkie. Dzieje się tak w przypadkach, gdy definicja języka zasadza się głównie na strukturze rozmieszczenia liter w słowach, co bliskie jest idei wyrażeń regularnych. W obecnym przykładzie litery a i b mogą występować w słowach języka w dowolnym układzie, dowolnie długimi seriami, ważne jest jedynie, aby liczby wystąpień każdej z liter były nieparzyste. Na pierwszy rzut oka nie jest więc jasne jak powinno wyglądać odpowiednie wyrażenie regularne. Stosunkowo łatwo jest jednak zbudować automat dla tego języka. Zaczniemy więc od automatu, a potem metodą redukcji odtworzymy równoważne wyrażenie regularne.

Oznaczmy stany automatu jako s_{pp}, s_{pn}, s_{np} i s_{nn}, gdzie pierwszy indeks oznacza, że wczytaliśmy dotąd parzystą (p) lub nieparzystą (n) liczbę liter a, drugi zaś oznacza to samo dla liter b. Jako że 0 jest liczbą parzystą, stanem startowym jest s_{pp}, stanem końcowym zgodnie z treścią zadania jest natomiast s_{nn}. Automat przedstawiliśmy na rysunku 3.16 a. Jest on deterministyczny, co oczywiście nie przeszkadza metodzie redukcji.

Eliminacji podlegają stany s_{pn} i s_{np}. Zacznijmy procedurę redukcji od stanu $s_i = s_{np}$, w pierwszym kroku biorąc $s_j = s_{pp}$ oraz $s_k = s_{nn}$. Przejście bezpośrednie z s_{pp} do s_{nn} nie istnieje (brak krawędzi),[4] a przejściu pośredniemu przez s_{np} odpowiada wyrażenie ab. Tworzymy więc krawędź między s_{pp} i s_{nn} z etykietą ab, która przechowa informację o istnieniu tego przejścia w oryginalnym grafie, gdy usuniemy stan s_{np}. Podobnie przyjmując, że $s_j = s_{nn}$ oraz $s_k = s_{pp}$, otrzymamy między nimi skierowaną przeciwnie krawędź z etykietą ba.

Wybierzmy teraz w grafie kolejna parę stanów s_j i s_k. Zauważmy, że ponieważ stan s_{pn} nie posiada krawędzi łączących go z podlegającym aktualnie eliminacji stanem $s_i = s_{np}$, nie ma potrzeby uwzględnienia go w roli s_j lub s_k (przypomnijmy, że w punkcie (i) procedury eliminacji wymagamy istnienia obydwu krawędzi $s_j \to s_i$ oraz $s_i \to s_k$). Do ukończenia iteracji pozostaje więc przyjąć $s_j = s_k = s_{pp}$ oraz

[4] Można uważać, że tej sytuacji odpowiada wyrażenie regularne $\emptyset$.

$s_j = s_k = s_{nn}$. W pierwszym przypadku, istnieje jedynie pośrednie przejście z s_{pp} do samego siebie przez s_{np}, co odpowiada wyrażeniu aa. Tworzymy więc pętlę przy wierzchołku s_{pp} z etykietą aa. Podobna analiza dla s_{nn} skutkuje utworzeniem przy nim pętli z etykietą bb. Ponieważ rozpatrzyliśmy wszystkie możliwe pary s_j, s_k, iteracja kończy się i usuwamy stan $s_i = s_{np}$ z grafu wraz z przyległymi do niego krawędziami, por. Rys. 3.16 b.

Pozostaje teraz wyeliminować stan s_{pn}. W iteracji w roli s_j i s_k kolejno rozważyć trzeba stany s_{pp} z s_{nn} oraz odwrotnie, a także s_{pp} i s_{nn} same z sobą. Rozpocznijmy od $s_j = s_{pp}$ i $s_k = s_{nn}$. Łącząca je krawędź ma etykietę ab, natomiast przejście pośrednie $s_{pp} \to s_{pn} \to s_{nn}$ odpowiada wyrażeniu ba. Zatem nowa etykieta krawędzi z s_{pp} do s_{nn} ma postać ab+ba. Zamieniając następnie rolami s_{pp} i s_{nn}, otrzymamy dla łączącej je odwrotnej krawędzi podobne wyrażenie $ba + ab$. W końcu biorąc $s_j = s_k = s_{pp}$, do wyrażenia aa etykietującego pętlę przy tym wierzchołku dopisać trzeba człon bb odpowiadający alternatywnemu przejściu pośredniemu z s_{pp} do samego siebie przez s_{pn}. Podobnie w etykiecie pętli przy s_{nn} pojawi się alternatywny człon aa.

W tym momencie można już wyeliminować stan s_{pn} i procedura kończy się. Ostateczną postać grafu przejścia pokazuje rysunek 3.16 c. Możemy więc odczytać z niego globalne wyrażenie regularne dla naszego języka, zgodnie ze schematem (3.3):

$$\rho \;=\; (aa + bb)^*(ab + ba)\Big(bb + aa \;+\; (ba + ab)(aa + bb)^*(ab + ba)\Big)^* .$$

Wyrażenia regularne uzyskane powyższą metodą mogą przyjmować dość skomplikowaną postać, algorytm nie daje bowiem gwarancji utworzenia najbardziej zwartej ich formy. Warto zauważyć, że upraszczanie wyrażeń regularnych jest samo w sobie raczej skomplikowanym zadaniem.

Zakończymy niniejszy podrozdział podsumowującym twierdzeniem. W literaturze wynik ten nazywany jest twierdzeniem Kleenego.

TWIERDZENIE 3.6 *Klasa języków regularnych (a więc akceptowanych przez skończone automaty deterministyczne lub niedeterministyczne) jest tożsama z klasą języków definiowanych przez wyrażenia regularne.*

III.3 Gramatyki regularne

Trzecią metodą charakteryzacji języków regularnych jest ich opis przy pomocy gramatyk o specyficznie ograniczonej postaci produkcji. W Rozdziale II.4 podaliśmy ogólną definicję gramatyki i definiowanego przez nią języka, por. Definicje 2.4 i 2.5. Zajmiemy się teraz gramatykami szczególnej postaci i pokażemy w jaki sposób powiązane one będą ze skończonymi automatami niedeterministycznymi.

DEFINICJA 3.8 *Gramatykę nazywamy liniową jeśli wszystkie jej produkcje mają postać*

$$X \to \alpha Y \beta \quad lub \quad X \to \alpha \,, \tag{3.4}$$

gdzie X i Y są nieterminalami, natomiast α i β zbudowane są wyłącznie z liter terminalnych.

Zatem produkcje gramatyk liniowych posiadają wyłącznie pojedyncze symbole nieterminalne po lewej i co najwyżej po jednym takim symbolu po prawej stronie. Jak się przekonamy, postać produkcji wpływa na klasę języków, które można opisać przy ich pomocy. Gramatyki *prawostronnie liniowe* są to gramatyki liniowe, takie, że dodatkowo żądamy by $\beta = \lambda$ w (3.4). A zatem ich wszystkie bez wyjątku produkcje przyjmują postać

$$X \to \alpha Y \quad \text{lub} \quad X \to \alpha\,,$$

gdzie jak poprzednio $X, Y \in V$ oraz $\alpha \in A^*$. Podobnie, gramatyki *lewostronnie* liniowe spełniają żądanie $\alpha = \lambda$ we wszystkich produkcjach (3.4).

DEFINICJA 3.9 *Gramatyka regularna jest to gramatyka jednostronnie (t.j. prawo- lub równoważnie lewostronnie) liniowa.*

Można łatwo uzasadnić dlaczego kategorie gramatyk lewo- i prawostronnie liniowych są równoważne w tym sensie, że opisują tę samą klasę języków. Po pierwsze, dwa twierdzenia, które udowodnimy za chwilę pokazują, że języki generowane przez prawostronnie liniowe gramatyki są to dokładnie języki regularne. Po drugie zauważmy, że jeśli przetransformujemy dowolną gramatykę G, zastępując lewe i prawe strony jej produkcji $\alpha \to \beta$ ich lustrzanymi odbiciami, $\overleftarrow{\alpha} \to \overleftarrow{\beta}$, wówczas otrzymamy gramatykę $\overleftarrow{G}$ generującą lustrzane odbicie języka $L(G)$, a więc $L(\overleftarrow{G}) = \overleftarrow{L(G)}$. Po trzecie, rozwiązując Zadanie 15 b) na końcu rozdziału przekonamy się, że odbicie lustrzane języka regularnego jest językiem regularnym. Z tych faktów łatwo wynika, że dwie wersje definicji regularności gramatyki w oparciu o prawo- lub lewostronnie liniowe produkcje są równoważne.

PRZYKŁAD 3.12 Rozważmy prawostronnie liniową gramatykę postaci[5]

$$\begin{aligned} S &\to aaS \mid bbS \mid abX \mid baX \\ X &\to aaX \mid bbX \mid abS \mid baS \mid \lambda\,. \end{aligned}$$

Warto zauważyć, że samo podanie zbioru produkcji w zasadzie wystarczy za definicję gramatyki, bowiem określa ono jednoznacznie zbiory A i V. Przykładowe wyprowadzenie w tej gramatyce to

$$S \Rightarrow aaS \Rightarrow aabaX \Rightarrow aabaabS \Rightarrow aabaabbbS \Rightarrow aabaabbbabX \Rightarrow aabaabbbab\,,$$

gdzie w ostatnim kroku użyliśmy produkcji $X \to \lambda$ w celu pozbycia się symbolu nieterminalnego. Końcowy napis jest więc elementem języka generowanego przez tę gramatykę. Choć być może nie jest to oczywiste na tym etapie, jest to język z Przykładu 3.11. Zaobserwujmy, że wyprowadzenie w gramatyce prawostronnie liniowej ma tę szczególną własność, że na każdym jego etapie mamy tylko jeden symbol nieterminalny stojący dokładnie na końcu tworzonej formy zdaniowej.

W dwóch kolejnych twierdzeniach pokażemy, że języki generowane przez gramatyki regularne są to dokładnie języki akceptowane przez skończone automaty,

[5]Przypomnijmy, że stosujemy skrótowy zapis $X \to \gamma \mid \eta$ na oznaczenie dwóch (lub ogólnie kilku) produkcji $X \to \gamma$ i $X \to \eta$ na raz.

a więc języki regularne. Charakteryzacja przez gramatyki jednostronnie liniowe jest więc czwartą, równoważną metodą określania języków regularnych, obok automatów deterministycznych, niedeterministycznych oraz wyrażeń regularnych.

TWIERDZENIE 3.7 *Dla każdej gramatyki regularnej (prawostronnie liniowej) G istnieje skończony automat niedeterministyczny M taki, że $L(M) = L(G)$.*

DOWÓD. Dla każdej produkcji $X \to a_1 a_2 \ldots a_k Y$ w G utwórzmy podstrukturę automatu niedeterministycznego postaci

natomiast dla każdej produkcji terminalnej $X \to a_1 a_2 \ldots a_k$ — podstrukturę

Nieoznaczonym na rysunkach stanom trzeba oczywiście nadać niepowtarzalne etykiety, zaś s_f dodajemy jako stan końcowy. Kompletny automat uzyskujemy przez odpowiednie połączenie utworzonych podstruktur — przez identyfikację stanów etykietowanych tą samą literą nieterminalną. Stanem startowym jest S — symbol początkowy gramatyki. Otrzymany automat jest na ogół niedeterministyczny (np. w sytuacji gdy $X \to aY$ i $X \to aZ$ są produkcjami gramatyki). Jeśli teraz

$$S \Rightarrow_G \alpha_1 Y_1 \Rightarrow_G \alpha_1\alpha_2 Y_2 \Rightarrow_G \ldots \Rightarrow_G \alpha_1\alpha_2\ldots\alpha_m Y_m \Rightarrow_G \alpha_1\alpha_2\ldots\alpha_m\alpha_{m+1}$$

jest wyprowadzeniem w G realizowanym przy pomocy stosownych produkcji $Y_i \to \alpha_{i+1} Y_{i+1}$, odpowiada mu jednoznacznie akceptująca ścieżka w grafie skonstruowanego automatu i podobnie na odwrót. □

Twierdzenie odwrotne — konstrukcję gramatyki odpowiadającej danemu automatowi, sformułujemy w równoważnej postaci dla automatu deterministycznego.

TWIERDZENIE 3.8 *Każdemu skończonemu automatowi deterministycznemu M odpowiada gramatyka regularna G taka, że $L(G) = L(M)$.*

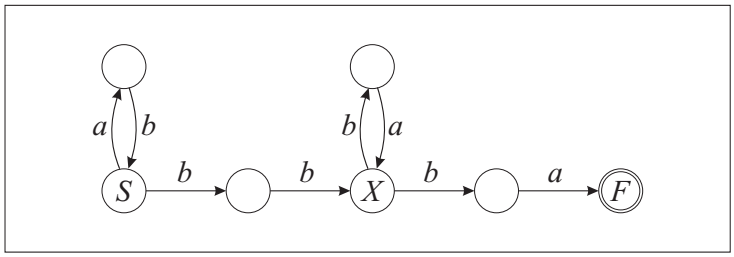
DOWÓD. Gramatykę konstruujemy następująco. W roli symboli nieterminalnych użyjemy bezpośrednio stanów automatu. W szczególności symbolem początkowym jest stan startowy S_0. Na podstawie wartości funkcji przejścia $\pi(S_i, a) = S_j$ tworzymy odpowiadające im prawostronnie liniowe produkcje gramatyki $S_i \to aS_j$. Ponadto dla każdego stanu końcowego S_k dodajemy produkcję $S_k \to \lambda$.

Niech teraz $\alpha = a_1 a_2 \ldots a_n \in L(M)$. Mamy więc ciąg relacji przejścia

$$S_0 a_1 a_2 \ldots a_n \Rightarrow_M a_1 S_{i_1} a_2 \ldots a_n \Rightarrow_M \cdots \Rightarrow_M a_1 a_2 \ldots a_n S_{i_n},$$

gdzie S_{i_n} jest stanem końcowym. Ciąg ten wzajemnie jednoznacznie odpowiada ciągowi wyprowadzeń w G

$$S_0 \Rightarrow_G a_1 S_{i_1} \Rightarrow_G \cdots \Rightarrow_G a_1 a_2 \ldots a_n S_{i_n} \Rightarrow_G a_1 a_2 \ldots a_n,$$



Rys. 3.17: Automat niedeterministyczny rozpoznający język zadany przez gramatykę (3.5) z Przykladu 3.13.

gdzie w ostatnim wyprowadzeniu skorzystaliśmy z produkcji $S_{i_n} \to \lambda$. Stąd $L(G) = L(M)$. □

PRZYKŁAD 3.13 Utworzymy gramatykę regularną dla automatu z Rys. 3.16 a. Symbole nieterminalne oznaczamy tak samo jak stany automatu S_{pp}, S_{np}, S_{pn} i S_{nn}. Symbolem początkowym jest S_{pp}. Postać produkcji ściśle imituje przejścia automatu:

$$\begin{aligned} S_{pp} &\to aS_{np} \mid bS_{pn} \\ S_{np} &\to aS_{pp} \mid bS_{nn} \\ S_{pn} &\to aS_{nn} \mid bS_{pp} \\ S_{nn} &\to aS_{pn} \mid bS_{np} \mid \lambda \end{aligned}$$

Ostatnią pustą produkcję dodaliśmy, ponieważ S_{nn} jest stanem finalnym automatu.

Równoważna gramatyka (dla tego samego języka) w nieco innej postaci pojawiła się już w Przykładzie 3.12. Można ją łatwo utworzyć na podstawie uogólnionego grafu przejścia z Rys. 3.16 c, taką samą jak powyższa metodą.

Z kolei zbudujemy automat dla języka generowanego przez następującą gramatykę

$$\begin{aligned} S &\to abS \mid bbX \\ X &\to baX \mid ba \end{aligned} \tag{3.5}$$

Nawiasem mówiąc, na podstawie tak prostej gramatyki jak (3.5) można łatwo napisać odpowiednie wyrażenie regularne, w tym wypadku $(ab)^*bb(ba)^*ba$. Automat budujemy zgodnie z przepisem podanym w dowodzie Twierdzenia 3.7, por. Rys. 3.17.

III.4 Własności języków regularnych

W poprzednich podrozdziałach omówiliśmy kilka równoważnych metod definiowania języków regularnych. Obecnie zajmiemy się pewnymi ich własnościami, istotnymi w praktycznych zastosowaniach.

1 Zamkniętość klasy języków regularnych

W Rozdziale II.3 zdefiniowaliśmy szereg operacji na językach. Gdy zajmujemy się analizą specyficznej klasy języków formalnych pojawia się naturalne pytanie o zamkniętość tej klasy ze względu na poszczególne operacje. A więc np. czy z faktu,

że L_1 i L_2 są językami klasy $\mathcal{L}$ automatycznie wnioskować można, że językiem tej samej klasy jest np. zbiór $L_1 \cap L_2$?

Pytania tego typu okazują się z reguły łatwo rozstrzygalne dla języków regularnych. W dalszej części skryptu przekonamy się, że w przypadku klas języków o bardziej skomplikowanej strukturze te same pytania są w wielu wypadkach nieporównanie trudniejsze.

Zacznijmy od rezultatu będącego prostą, bezpośrednią konsekwencją faktu, że języki regularne można opisać przy pomocy wyrażeń regularnych.

TWIERDZENIE 3.9 *Klasa $\mathcal{L}_3$ jest zamknięta ze względu na sumę mnogościową, konkatenację i domknięcie konkatenacyjne.*[6]

Wynika to wprost z Definicji 3.7 języka opisywanego przez wyrażenie regularne i faktu, że języki te są klasy $\mathcal{L}_3$. Z tych samych powodów można powiedzieć nawet więcej: rodzina języków regularnych nad ustalonym alfabetem A jest najmniejszą rodziną języków, która

— zawiera język $\emptyset$ oraz wszystkie języki jednoliterowe $\{a\}$ dla $a \in A$,

— jest zamknięta ze względu na skończone konkatenacje, sumy i domknięcie konkatenacyjne.

Okazuje się, że oprócz zaliczonej do operacji regularnych sumy, języki regularne są zamknięte również ze względu na wszelkie inne operacje mnogościowe.

TWIERDZENIE 3.10 *Klasa $\mathcal{L}_3$ jest zamknięta ze względu na dopełnienie, przekrój, różnicę i różnicę symetryczną.*

Dowód. Zamkniętość języków regularnych ze względu na dopełnienie, $L^c = \{\alpha \in A^* : \alpha \notin L\}$, otrzymujemy przez konstrukcję automatu deterministycznego M' dla L^c na podstawie automatu M dla języka L. Niech M akceptuje L. Dodając jeśli trzeba do M stan pułapkowy, możemy przyjąć, że funkcja przejścia π jest totalna. Automat M' powstaje z M przez zastąpienie w nim dotychczasowego zbioru stanów końcowych jego dopełnieniem, $S'_F = \Sigma - S_F$. A więc $s \in S'_F$ wtedy i tylko wtedy, gdy s nie jest stanem końcowym w M. Funkcja π pozostaje niezmieniona. Ponieważ jest ona funkcją totalną, każdy napis wejściowy czytany jest do końca: $s_0\alpha \Rightarrow^*_M \alpha s$, przy czym relacja przejścia $\Rightarrow$ jest identyczna dla obu automatów M i M'. Jeśli więc $\alpha \notin L$, wówczas $s \notin S_F$, skąd $\alpha \in L(M')$. Odwrotnie, gdy $\alpha \in L(M')$, wówczas $s \in S'_F$, a więc $s \notin S_F$ czyli $\alpha \notin L$. W rezultacie L^c jako język akceptowany przez SAD M' jest regularny.

Korzystając teraz z podstawowej tożsamości de Morgana $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$, wnioskujemy, że dla regularnych języków L_1 i L_2 także $L_1 \cap L_2$ musi być regularny. Stąd na podstawie $L_1 - L_2 = L_1 \cap L_2^c$ oraz $L_1 \oplus L_2 = (L_1 - L_2) \cup (L_2 - L_1)$, por. (1.1), otrzymujemy natychmiast identyczny wynik dla różnicy zwykłej i symetrycznej języków regularnych.

[6] Operacje te nazywamy nb. operacjami *regularnymi*.

Ostatni argument, jakkolwiek logicznie kompletny, ma jednak pewną wadę, bowiem nie podaje bezpośrednio metody konstrukcji skończonego automatu akceptującego język $L_1 \cap L_2$. Konstrukcja taka może okazać się pożyteczna przy rozwiązywaniu konkretnych zadań programistycznych. Opiszemy obecnie tę technikę.

Niech $M_1 = (A, \Sigma_1, s_0^{(1)}, S_F^{(1)}, \pi_1)$ oraz $M_2 = (A, \Sigma_2, s_0^{(2)}, S_F^{(2)}, \pi_2)$ będą automatami deterministycznymi akceptującymi L_1 i odpowiednio L_2. Dla prostoty zakładamy, że są to języki nad tym samym alfabetem A, w przeciwnym wypadku należałoby najpierw ograniczyć działanie automatów, a więc ich funkcje przejścia, do części wspólnej alfabetów. Tworzymy teraz nowy automat M, biorąc jako jego zbiór stanów iloczyn kartezjański $\Sigma = \Sigma_1 \times \Sigma_2$. Stanem początkowym jest $(s_0^{(1)}, s_0^{(2)})$, podobnie jak $S_F = S_F^{(1)} \times S_F^{(2)}$. Funkcja przejścia określona jest następująco:

$$\pi\Big((s_i^{(1)}, s_j^{(2)}), a\Big) \;=\; (s_k^{(1)}, s_l^{(2)}) \quad \Leftrightarrow \quad \pi_1(s_i^{(1)}, a) = s_k^{(1)} \quad \text{i} \quad \pi_2(s_i^{(2)}, a) = s_k^{(2)}\,. \tag{3.6}$$

Tak więc przejście z literą a ze stanu $(s_i^{(1)}, s_j^{(2)})$ do $(s_k^{(1)}, s_l^{(2)})$ możliwe jest w M dokładnie wtedy, gdy w M_1 istnieje takie przejście między $s_i^{(1)}$ a $s_k^{(1)}$ i jednocześnie w M_2 przejście takie istnieje między $s_j^{(2)}$ oraz $s_l^{(2)}$. W ten sposób każdemu ruchowi automatu M odpowiadają "równoległe" ruchy automatów M_1 i M_2 podczas skanowania tej samej litery. Nietrudno więc widzieć, że gdy M czytając słowo α osiąga pewien stan końcowy $(s_p^{(1)}, s_q^{(2)}) \in S_F^{(1)} \times S_F^{(2)}$, wówczas automaty M_1 i M_2 w serii równoległych ruchów na tym samym słowie wejściowym α osiągają swoje stany końcowe $s_p^{(1)}$ i $s_q^{(2)}$, a także na odwrót. Zatem akceptacja α przez M zachodzi dokładnie wtedy, gdy każdy z automatów M_1 i M_2 oddzielnie także akceptuje α. Język $L_1 \cap L_2$ jest więc akceptowany przez jawnie skonstruowany automat deterministyczny M, co kończy dowód. □

Prawo- i lewostronne ilorazy języków regularnych prowadzą także do języków tej samej klasy. Podobnie rzecz ma się z lustrzanym odbiciem i homomorfizmem języka regularnego. Dowody tych faktów pozostawiamy jako zadania do samodzielnego rozwiązania, p. Zad. 15.

2 Rozstrzygalność w klasie języków regularnych

W przykładach widzieliśmy, że ten sam język może być definiowany na wiele sposobów, przy czym równoważność takich opisów nie musi być widoczna na pierwszy rzut oka. Zasadne jest więc np. następujące pytanie: czy dwie gramatyki opisują ten sam język? Okazuje się, że w przypadku języków bardziej ogólnych niż języki regularne pytania tego typu są z reguły trudne lub wręcz nierozstrzygalne.[7] Klasa $\mathcal{L}_3$ jest w tym kontekście wyjątkiem: większość problemów decyzyjnych posiada tu dobre, konstruktywne rozwiązanie. Wymienimy dalej najważniejsze z nich. Mówiąc *opis języka* w pierwszej kolejności będziemy mieli na myśli opis przez SAD, lecz w ogólniejszym sformułowaniu może to być dowolna inna charakteryzacja języka regularnego: przez SAN, wyrażenie regularne lub gramatykę regularną. Pamiętajmy, że dla każdej z

[7] Nierozstrzygalność rozumiana jest tu jako brak ogólnego algorytmu, który w skończonym czasie odpowiedziałby "tak" lub "nie" na postawione pytanie. Na obecnym etapie posługujemy się tym pojęciem intuicyjnie, wkrótce jednak nadamy mu ścisły, formalny sens.

tych alternatywnych definicji języka potrafimy już zbudować w skończonej liczbie kroków równoważny automat deterministyczny.

Oto niektóre z ważniejszych problemów decyzyjnych:

a) **Problem należenia**: dla danego opisu języka L i słowa $\alpha \in A^*$ stwierdzić, czy $\alpha \in L$, czy też $\alpha \notin L$.

b) **Problem skończoności**: dla danego opisu języka L stwierdzić, czy jest on pusty, skończony, bądź nieskończony.

c) **Problem identyczności**: dla opisów dwóch języków L_1 i L_2 stwierdzić, czy $L_1 = L_2$.

d) **Problem inkluzji**: dla opisów dwóch języków L_1 i L_2 stwierdzić, czy $L_1 \subset L_2$.

Rozstrzygalność problemu należenia wynika z faktu, że na podstawie opisu języka regularnego jesteśmy w stanie wygenerować akceptujący go skończony automat deterministyczny, a następnie zasymulować jego działanie na danych wejściowych α. Odpowiedź "tak" lub "nie" zależy wyłącznie od tego w jakim stanie zatrzyma się automat.

Dla rozpoznania rozstrzygalności problemu skończoności najłatwiej posłużyć się grafową reprezentacją automatu deterministycznego akceptującego język L. Pytania o liczebność L tłumaczymy wówczas na rozstrzygalne pytania o istnienie określonych ścieżek w grafie. Jeśli istnieje w tym grafie przynajmniej jedna ścieżka z s_0 do któregokolwiek ze stanów końcowych, język L jest niepusty. W celu stwierdzenia czy L jest skończony czy też nie, należy przeanalizować strukturę cykli w grafie. Jeśli istnieje w nim choć jeden cykl prosty posiadający wspólną część ze ścieżką łączącą stan początkowy z końcowym, język L jest nieskończony.

Problem identyczności języków rozwiążemy zauważając, że skoro L_1 i L_2 są regularne, język $L = L_1 \oplus L_2$ też jest taki. Dla stwierdzenia czy $L_1 = L_2$ wystarczy zbadać, czy $L = \emptyset$. To zadanie jednak potrafimy już wykonać. Podobnie dla rozstrzygnięcia problemu inkluzji możemy posłużyć się formułą $L_1 \cup L_2 = L_2$ spełnioną jedynie wtedy, gdy istotnie $L_1 \subset L_2$. Tak więc problem inkluzji redukujemy jak poprzednio do pytania o to, czy język $L = (L_1 \cup L_2) \oplus L_2$ jest pusty. Podkreślmy raz jeszcze: metoda rozstrzygająca polega na skonstruowaniu automatu M akceptującego L i sprawdzeniu czy w jego grafie istnieje ścieżka z s_0 do $s \in S_F$. Samej konstrukcji M dokonujemy wyrażając różnicę symetryczną $L_1 \oplus L_2$ w jednej z równoważnych postaci (1.1) i systematycznie stosując opisane wyżej metody uzyskiwania automatów dla dopełnień oraz sum i/lub przekrojów języków regularnych.

3 Twierdzenie Myhilla–Nerode'a i lemat o pompowaniu

Charakteryzacje języków regularnych, które omówiliśmy do tej pory są z reguły mało użyteczne, gdy zadanie polega na wykazaniu, że pewien język nie należy do klasy $\mathcal{L}_3$. W tym wypadku należałoby bowiem udowodnić, że *nie istnieje* skończony automat, wyrażenie regularne lub gramatyka, które opisywałyby taki język. Jest to oczywiście zadanie wielokrotnie bardziej skomplikowane, niż proste stwierdzenie, że nie potrafimy skonstruować ani automatu ani też wyrażenia czy gramatyki dla badanego języka.

Dwa ważne twierdzenia, które omówimy w tym podrozdziale są bardziej odpowiednimi narzędziami do rozstrzygania tego typu problemów. Obydwa wykorzystują konsekwencje *skończoności* pamięci automatów (a więc skończoności zbioru stanów) do wykazania, że analizowany język nie może być regularny. Idea opiera się na prostym, intuicyjnym fakcie wykorzystywanym często w matematyce dyskretnej, opisowo nazywanym "zasadą gołębnika". Wyraża się ona stwierdzeniem, że jeśli w gołębniku znajduje się więcej gołębi niż jest w nim klatek, wówczas co najmniej jedną z klatek zajmują przynajmniej dwa gołębie. W roli klatek występują stany automatu, a w roli gołębi — napisy wejściowe lub ich poszczególne litery.

Przed właściwym sformułowaniem twierdzeń, zwróćmy uwagę na kilka prostych lecz istotnych faktów. Niech M będzie skończonym automatem deterministycznym o m stanach $s_0, \ldots s_{m-1}$ i totalnej funkcji przejścia. Przez M_i oznaczmy automat powstały z M przez zamianę stanu startowego z s_0 na s_i. W tej konwencji oczywiście $M = M_0$. Każdy z automatów definiuje pewien język $L_i = L(M_i)$. Języki L_i mogą (choć nie muszą) być różne, istotne jest jednak to, że powstać może w ten sposób nie więcej niż m różnych języków.

Z kolei wróćmy do definicji lewostronnej pochodnej języka (2.1). Ponieważ zamierzamy teraz używać wyłącznie lewostronnych pochodnych, dla czytelności pomijać będziemy indeks "minus" w mianowniku, pisząc $\frac{dL}{d\beta}$ zamiast $\frac{dL}{d\beta^-}$. Jeśli L jest językiem akceptowanym przez automat M, wówczas dla dowolnego ustalonego $\beta \in A^*$ zbiór $\frac{dL}{d\beta}$ jest po prostu jednym z języków L_i. Wystarczy mianowicie sprawdzić, do którego ze swoich stanów dochodzi M czytając napis β,

$$s_0\beta \;\Rightarrow_M^* \; \beta s_i\,.$$

Zgodnie z definicją pochodnej lewostronnej, $\frac{dL}{d\beta}$ jest zbiorem tych napisów α, które po dołączeniu do β produkują słowo $\beta\alpha$ akceptowane przez M. Jest to więc dokładnie zbiór tych α, które zostaną zaakceptowane przez M jeśli wystartować go ze stanu s_i, czyli $\alpha \in L_i$. Pochodne języka L obliczać można względem dowolnych napisów β, ważne jest jednak to, że w przypadku języka regularnego otrzymamy w ten sposób *nie więcej niż* m różnych wartości pochodnej. Zasada gołębnika działa tu w ten sposób, że jeśli wziąć więcej napisów β niż wynosi liczba stanów automatu M, co najmniej dwa z tych napisów produkują identyczną wartość pochodnej $\frac{dL}{d\beta}$.

Zauważmy także, że jeśli opuścić założenie o regularności języka L, liczba różnych jego pochodnych może okazać się nieskończona. Twierdzenie, które zamierzamy właśnie sformułować orzeka, że w tym wypadku liczba ta wręcz *musi* być nieskończona.

TWIERDZENIE 3.11 (Myhill–Nerode) *L jest regularny wtedy i tylko wtedy, gdy liczba jego różnych lewostronnych pochodnych $\frac{dL}{d\beta}$, $\beta \in A^*$, jest skończona.*

DOWÓD. Jeśli L jest regularny, posiada akceptujący go automat deterministyczny. Niech m będzie liczbą jego stanów. Uzasadniliśmy powyżej, że liczba różnych lewostronnych pochodnych L nie może przekraczać m, a więc dowód konieczności warunku Myhilla–Nerode'a jest już kompletny.

Odwrotnie, niech $L_0, L_1, \ldots L_{m-1}$ oznaczają wszystkie bez wyjątku przyjmowane wartości pochodnej $\frac{dL}{d\beta}$. Jest wśród nich sam język L, bowiem $\frac{dL}{d\lambda} = L$, co widać wprost z definicji pochodnej. Możemy przyjąć, że $L = L_0$. Zbudujemy skończony

automat deterministyczny dla L. Przyjmijmy jako etykiety stanów symbole s_i, $i = 0, 1, \ldots m-1$, a więc każdy ze stanów s_i odpowiada unikalnie jednemu językowi L_i. Funkcję przejścia definiujemy następująco:

$$\pi(s_i, a) = s_j \qquad \text{wtedy i tylko wtedy, gdy} \qquad \frac{dL_i}{da} = L_j\,.$$

Stanem startowym jest s_0 (odpowiadający językowi $L_0 = L$). Jako stany końcowe oznaczymy s_k dla tych k, dla których $\lambda \in L_k$, a więc L_k powstające gdy pochodna języka L jest obliczana względem jego własnych elementów $\beta \in L$. Pozostaje sprawdzić, że skonstruowany właśnie automat M rzeczywiście akceptuje język L.

Jeśli więc $\alpha = a_1 a_2 \ldots a_n \in L$, wówczas kolejno $a_2 \ldots a_n \in \frac{dL}{da_1} = L_{i_1}$, $a_3 \ldots a_n \in \frac{dL_{i_1}}{da_2} = L_{i_2}$ itd. aż do $a_n \in \frac{dL_{i_{n-2}}}{da_{n-1}} = L_{i_{n-1}}$ oraz ostatecznie $\lambda \in \frac{dL_{i_{n-1}}}{da_n} = L_{i_n}$. Tłumacząc to na przejścia w automacie

$$s_0\, a_1 \ldots a_n \;\Rightarrow\; a_1\, s_{i_1}\, a_2 \ldots a_n \;\Rightarrow \cdots \Rightarrow\; a_1 \ldots a_{n-1}\, s_{i_{n-1}}\, a_n \;\Rightarrow\; a_1 \ldots a_n\, s_{i_n}\,,$$

przy czym s_{i_n} jest stanem końcowym, ponieważ $\lambda \in L_{i_n}$. Zatem $\alpha \in L(M)$.

Niech z kolei $s_0\alpha \Rightarrow^* \alpha s_f$ dla $s_f \in S_F$. Zgodnie z definicją automatu oznacza to, że $L_f = \frac{dL}{d\alpha}$ oraz, że $\lambda \in L_f$. To z kolei świadczy o tym, że $\alpha \in L$, co kończy nasz dowód. □

Dodajmy jeszcze jeden użyteczny wniosek, który pojawia się jako swoisty bonus z konstrukcji automatu przedstawionej w dowodzie.

Fakt. Automat skonstruowany w dowodzie twierdzenia Myhilla–Nerode'a jest optymalny pod względem liczby stanów.

Uzasadnienie tego faktu jest bardzo proste jeśli zauważymy, że różne wartości L_i pochodnej $\frac{dL}{d\beta}$ są dla skonstruowanego automatu M dokładnie językami $L(M_i)$ akceptowanymi, gdy stan s_i pełni rolę stanu startowego. Stany s_i i s_j byłyby więc nierozróżnialne w sensie Definicji 3.5 dokładnie wtedy, gdy L_i i L_j byłyby identyczne, a to oznacza, że s_i i s_j to już jeden i ten sam stan automatu M.

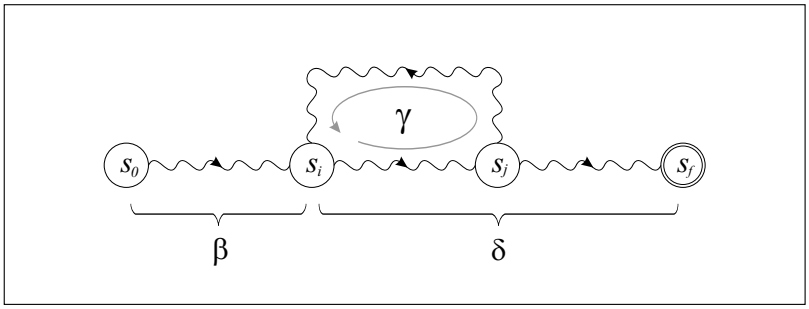
Przykład 3.14 Rozważmy dwa języki: $L = \{a^m b^n : \; m, n \geqslant 0\}$ oraz $L' = \{a^n b^n : \; n \geqslant 0\}$. Mimo formalnego podobieństwa, są to języki zupełnie różnych klas. Wykażemy najpierw przy pomocy twierdzenia Myhilla, że L jest regularny. Łatwo przekonać się, że

$$\frac{dL}{d\alpha} \;=\; \begin{cases} L_0 \;=\; L & \text{dla } \; \alpha = a^k, \;\; k \geqslant 0 \\ L_1 \;=\; \{b^n : \; n \geqslant 0\} & \text{dla } \; \alpha = a^k b^l, \;\; k \geqslant 0,\; l > 0 \\ L_2 \;=\; \emptyset & \text{dla pozostałych } \; \alpha \end{cases} \tag{3.7}$$

Tak więc mamy tylko 3 różne wartości pochodnej języka L, a zatem jest on regularny. Proponujemy czytelnikowi skonstruowanie 3-stanowego automatu dla tego języka na podstawie (3.7).

Z kolei dla języka L' wystarczy zauważyć, że np. biorąc jako α napisy postaci $a^n b$ dla $n \geqslant 1$ otrzymujemy jednoelementowe zbiory

$$\frac{dL'}{d\alpha} \;=\; \{b^{n-1}\}\,,$$



Rys. 3.18: Ilustracja do dowodu lematu o pompowaniu. Ponieważ długość wejściowego napisu $\alpha = \beta\gamma\delta$ jest nie mniejsza niż liczba stanów automatu, ścieżka w grafie odpowiadająca skanowaniu tego napisu musi odwiedzać jeden ze stanów co najmniej 2 razy, a więc zapętlać się.

a więc odmienną wartość dla każdego n. Zatem liczba różnych wartości pochodnej jest nieskończona, a język L' nie jest regularny.

Drugim ważnym narzędziem, służącym do rozpoznawania języków nieregularnych jest tzw. lemat o pompowaniu. W dalszej części skryptu spotkamy także analogiczny lemat dla języków bezkontekstowych.

LEMAT 3.1 (O pompowaniu dla jęz. regularnych) *Jeśli L jest nieskończonym językiem regularnym, istnieje stała $m > 0$ taka, że dowolne słowo $\alpha \in L$ o długości co najmniej m można zawsze podzielić na trzy części $\alpha = \beta\gamma\delta$ w taki sposób, że*

(i) $|\beta\gamma| \leqslant m$,

(ii) $|\gamma| \geqslant 1$,

(iii) wszystkie słowa postaci $\beta\gamma^k\delta$ dla $k = 0, 1, 2, \ldots$ są także elementami L.

DOWÓD. Sens tego niezbyt czytelnego na pierwszy rzut oka lematu stanie się bardziej zrozumiały, gdy omówimy ideę jego dowodu. Ograniczymy się tym razem jedynie do szkicowego przedstawienia argumentu, z pominięciem technicznych szczegółów dla maksymalnej jasności.

Jeśli więc L jest językiem regularnym, istnieje skończony automat deterministyczny M akceptujący go. Niech m będzie liczbą jego stanów. Ponieważ język L jest z założenia nieskończony, zawierać musi dowolnie długie słowa. Weźmy więc jakiekolwiek słowo $\alpha \in L$ o długości co najmniej m. Przeanalizujmy ścieżkę w grafie automatu M odpowiadającą sekwencji przejść akceptującej słowo α, por. Rys. 3.18.

Rozpoczynając w s_0, absorpcja pojedynczej litery w α przesuwa automat do kolejnego stanu. Jeśli założyć, że w procesie skanowania automat nie powraca do stanu s_0, do dyspozycji pozostaje nam $m - 1$ stanów $s_1, \ldots, s_{m-1}$. Skoro jednak α ma co najmniej m liter, zgodnie z zasadą gołębnika pewien stan musi być odwiedzony przynajmniej 2 razy. Załóżmy, że stanem który po raz pierwszy powtarza się w sekwencji przejść jest s_i. Oznaczmy symbolem β tę początkową część α, która została przeczytana do chwili, gdy automat po raz pierwszy dochodzi do stanu s_i, następnie symbolem γ tę dalszą część α, która podlega przetwarzaniu podczas cyklu

ruchów automatu od s_i do ponownego osiągnięcia s_i, a w końcu przez δ pozostałą część α skanowaną do osiągnięcia stanu końcowego s_f. Rys. 3.18 ilustruje ogólną sytuację.[8]

Skoro s_i jest z założenia pierwszym powtarzającym się stanem, wynika stąd, że długość części $\beta\gamma$ nie może być większa niż m, co pokazuje prawdziwość własności (i) w lemacie. Cykl z s_i do s_i zawiera co najmniej jedną krawędź, co uzasadnia nierówność (ii). Ponadto z rysunku wynika natychmiast, że jeśli jako napis wejściowy przyjąć $\alpha_0 = \beta\delta$, zostanie on zaakceptowany przez M. Podobnie dla napisu $\alpha_2 = \beta\gamma\gamma\delta$: wystarczy bowiem by automat wykonał dwa obroty pętli od s_i do s_i. Ogólnie każde słowo postaci $\alpha_k = \beta\gamma^k\delta$ dla $k = 0, 1, 2, \ldots$ musi być przez M zaakceptowane. Stanowi to dowód własności (iii). Stąd właśnie bierze się nazwa lematu: każde dostatecznie długie słowo regularnego języka musi dać się "napompować" przez powielanie pewnej jego części do napisów także należących do tego języka. □

Zanim przejdziemy do przykładów zwróćmy uwagę, że lemat o pompowaniu jest wynikiem istotnie słabszym od twierdzenia Myhilla–Nerode'a, ponieważ formułuje jedynie warunek konieczny dla języków regularnych. Znane są przykłady języków spełniających tezę tego lematu, które mimo to nie są regularne. Nie oznacza to jednak, że lemat jest mniej użyteczny od twierdzenia Myhilla. Przeważnie stosujemy go wykazując sprzeczność z przypuszczeniem, jakoby badany język L był regularny. Sprzeczność tą uzyskuje się podając choć jeden przykład słowa z L, które po napompowaniu tworzy napisy spoza L, niezależnie od jego podziału na części β, γ, δ zgodnie z warunkami (i) oraz (ii).

Przykład 3.15 Na początek zademonstrujemy zastosowanie lematu o pompowaniu do wykazania, że język $L' = \{a^n b^n : n \geqslant 0\}$ z Przykładu 3.14 nie jest regularny. Załóżmy, że mimo wszystko jest to język regularny. Zgodnie z lematem istnieje stała m taka, że *wszystkie* słowa z L' o długości większej niż m pozwalają na pompowanie niewyprowadzające poza L'. Weźmy więc słowo $\alpha = a^m b^m$. Jakkolwiek będziemy próbowali je podzielić na części β, γ, δ, warunek (i) lematu $|\beta\gamma| \leqslant m$ wymusza, że fragment $\beta\gamma$ składać się może wyłącznie z liter a. Tym samym $\gamma = a^p$ dla pewnego $p \geqslant 1$. Łatwo sprawdzić, że pompowanie produkuje wówczas słowa $\alpha_k = a^{n+(k-1)p} b^n$, $k = 0, 1, 2 \ldots$, które (poza przypadkiem $k = 1$, a więc wyjściowego słowa α) nie należą do języka L'. Stąd wniosek, że L' nie może być językiem regularnym.

Przeanalizujmy jeszcze raz krótko logiczną strukturę naszego wywodu. Warunek konieczny, który reprezentuje lemat o pompowaniu ma postać $p \Rightarrow q$, gdzie zdanie p orzeka, że język L jest regularny, natomiast q jest rozbudowanym warunkiem pompowania, który powinien zachodzić dla *wszystkich* wystarczająco długich słów α. W naszym rozumowaniu posługujemy się kontrapozycją lematu w postaci $\neg q \Rightarrow \neg p$. Zaprzeczenie q oznacza, że *istnieje przynajmniej jedno* słowo α, które choć zgodnie z wymogiem lematu ma długość przekraczającą m, generuje poprzez pompowanie

[8] Stan s_j może być tożsamy ze stanem s_i, co upraszcza nieco rysunek i podział α, lecz w ogólnym wypadku tak być nie musi — wówczas γ i δ mogą mieć identyczny początkowy fragment, ten który jest skanowany podczas przejścia z s_i do s_j.

słowa nienależące do L, niezależnie od tego, w jaki sposób zdecydujemy się je podzielić na części β, γ, δ. Jeśli więc jesteśmy w stanie wskazać przykład takiego słowa α, język L nie może być regularny. Kolejny przykład pokazuje, że właściwy wybór słowa α jest istotny dla skutecznego przeprowadzenia dowodu nieregularności L.

PRZYKŁAD 3.16 Tym razem niech $L = \left\{\alpha \in \{a,b\}^* : \ |\alpha|_a = |\alpha|_b\right\}$. Język L różni się od poprzednio rozważanego tym, że porządek liter a i b w słowach jest teraz najzupełniej dowolny. Załóżmy więc, że jest to język regularny i niech m oznacza stałą z lematu. Przekonamy się jak ważny jest odpowiedni wybór słowa α.

Jeśli jako α wybralibyśmy np. słowo $abab\ldots ab$, wówczas można by podzielić je tak, aby $\gamma = ab$, ale wtedy pompowanie dodawałoby jednakową liczbę liter a i b, produkując jedynie poprawne słowa z L. Cel nie zostałby więc osiągnięty. Podobnie każde słowo z L, które w obrębie początkowego m-literowego odcinka zawierałoby fragment o równej liczbie a i b nie spełniłoby oczekiwanego zadania, bowiem biorąc ten ostatni fragment jako γ, otrzymalibyśmy pompowanie niewyprowadzające poza język L.

Powyższe uwagi powinny już naprowadzić nas na właściwy wybór α: należy zagwarantować, by w jego początkowym m-literowym odcinku nie znalazł się podciąg o równej liczbie liter a i b. Jednym z takich słów jest $\alpha = a^m b^m$ (czytelnik zechce wskazać inne przykłady). Skoro lemat wymaga by $|\beta\gamma| \leqslant m$, wynika stąd, że γ składać się może wyłącznie z liter a, powiedzmy p z nich. Tym razem pompowanie $\alpha_k = a^{m+(k-1)p}\, b^m$ narusza równowagę między liczebnością a i b, a więc generuje słowa spoza języka L.

PRZYKŁAD 3.17 Ostatni z przykładów dotyczyć będzie języka $L = \{a^{n^2} : \ n \geqslant 0\}$. Zakładając, że jest on regularny, otrzymujemy stałą m, o której mowa w lemacie. Weźmy teraz słowo $\alpha = a^{m^2}$. Zauważmy, że następne dłuższe słowo z L, $a^{(m+1)^2}$, różni się od α o $2m+1$ liter, a więc L na pewno nie zawiera słów o jakiejkolwiek pośredniej długości $m^2 < l < m^2 + 2m + 1$. Jednak zgodnie z tezą lematu słowo α można przedstawić jako $\beta\gamma\delta$ w taki sposób, że $|\beta\gamma| \leqslant m$, a zatem $\gamma = a^p$ gdzie $1 \leqslant p \leqslant m$. Jednokrotne pompowanie słowa α generuje zatem $\alpha_2 = a^{m^2+p}$. Ponieważ jednak $m^2 < m^2 + p < (m+1)^2$, słowo α_2 nie może być elementem języka L. Nie jest on więc regularny.

III.5 Zadania do Rozdziału III

Zadanie 1

Skonstruować grafy skończonych automatów deterministycznych akceptujących języki nad alfabetem $A = \{a, b\}$:

a) $L = \{\omega : \ |\omega|_a = 2 \quad \text{i} \quad |\omega|_b > 2\}$

b) $L = \{\omega : \ |\omega| \mod 3 = 0\}$

c) $L = \{\omega : \ |\omega|_a \mod 3 = 1\}$

d) $L = \{\omega : \ (|\omega|_a - |\omega|_b) \mod 3 \geqslant 1\}$

e) $L = \{\omega : \ (|\omega|_a + 2|\omega|_b) \mod 3 < 2\}$

f) $L = \{a^k : \ k \geqslant 0 \quad \text{i} \quad k \neq 4\}$

g) $L = \{a^k : \ 3 \mid k \quad \text{lub} \quad 5 \mid k\}$ ($p \mid k$ oznacza, że p jest podzielnikiem k)

h) $L = \{a^k : \ 3 \mid k \quad \text{i} \quad 5 \nmid k\}$

i) $L = \{a^n : \ n > 0 \quad \text{i} \quad 6 \mid n(n+1)\}$

Zadanie 2

Segmentem w słowie ω nazywamy zawarty w nim podciąg złożony z co najmniej dwóch sąsiednich, identycznych liter. Np. słowo *aabaaaaabbb* zawiera segmenty *aa*, *aaaaa* oraz *bbb*. Zbudować SAD dla następujących języków nad $A = \{a, b\}$:

a) $L = \{\omega : \ \omega$ nie zawiera segmentów o długości mniejszej niż $4\}$

b) $L = \{\omega : \ $ każdy segment liter a w ω ma długość 2 lub $3\}$

c) $L = \{\omega : \ \omega$ zawiera co najwyżej 2 segmenty liter a długości $3\}$

d) $L = \{\omega : \ \omega$ zawiera po 1 segmencie liter a i b długości $3\}$

Zadanie 3

Zbudować SAD dla języka nad $\{0, 1\}$ złożonego z zapisanych w postaci binarnej liczb podzielnych przez 3 lub przez 5.

Zadanie 4

Dla języka z poprzedniego zadania zbudować automat niedeterministyczny o mniejszej liczbie stanów niż w rozwiązaniu deterministycznym.

Zadanie 5

Skonstruować automat niedeterministyczny akceptujący język

$$L = \{a^{2n+1}b : \ n \geqslant 0\} \cup \{a^{2n}bb : \ n \geqslant 1\}.$$

Zadanie 6

Znaleźć wyrażenia regularne generujące języki:

a) $L = \{a^n b^m : \ 2 \mid (n+m)\}$

b) $L = \{a^n b^m : \ n \geqslant 4, \ m \leqslant 3\}$

c) $L^c = A^* - L$ dla L z Zad. b)

d) $L = \{a^n b^m : \ n \geqslant 1, \ m \geqslant 1 \ nm \geqslant 3\}$

Następnie skonstruować automaty niedeterministyczne odpowiadające utworzonym wyrażeniom.

Zadanie 7

Zbudować automaty niedeterministyczne dla języków zadanych przez wyrażenia regularne:

a) $ab^*aa + bba^*aab$

b) $aa^*(a + b)$

c) $((aaa^*)^*b)^*$

d) $(aab)^* + (aa^* + bb)^*$

Zadanie 8

Zamienić automaty skonstruowane w poprzednich dwóch zadaniach na równoważne automaty deterministyczne.

Zadanie 9

Przeprowadzić procedury optymalizacji dla automatów uzyskanych w poprzednim zadaniu.

Zadanie 10

Metodą redukcji SAD skonstruować wyrażenia regularne generujące języki z Zad. 1.

Zadanie 11

Zaproponować ogólną metodę zamiany wyrażenia regularnego ρ na wyrażenie $\overleftarrow{\rho}$ generujące lustrzane odbicie języka $L(\rho)$.

Zadanie 12

Wykazać prawdziwość lub nieprawdziwość następujących tożsamości dla wyrażeń regularnych:

a) $\rho + \emptyset = \rho\,, \quad \rho\emptyset = \emptyset\rho = \emptyset\,, \quad \emptyset^* = \lambda$

b) $\rho\lambda = \lambda\rho = \rho\,, \quad \lambda^* = \lambda$

c) $(\rho^*)^* = \rho^*\,, \quad (\rho + \sigma)^* = \rho^* + \sigma^*\,, \quad (\rho\sigma)^* = \rho^*\sigma^*$

d) $\rho^*(\rho + \sigma)^* = (\rho + \sigma)^*\,, \quad (\rho + \sigma)^* = (\rho^*\sigma^*)^*$

Zadanie 13

Znaleźć gramatyki regularne generujące języki:

a) $L(aa^*(ab + a)^*)$

b) $L((aab^*ab)^*)$

c) $L((aab)^* + (aa^* + bb)^*)$

d) $L = \{\omega : \ |\omega|_a \leqslant 3\}$

e) $L = \{a^n b^m : \ 2|(n + m)\}$

f) $L = \{\omega : \ |\omega|_a + 3|\omega|_b \ \text{ jest parzyste}\}$

g) $L = \{\omega : \ |\omega|_a \ \text{i} \ |\omega|_b \ \text{ są parzyste}\}$

h) $L = \{\omega : \ (|\omega|_a - |\omega|_b) \mod 3 = 1\}$

i) $L = \{\omega : \quad \big||\omega|_a - |\omega|_b\big| \text{ jest parzyste}\}$

j) $L = \{a^n b^{n \mod 3} : \quad n \geqslant 0\}$

Zadanie 14

Znaleźć skończony automat niedeterministyczny akceptujący język generowany przez gramatykę

$$\begin{aligned} S &\to abX \\ X &\to baY \\ Y &\to aX \mid bb \end{aligned}$$

Zadanie 15

Niech L_1 i L_2 będą językami regularnymi. Konstruując odpowiednie automaty skończone pokazać, że regularne są także

a) ilorazy prawo- i lewostonne $L = L_1/L_2$ oraz $L = L_1 \setminus L_2$

b) lustrzane odbicie $L = \overleftarrow{L_1}$

c) dowolny obraz homomorficzny $L = h(L_1)$.

Zadanie 16

Funkcja $\chi(a) = L_a$, $a \in A$, $L_a \subseteq B_a^*$ indukuje podstawienie $\chi(L)$ dla języka $L \subseteq A^*$, por. (2.2) str. 31. Pokazać, że jeśli L oraz wszystkie języki L_a są regularne, regularny jest także język $\chi(L)$. (Wskazówka: wykorzystać reprezentacje L oraz L_a przez wyrażenia regularne).

Zauważmy, że z zamkniętości klasy $\mathcal{L}_3$ ze względu na regularne podstawienia wynika zbiorczo szereg podobnych, szczegółowych rezultatów, których odrębne dowody przytoczyliśmy w podrozdziale III.4.1.

Zadanie 17

Określmy operację $\min L$ dla języka L jako

$$\min L \;=\; \{\omega \in L : \quad \text{nie istnieją } \alpha \in L \text{ oraz } \beta \in A^* \text{ takie, że } \; \omega = \alpha\beta\}\,.$$

Pokazać, że dla języka regularnego $\min L$ także jest językiem regularnym.

Zadanie 18

Skonstruować automaty dla przekrojów języków:

a) $L((a+b)a^*) \cap L(baa^*)$

b) $L(ab^*a^*) \cap L(a^*b^*a)$

Zadanie 19

Które z poniższych stwierdzeń są prawdziwe dla dowolnych języków regularnych i homomorfizmów?

a) $h(L_1 \cup L_2) = h(L_1) \cup h(L_2)$

b) $h(L_1 \cap L_2) = h(L_1) \cap h(L_2)$

c) $h(L_1 L_2) = h(L_1)h(L_2)$

Zadanie 20

Niech $L_1 = L(a^*baa^*)$, $L_2 = L(aba^*)$. Wyznaczyć L_1/L_2.

Zadanie 21

Pokazać, że własność $L_1 = L_1 L_2 / L_2$ nie jest na ogół prawdziwa.

Zadanie 22

Niech $L_1 \cup L_2$ będzie językiem regularnym, a L_1 skończonym. Czy wynika stąd, że L_2 jest regularny?

Zadanie 23

Zaproponować algorytmy sprawdzające czy

a) $|L| \geqslant 5$ dla dowolnego regularnego języka L

b) dany język regularny L zawiera nieskończenie wiele słów parzystej długości

c) dla zadanej gramatyki regularnej G czy $L(G) = \{a, b\}^*$

Zadanie 24

Wyznaczyć wszystkie wartości lewostronnych pochodnych $\frac{dL}{d\alpha}$ dla języków z Zadania 6.

Zadanie 25

Wyznaczyć wszystkie wartości lewostronnych pochodnych $\frac{dL}{d\alpha}$ dla języka z Przykładu 3.2.

Zadanie 26

Pokazać, że następujące języki nie są regularne:

a) $L = \{a^n b^m : \quad n \leqslant m\}$

b) $L = \{\omega : \quad |\omega|_a \neq |\omega|_b\}$

c) $L = \{\omega\omega : \quad \omega \in A^*\}$

Zadanie wykonać dwukrotnie używając 1) twierdzenia Myhilla-Nerode'a oraz 2) lematu o pompowaniu.

Zadanie 27

Rozstrzygnąć (dowolną metodą) czy poniższe języki są regularne, czy też nie:

a) $L = \{a^n : \quad n \geqslant 2 \text{ jest liczbą pierwszą}\}$

b) $L = \{a^n : \ n \text{ nie jest liczbą pierwszą}\}$

c) $L = \{a^m b^n : \ 6 \mid mn\}$

c) $L = \{a^n : \ n = k^3, \ k \geqslant 0\}$

d) $L = \{a^n : \ n = 2^k, \ k \geqslant 0\}$

e) $L = \{a^n : \ n \text{ jest iloczynem dwóch liczb pierwszych}\}$

f) $L = \{a^{n!} : \ n \geqslant 1\}$

Rozdział IV

Gramatyki i języki bezkontekstowe

IV.1 Gramatyki bezkontekstowe

W poprzednim rozdziale spotkaliśmy się z przykładami języków nieregularnych, np. $L = \{a^n b^n : \ n \geqslant 0\}$ lub $L = \{a^{n^2} : \ n \geqslant 0\}$. Niektóre z nich należą do bardziej ogólnej kategorii — klasy tzw. języków bezkontekstowych, której opisem zajmiemy się obecnie. Języki bezkontekstowe będziemy charakteryzować przede wszystkim przez odpowiednie gramatyki, a także przez automaty o konstrukcji bardziej ogólnej niż skończone automaty deterministyczne. Będą one wyposażone w pamięć roboczą działającą na zasadzie stosu. Klasa języków bezkontekstowych oznaczana jest w hierarchii Chomsky'ego symbolem $\mathcal{L}_2$, czasem także symbolem $\mathcal{L}_{\mathrm{Cf}}$. Skrót "Cf" pochodzi od angielskich słów "context free". W dalszej części rozdziału zbadamy, które z operacji na językach nie wyprowadzają poza klasę $\mathcal{L}_2$ oraz omówimy bezkontekstową wersję lematu o pompowaniu, która pozwoli nam na wskazanie przykładów języków spoza $\mathcal{L}_2$.

Zacznijmy od definicji gramatyki bezkontekstowej.

DEFINICJA 4.1 *Gramatykę $G = (A, V, S, \Pi)$ nazywamy bezkontekstową wtedy i tylko wtedy, gdy wszystkie jej produkcje są postaci*

$$X \ \to \ \alpha \,, \tag{4.1}$$

gdzie $X \in V$ oraz $\alpha \in (A \cup V)^*$. Język L nazywamy bezkontekstowym, gdy istnieje generująca go gramatyka bezkontekstowa G, a więc gdy $L = L(G)$. Klasę języków bezkontekstowych oznaczamy symbolem $\mathcal{L}_2$.

Relacja wyprowadzenia $\Rightarrow_G^*$ w gramatyce bezkontekstowej określona jest tak samo jak w ogólnym przypadku, zgodnie z Definicją 2.5. Postać produkcji ma bezpośredni związek z terminem "bezkontekstowa". Pozwala ona bowiem na bezwarunkowe zastępowanie symboli nieterminalnych X odpowiednimi napisami α wszędzie, gdzie tylko się pojawią, niezależnie od "kontekstu" symboli otaczających je w formach zdaniowych gramatyki. Inaczej rzecz ma się z gramatykami ogólniejszego typu, tzw. gramatykami kontekstowymi, których produkcje muszą mieć postać $\beta_1 X \beta_2 \to \beta_1 \alpha \beta_2$. Tu symbol X może być zastąpiony napisem α jedynie w wówczas, gdy występuje *w kontekście* β_1, β_2.

1 Gramatyki i wyprowadzenia

Przyjrzymy się konstrukcji gramatyk bezkontekstowych na podstawie kilku przykładów. Chcemy zwrócić uwagę czytelnika na fakt, iż na ogół słowa w $L(G)$ można wyprowadzić w gramatyce na wiele alternatywnych sposobów. Z punktu widzenia zastosowań programistycznych można to uznać za przeszkodę — wolelibyśmy by sposób generowania słów był możliwie jednoznaczny na każdym etapie.

PRZYKŁAD 4.1 W Przykładzie 2.5 opisaliśmy gramatykę bezkontekstową generującą $L_1 = \{a^n b^n : \ n \geqslant 0\}$. Rozważmy powiązany język $L = \{a^m b^n : \ m \neq n\}$. Zwróćmy przy tym uwagę, że nie jest to dopełnienie języka L_1. By zbudować gramatykę dla L, rozłóżmy warunek $m \neq n$ na dwa przypadki: $m < n$ i $m > n$. Chcąc wyprowadzić słowa $a^m b^n$, w których $m < n$, wygenerujmy najpierw $a^m b^m$, by następnie rozszerzyć je o co najmniej 1 literę b. Rozwiązaniem jest gramatyka o produkcjach

$$\begin{aligned} S &\to S_1 B \\ S_1 &\to aS_1 b \mid \lambda \\ B &\to bB \mid b \end{aligned}$$

W bliźniaczy sposób uzyskamy gramatykę generującą słowa $a^m b^n$ dla $m > n$, by ostatecznie połączyć obydwa warianty:

$$\begin{aligned} S &\to AS_1 \mid S_1 B \\ S_1 &\to aS_1 b \mid \lambda \\ A &\to aA \mid a \\ B &\to bB \mid b\,. \end{aligned}$$

Zauważmy, że w pierwszym kroku wyprowadzenia, gdy wybieramy $S \Rightarrow AS_1$ lub $S \Rightarrow S_1 B$, rozstrzygamy nieodwracalnie, który z przypadków $m > n$ lub $m < n$ będzie realizowany. Dalsze wyprowadzenie zawiera dwie niezależne od siebie iteracje: jedną, bazującą na produkcji $S_1 \to aS_1 b$, generującą równą liczbę a i b, oraz drugą zastępującą nieterminal A lub B ciągiem dodatkowych liter a lub b. Kolejność, w jakiej w wyprowadzeniu pojawiają się te iteracje nie ma wpływu na kształt końcowego napisu terminalnego.

PRZYKŁAD 4.2 Drugi z języków analizowanych w Przykładzie 2.5 miał postać $L_2 = \{\omega \in A^* : \ |\omega|_a = |\omega|_b\}$. Jeśli z jego gramatyki usuniemy produkcję $S \to bSa$ (redukując ich zbiór do $S \to \lambda \mid aSb \mid SS$), otrzymamy nowy język L będący podzbiorem L_2. Słowa w L zawierają jak poprzednio po równo liter a i b, różnica w stosunku do L_2 polega jednak na tym, że funkcja Δ określona w (2.3) nigdy nie przyjmuje na nich wartości ujemnych. Innymi słowy, licząc od lewej, liczba liter a nigdy nie będzie mniejsza niż liczba b. Język ten okaże się bardziej znajomy, jeśli zamiast a użyjemy znaku nawiasu otwierającego "(", a zamiast b zamykającego ")". L jest bowiem językiem "poprawnie zagnieżdżonych nawiasów". Przykładem słowa posiadającego wyprowadzenie z S w tej gramatyce jest np. (()())(), podczas gdy

()())(() nie posiada takiego wyprowadzenia. Mamy

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow (SS)S \Rightarrow ((S)S)S \Rightarrow (()S)S \Rightarrow (()(S))S \Rightarrow (()())S \\ \Rightarrow (()())(S) \Rightarrow (()())() \,.$$

To samo słowo końcowe można uzyskać zmieniając kolejność zastępowania symboli S w pośrednich formach zdaniowych, np. w $(SS)S$. W wyprowadzeniu powyżej przyjęliśmy, że w kolejnym kroku zastępowaniu podlega zawsze pierwszy od lewej symbol nieterminalny. Równie dobrze mógłby to być zawsze ostatni z nieterminali w formie zdaniowej,

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow S() \Rightarrow (S)() \Rightarrow (SS)() \Rightarrow (S(S))() \Rightarrow (S())() \\ \Rightarrow ((S))() \Rightarrow (()())() \,,$$

lub też nieterminal wybierany w dowolny inny, mniej systematyczny sposób. Istotne jest jednak użycie w pierwszym kroku produkcji $S \to SS$ jeśli końcowa struktura napisu ma zawierać na najwyższym poziomie 2 lub więcej równoległych par nawiasów $(\ldots)(\ldots)$. Jeśli zaś w pierwszym kroku wyprowadzenia użyjemy produkcji $S \to (S)$, struktura końcowego napisu będzie miała postać $(\ldots)$.

Tak więc, podobnie jak w poprzednim przykładzie, wczesne etapy w wyprowadzeniu przesądzają końcowy kształt generowanego słowa.

Przykład 4.3 Kolejnym przykładem jest gramatyka postaci

$$\begin{aligned} S &\to S+S \mid S*S \mid X \\ X &\to x \mid y \mid z \mid (S) \end{aligned}$$

generująca język złożony z wyrażeń arytmetycznych w ograniczonej postaci, np. $x*(y+z)$ lub $x+y*z+z$. Jedno z możliwych wyprowadzeń pierwszej formuły ma postać

$$S \Rightarrow S*S \Rightarrow X*S \Rightarrow x*S \Rightarrow x*X \Rightarrow x*(S) \Rightarrow x*(S+S) \\ \Rightarrow x*(X+S) \Rightarrow x*(y+S) \Rightarrow x*(y+X) \Rightarrow x*(y+z) \,. \tag{4.2}$$

Inne wyprowadzenia uzyskamy zmieniając kolejność zastępowania nieterminali w pośrednich formach zdaniowych. Jednak jak poprzednio, istotne jest użycie produkcji $S \to S*S$ w pierwszym kroku. Czytelnik zechce przekonać się, że jeśli rozpoczniemy wyprowadzenie od produkcji $S \to S+S$, nie zdołamy wyprowadzić formuły $x*(y+z)$

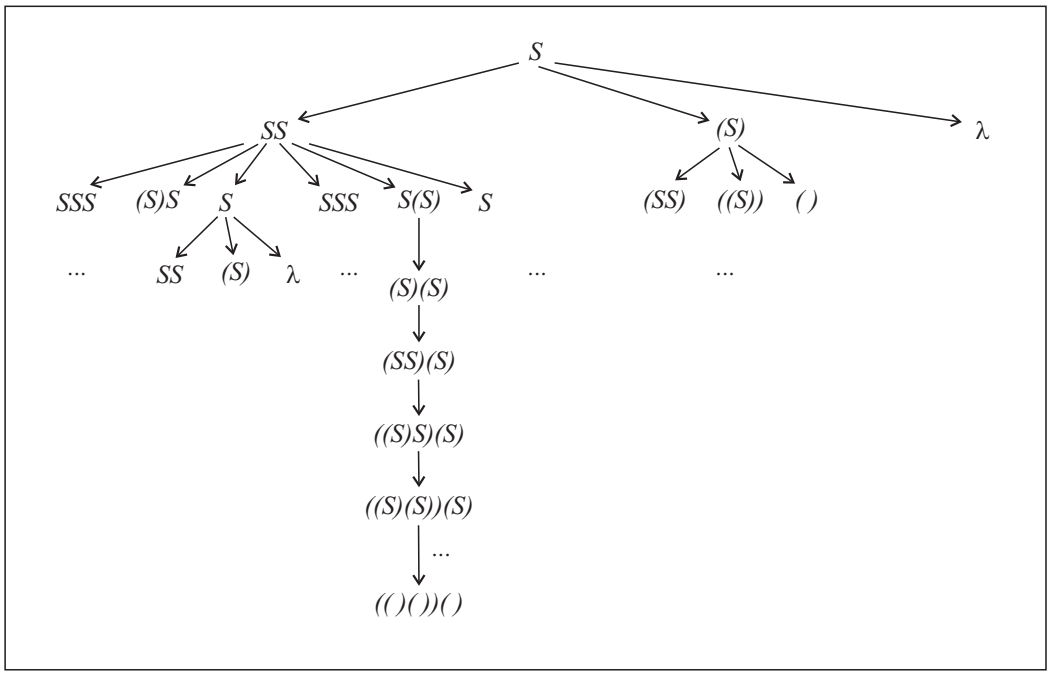
Przeanalizujmy z kolei początkowe fragmenty dwóch spośród wielu możliwych wyprowadzeń wyrażenia $x+y*z+z$:

$$S \Rightarrow S+S \Rightarrow S+S*S \Rightarrow S+S*S+S \Rightarrow \ldots \tag{4.3}$$

oraz

$$S \Rightarrow S*S \Rightarrow S*S+S \Rightarrow S+S*S+S \Rightarrow \ldots \tag{4.4}$$

Widzimy więc, że ten sam napis końcowy można niekiedy wyprowadzić zupełnie różnymi drogami, niezależnie od kolejności użycia produkcji nawet w początkowych krokach wyprowadzenia.



Rys. 4.1: Drzewo wyprowadzeń w gramatyce "poprawnie rozstawionych nawiasów", $S \to SS \mid (S) \mid \lambda$.

Przykład 4.4 W końcu rozważmy język postaci $L = \{a^m b^n c^{m+n} : \ m, n \geqslant 0\}$. Generuje go gramatyka z produkcjami

$$\begin{aligned} S &\to aSc \mid X \\ X &\to bXc \mid \lambda . \end{aligned}$$

Wszystkie wyprowadzenia w tej gramatyce mają tylko jedną możliwą postać:

$$S \Rightarrow aSc \Rightarrow^* a^m S c^m \Rightarrow a^m X c^m \Rightarrow a^m b X c c^m \Rightarrow^* a^m b^n X c^{m+n} \Rightarrow a^m b^n c^{m+n} .$$

Jest to po części konsekwencją faktu, że tym razem gramatyka jest liniowa, por. Def. 3.8), a więc że w każdej pośredniej formie zdaniowej występuje tylko jeden nieterminal.

Spróbujmy usystematyzować nasze spostrzeżenia. Rys. 4.1 przedstawia częściowe drzewo możliwych wyprowadzeń w gramatyce z Przykładu 4.2. W węzłach drzewa znajdują się formy zdaniowe, a ich potomkami są wszystkie kolejne formy, które można z nich wyprowadzić przy użyciu produkcji gramatyki. Niektóre z tych form mogą się powtarzać, jeśli istnieje więcej niż jedna droga ich uzyskania. Np. forma SSS występująca dwukrotnie w węzłach 3-ciego piętra może być wyprowadzona z SS na drugim poziomie przez zamianę pierwszego lub też drugiego symbolu S na SS. Drzewo ma za zadanie dokumentować *wszystkie* możliwe ciągi wyprowadzeń zgodne z produkcjami gramatyki. Oto najważniejsze z obserwacji:

— Ten sam napis terminalny może pojawić się w wielu liściach drzewa, ponieważ na ogół istnieją alternatywne drogi wyprowadzenia go z symbolu S.

— Dla niektórych słów języka można przewidzieć, że nie pojawią się one wśród liści określonego poddrzewa. Np. na Rys. 4.1 napis (()())() nie pojawi się w poddrzewie rozpoczynającym się w (S), widać to jednak dopiero na podstawie jego dwóch ostatnich znaków. Podobnie dla języka z Przykładu 4.1, decyzja, w którym poddrzewie należałoby szukać słowa $a^m b^n$ podjęta może być dopiero po przeczytaniu całego słowa i stwierdzeniu czy $m > n$, czy odwrotnie.

— Możliwe jest rekurencyjne pojawianie się tej samej formy na różnych piętrach drzewa, a więc "zapętlonych" wyprowadzeń, takich jak $S \Rightarrow SS \Rightarrow S \Rightarrow \ldots$. Jest to możliwe, gdy gramatyka zawiera puste produkcje $X \to \lambda$.

Zwróćmy uwagę na istotną różnicę miedzy opisem języka przy pomocy automatu w stosunku do jego charakteryzacji przez gramatykę. Automat jest urządzeniem analitycznym, podejmującym decyzje na podstawie wczytywanych liter badanego napisu, a jego kolejne stany w pewnym sensie uznać można za pamięć przechowującą wzorzec przeskanowanego dotąd fragmentu. Natomiast gramatyka jest narzędziem *generatywnym*, które rozpoczynając od najbardziej ogólnego wzorca, reprezentowanego przez symbol startowy S, rozwija go stopniowo aż do uzyskania terminalnego napisu. Odwołując się do obrazu drzewa wyprowadzeń mówimy, że opis gramatyczny języka jest typu "top–down", generowanie słów następuje bowiem od korzenia w dół do liści, natomiast opis przez automaty jest typu "bottom-up", od liścia z wejściowym napisem terminalnym do korzenia reprezentującego stan finalny. Jeśli więc zadanie postawione jest w typowej postaci:

$$\textit{mając dane słowo } \alpha \in A^*, \textit{ zdecydować czy } \alpha \in L\,,$$

automat może wydawać się na ogół lepszym narzędziem, bowiem algorytm bazujący na opisie gramatycznym musi "odszukać" w drzewie właściwą ścieżkę wyprowadzenia $S \Rightarrow^* \alpha$, jeśli taka istnieje. Istnienie wielu różnych wyprowadzeń dla tego samego słowa sugeruje, że mamy przy tym do czynienia z niedeterminizmem, a zatem należy spodziewać się kosztownego obliczeniowo, systematycznego przeglądu drzewa. Ponadto możliwość pojawiania się zapętlonych wyprowadzeń jest dodatkowym utrudnieniem, które bez specjalnych zabezpieczeń w algorytmie mogłoby prowadzić do powstania martwych pętli.

Oszacujmy złożoność przeszukiwania drzewa wyprowadzeń, które musiałby wykonać naiwny algorytm rozstrzygający czy $\alpha \in L(G)$. Założymy przy tym, że gramatyka G nie zawiera pustych produkcji ani też produkcji jednostkowych,[1] tj. produkcji postaci $X \to Y$. Dzięki temu wszystkie produkcje $X \to \gamma$ spełniają $|\gamma| > 1$, za wyjątkiem co najwyżej produkcji końcowych $X \to a$, $a \in A$, eliminujących nieterminal z formy zdaniowej, a więc ograniczających dalsze rozgałęzianie się drzewa. Warunek $|\gamma| > 1$ powoduje, że wyprowadzenia stają się *nieskracające*, a więc generują monotoniczny wzrost długości form zdaniowych. Niech p oznacza liczbę produkcji gramatyki. Pierwsze piętro drzewa wyprowadzeń zawiera zatem co najwyżej p elementów, drugie — co najwyżej p^2 itd. Ze względu na monotoniczność wyprowadzeń, każde słowo α wyprowadzane z S, o długości nie przekraczającej n, znajdzie się w drzewie nie później niż na jego $2n$-tym piętrze (potrzeba bowiem maksymalnie n

[1] Przekonamy się niebawem, że każdą gramatykę bezkontekstową można przekształcić do równoważnej postaci bez pustych lub jednostkowych produkcji.

kroków wyprowadzenia by wygenerować n-znakową formę zdaniową oraz n podstawień $X \to a$ eliminujących z niej nieterminale). W rezultacie algorytm ekstensywnie przeglądający drzewo wyprowadzeń w poszukiwaniu napisu α musiałby odwiedzić maksymalnie

$$1 + p + p^2 + \cdots + p^{2n} \;=\; O(p^{2n+1}) \tag{4.5}$$

węzłów. Przeszukanie to byłoby zatem obliczeniowo bardzo kosztowne.

W dalszej części tego rozdziału zajmiemy się rozpoznaniem niepożądanych własności gramatyk bezkontekstowych i metodami ich eliminowania. W rezultacie będziemy w stanie zredukować koszt obliczeniowy rozpoznawania słów języka bezkontekstowego do rzędu $O(|\alpha|^3)$.

2 Niejednoznaczność gramatyk i języków

Dążąc do usystematyzowania wyprowadzeń w gramatyce bezkontekstowej G, definiujemy tzw. wyprowadzenia jednostronne. Używaliśmy ich już w Przykładzie 4.2.

DEFINICJA 4.2 *Wyprowadzenie $\alpha \Rightarrow_G^* \beta$ nazywamy lewostronnym (odp. prawostronnym) jeśli we wszystkich jego krokach zastępowaniu podlega pierwszy od lewej (odp. od prawej) symbol nieterminalny.*

Wyprowadzenia jednostronne porządkują w pewnym stopniu nasze działania, jednak wciąż pozostaje ewentualność, iż słowo może posiadać więcej niż jedno takie wyprowadzenie. Wróćmy bowiem do Przykładu 4.3: napis $x + y * z + z$ posiada dwa całkiem różne wyprowadzenia lewostronne, rozpoczynające się odpowiednio od $S \Rightarrow S + S$ oraz $S \Rightarrow S * S$, podobnie jak w (4.3) i (4.4). Z drugiej strony słowo $x * (y + z)$ posiada tylko jedno lewostronne wyprowadzenie (4.2).

Okazuje się jednak, że gramatykę można w tym wypadku skorygować. Oto równoważna postać:

$$\begin{aligned} S &\to T \mid T + S \\ T &\to F \mid F * T \\ F &\to x \mid y \mid z \mid (S) \end{aligned} \tag{4.6}$$

Tym razem istnieje już tylko jedno wyprowadzenie lewostronne dla $x + y * z + z$:

$$\begin{aligned} S &\Rightarrow T + S \Rightarrow F + S \Rightarrow x + S \Rightarrow x + T + S \Rightarrow x + F * T + S \\ &\Rightarrow x + y * T + S \Rightarrow x + y * F + S \Rightarrow x + y * z + S \Rightarrow x + y * z + T \\ &\Rightarrow x + y * z + F \Rightarrow x + y * z + z\,. \end{aligned}$$

Ta postać gramatyki ma jeszcze jedną zaletę: odzwierciedla ona naturalny porządek operacji arytmetycznych, w którym mnożenie bierze pierwszeństwo przed dodawaniem. Symbol T reprezentujący składnik poprzedza w wyprowadzeniu symbol F reprezentujący czynnik. Składnik T może co prawda pojawić się w kolejnych piętrach drzewa, jednak pod warunkiem, że będzie ujęty w nawiasy w rezultacie użycia produkcji $F \to (S)$. W tym sensie wyprowadzenie zawiera informację *semantyczną*: czynniki i ich iloczyny redukują się do składników, a więc muszą być obliczone wcześniej, natomiast wyrażenia o strukturze sumy, jeśli mają redukować się do czynników,

wymagają nawiasów wymuszających ich obliczenie w pierwszej kolejności. Z drugiej strony wyprowadzenie $S \Rightarrow S * S \Rightarrow^* S + S * S + S \Rightarrow \ldots$ implikuję niepoprawną interpretację końcowego wyrażenia jako $(x + y) * (z + z)$.

Rozróżnienie opisane powyżej prowadzi nas do następującej definicji.

DEFINICJA 4.3 *Gramatykę bezkontekstową G nazywamy niejednoznaczną, jeśli istnieje słowo $\alpha \in L(G)$ posiadające więcej niż jedno wyprowadzenie lewostronne (równoważnie prawostronne) w G. W przeciwnym wypadku mówimy, że gramatyka G jest jednoznaczna.*

Nie wszystkie gramatyki posiadają równoważną postać jednoznaczną. W takim wypadku przenosimy pojęcie niejednoznaczności na generowany język.

DEFINICJA 4.4 *Mówimy, że bezkontekstowy język L jest wewnętrznie niejednoznaczny, jeśli wszystkie generujące go gramatyki są niejednoznaczne.*

PRZYKŁAD 4.5 Niech $L_1 = \{a^n b^n c^m : m, n \geqslant 0\}$ oraz $L_2 = \{a^m b^n c^n : m, n \geqslant 0\}$. Każdy z tych języków jest bezkontekstowy. Łatwo także skonstruować bezkontekstową gramatykę G dla języka $L = L_1 \cup L_2$: zakładając, że symbolami startowymi w G_1 i G_2 są S_1 i odpowiednio S_2, wystarczy jeśli połączymy ich zbiory produkcji (przyjmujemy przy tym standardowo, że $V_1 \cap V_2 = \emptyset$) i uzupełnimy je produkcją $S \to S_1 \mid S_2$. Wówczas słowo $a^n b^n c^n$ należące do obydwu tych języków posiada dwa odmienne wyprowadzenia — jedno w G_1, drugie w G_2. Choć nie jest to oczywiste na pierwszy rzut oka, można wykazać, że nie istnieje równoważna jednoznaczna gramatyka dla tego języka. Dowód tego faktu znaleźć można w literaturze.

3 Postać normalna gramatyki bezkontekstowej

Większość współczesnych języków programowania posiada strukturę języka bezkontekstowego. Proste elementy, jak nazwy, stałe itp. można opisać przez wyrażenia regularne, natomiast struktury bardziej złożone, takie jak wyrażenia, deklaracje struktur danych, instrukcje złożone, wymagają opisu przez gramatyki bezkontekstowe. Zauważmy, że przywołane przez nas języki poprawnie zagnieżdżonych nawiasów i wyrażeń arytmetycznych (por. Przykłady 2.4, 4.2, 4.3 i poprzedni podrozdział) są elementarnymi przykładami języków nieregularnych, jak łatwo przekonać się stosując Twierdzenie 3.11 lub Lemat 3.1. To właśnie między innymi te powszechnie obecne struktury sprawiają, że wysokopoziomowe języki programowania wymagają bezkontekstowego formalizmu.[2]

Obecność pustych i jednostkowych produkcji, a także wspomniana w poprzednim podrozdziale niejednoznaczność gramatyk są elementami obniżającymi sprawność algorytmów analizy składniowej, czyli tzw. parsingu. Dlatego w praktyce staramy się

[2]W gruncie rzeczy w wielu językach można dopatrzyć się nawet pewnych struktur zależnych od kontekstu, jak np. mechanizmy domyślnego przyporządkowania typów obiektom lub zgodności typów danych. Są to więc na ogół pewne własności semantyczne. Jednak użycie gramatyk kontekstowych Chomsky'ego do ich opisu nie jest praktycznym rozwiązaniem, ponieważ nie są dla nich znane ogólne, efektywne algorytmy. Na poziomie kompilatora stosuje się powszechnie inne, mniej uniwersalne lecz wydajne techniki do ich obsługi, np. stosy i tablice symboli oraz ich właściwości.

projektować gramatyki tak, aby wyeliminować z nich niepożądane własności i nadać ich produkcjom pewną, szczególnie prostą postać, tzw. *postać normalną*. Jest ona punktem wyjścia do budowy efektywnego algorytmu parsingu. Opiszemy obecnie systematyczne metody upraszczania postaci gramatyki bezkontekstowej: transformujemy G do nowej postaci G' wolnej od wspomnianych wad, gwarantując jednak, że $L(G) = L(G')$.

Jak widzieliśmy w poprzednich rozdziałach, gramatyki mogą powstawać w sposób algorytmiczny, a więc za pośrednictwem ogólnych konstrukcji, odpowiadających np. operacjom mnogościowym na językach lub generujących opis gramatyczny na podstawie opisu akceptującego automatu. W takich sytuacjach w gramatyce mogą pojawić się produkcje zbędne lub redundantne, nie wnoszące nic do generowanego języka. Nazywamy je produkcjami bezużytecznymi.

DEFINICJA 4.5 *Symbol nieterminalny X w bezkontekstowej gramatyce G nazywamy użytecznym, jeśli istnieje przynajmniej jedno słowo $\alpha \in L(G)$ takie, że*

$$S \;\Rightarrow_G^* \; \beta X \gamma \;\Rightarrow_G^* \; \alpha\,, \qquad \beta, \gamma \in (A \cup V)^*\,.$$

Innymi słowy, X jest użyteczny jeśli pojawia się w co najmniej jednym wyprowadzeniu terminalnego słowa w $L(G)$. Wszystkie inne nieterminale w G i produkcje, w których występują nazywamy bezużytecznymi.

Zgodnie z powyższą definicją nieterminal może być bezużyteczny z dwóch powodów. Po pierwsze, gdy nie jest on osiągalny z S, a więc gdy nie istnieje wyprowadzenie $S \Rightarrow^* \beta X \gamma$. Przykładem może być symbol X w następującej gramatyce:

$$\begin{aligned} S &\to aSb \,|\, Y \,|\, \lambda \\ Y &\to bbY \\ X &\to bYa \,|\, aS \end{aligned}$$

Drugim powodem bezużyteczności nieterminala może być to, że nie można wyprowadzić z niego żadnego napisu terminalnego. Przykładem takiej zmiennej jest tu Y. Po usunięciu z gramatyki wszystkich produkcji, w których pojawiają się X lub Y, pozostaje $S \to aSb \,|\, \lambda$, a generowany język nie ulega zmianie.

Eliminowanie bezużytecznych symboli i produkcji z G przebiega w dwóch etapach: najpierw usuwamy symbole niewyprowadzające napisów końcowych, a następnie te, których nie można osiągnąć z S. Proponujemy czytelnikowi przekonanie się, że odwrócenie tej kolejności może skutkować pozostawieniem w G niektórych bezużytecznych symboli. Pierwszą część algorytmu opiszemy następująco:

(i) $U := \emptyset$;

(ii) Dla każdego nieterminala X takiego, że w G znajduje się produkcja postaci

$$X \to x_1 x_2 \ldots x_m\,, \qquad \text{gdzie wszystkie} \;\; x_i \in U \cup A\,,$$

dodaj X do zbioru U. Powtarzaj (ii) do chwili, aż żaden nowy symbol X nie zostanie dodany do U.

(iii) Usuń z G wszystkie symbole spoza zbioru U i produkcje, w których te symbole występują. U staje się więc nowym alfabetem nieterminalnym.

Usunięcia symboli nieosiągalnych z S najłatwiej dokonać na podstawie analizy grafu zależności nieterminali w G. Graf ten tworzymy w taki sposób, że jako jego wierzchołki bierzemy nieterminale w U, a następnie łączymy X i Y strzałką, jeśli istnieje produkcja postaci $X \to \beta Y \gamma$. Wystarczy teraz oznaczyć wszystkie te wierzchołki grafu, do których prowadzą ścieżki z wierzchołka S. Pozostałe odpowiadają symbolom bezużytecznym. Jak poprzednio, usuwamy je z G wraz z produkcjami, w których występują.

Zajmiemy się obecnie opisem metody eliminowania λ-produkcji z G. Jedyny dopuszczalny wyjątek stanowić może produkcja $S \to \lambda$, która jest niezbędna w sytuacji, gdy λ jest poprawnym słowem języka. Umawiamy się przy tym, że produkcja ta nie będzie używana w wyprowadzeniach innych słów języka $L(G)$, co oznacza, że symbol S nie pojawia się z prawej strony jakiejkolwiek produkcji.

Zauważmy, że metoda nie może polegać po prostu na usunięciu z gramatyki pustych produkcji, bowiem tą drogą można nieuważnie zmienić język $L(G)$. Oto przykład:

$$\begin{aligned} S &\to aSbC \mid ab \\ C &\to c \mid \lambda \end{aligned}$$

Usunięcie produkcji $C \to \lambda$ spowoduje, że z $L(G)$ znikną słowa $aabb, aaabbb$ itd. Widać więc, że potrzebna jest bardziej przemyślana metoda. Tworzymy najpierw zbiór W symboli nieterminalnych X takich, że $X \Rightarrow^* \lambda$:

(i) Dodaj do W wszystkie symbole X, które występują w produkcjach $X \to \lambda$.

(ii) Dodaj do W wszystkie nieterminale Y takie, że G zawiera produkcję

$$Y \to Z_1 Z_2 \ldots Z_k\,, \qquad \text{gdzie wszystkie} \quad Z_i \in W\,.$$

Powtarzaj (ii) do czasu, aż żaden nowy symbol Y nie zostanie dodany do W.

Mając skonstruowany zbiór W, możemy przystąpić do modyfikacji zbioru produkcji. Niech $X \to y_1 y_2 \ldots y_m$, gdzie $y_i \in (A \cup V)$, będzie kolejną produkcją w G. Tworzymy nowy zbiór produkcji dodając do niego kolejno:

— samą produkcję $X \to y_1 y_2 \ldots y_m$

— wszystkie produkcje powstałe z niej przez opuszczenie k symboli y_i jeśli należą one do zbioru W, kolejno dla $k = 1, 2, \ldots m-1$.

Zilustrujmy ostatnią regułę przykładem. Niech $P, Q, R \in W$ i rozważmy produkcję $X \to PQQR$. W nowym zbiorze znajdą się wszystkie produkcje

$$X \to PQQR|QQR|PQR|PQQ|QR|PR|PQ|QQ|P|Q|R.$$

Opuszczenie wszystkich czterech symboli P, Q, R generuje produkcję $X \to \lambda$, lecz jej oczywiście nie dołączamy do nowego zbioru.

W końcu jeśli $S \in W$, trzeba dodać w charakterze wyjątku produkcję $S \to \lambda$ by zachować słowo λ w języku. Tę czynność jednak przesuniemy na koniec całej procedury przekształcania gramatyki. To zamyka etap eliminacji λ-produkcji.

Pozostaje nam opisać metodę pozbywania się z G produkcji jednostkowych $X \to Y$. Produkcje tego typu mogą powstawać podczas wcześniejszej eliminacji λ-produkcji. Możliwe jest w szczególności, że w G pojawią się zapętlone ciągi produkcji jednostkowych, takie jak $X \to Y$ i $Y \to X$. Algorytm eliminacji musi radzić sobie z takimi przypadkami.

Zaczynamy od zbudowania grafu H zależności między nieterminalami występującymi w jednostkowych produkcjach $X \to Y$. Następnie tworzymy jego tranzytywne domknięcie H^*. W ten sposób krawędź z X do Z sygnalizuje, że w naszej gramatyce $X \Rightarrow^* Z$. Utworzymy teraz nowy zbiór produkcji Ω nie zawierający już produkcji jednostkowych.

(i) Niech Ω będzie zbiorem wszystkich niejednostkowych produkcji w G.

(ii) Dla każdej pary symboli X i Z połączonych krawędzią w H^* dodajemy do Ω produkcje postaci $X \to \beta$ dla wszystkich β, dla których $Z \to \beta$ znajduje się w Ω.

Zauważmy, że ograniczenie się w (ii) do produkcji $Z \to \beta$ ze zbioru Ω gwarantuje, że w tym procesie nie powstają produkcje jednostkowe. Łatwo sprawdzić, że transformacja ta nie zmienia języka. Jeśli bowiem w wyprowadzeniu $S \Rightarrow^* \alpha$ uczestniczą ciągi jednostkowych produkcji odpowiadające relacji $X \Rightarrow^* Z$, będą one zredukowane przez odpowiednią nową produkcję $X \to \beta$ wygenerowaną w (ii).

By ostatecznie sformułować kompletną procedurę upraszczania gramatyki, wystarczy jeśli zaobserwujemy, że eliminacja pustych produkcji może generować produkcje jednostkowe, ale nie odwrotnie. Poza tym usuwanie zbędnych symboli nie wprowadza ani pustych ani też jednostkowych produkcji. Reasumując, metodę naszą realizujemy następująco:

1) Usuwamy z gramatyki produkcje puste.

2) Eliminujemy produkcje jednostkowe.

3) Usuwamy symbole i produkcje bezużyteczne.

4) Jeśli w punkcie 1) okazałoby się, że $S \in W$, a więc, że $\lambda \in L(G)$, uzupełniamy przekształconą gramatykę o dodatkową produkcję $S \to \lambda$.

Podsumujmy nasze rezultaty w formie twierdzenia.

TWIERDZENIE 4.1 *Każdą gramatykę bezkontekstową G, dla której $\lambda \notin L(G)$ można przekształcić do równoważnej*[3] *postaci G' nie zawierającej produkcji pustych, jednostkowych, ani też bezużytecznych. Oznacza to, że jeśli $\alpha \Rightarrow_{G'} \beta$, wówczas $|\alpha| \leqslant |\beta|$, przy czym równość możliwa jest tylko wtedy, gdy w wyprowadzeniu użyto produkcji typu $X \to a$ z $a \in A$.*

Jeśli $\lambda \in L(G)$, wówczas G' zawiera wyjątkową produkcję $S \to \lambda$, a poza tym posiada wszystkie wymienione wyżej własności.

[3]Przypomnijmy, że gramatyki G i G' są równoważne jeśli $L(G) = L(G')$.

Postać normalna.

Opiszemy dwie specjalne postaci gramatyk bezkontekstowych, które pozwalają na znaczne uproszczenie opartych na nich algorytmów analizy składniowej.

DEFINICJA 4.6 *Mówimy, że bezkontekstowa gramatyka G jest dana w normalnej postaci Chomsky'ego, jeśli wszystkie jej produkcje mają formę*

$$X \to YZ \quad lub \quad X \to a\,,$$

gdzie $X, Y, Z \in V$ oraz $a \in A$.

Definicja jest na tyle czytelna, że nie wymaga komentarzy, odnotujmy jedynie, że postać Chomsky'ego wyklucza występowanie w gramatyce produkcji $X \to \lambda$. Zatem jeśli rozważamy języki ze słowem λ, musi być ono traktowane oddzielnie. Oto ważny dla nas rezultat.

TWIERDZENIE 4.2 (Postać normalna Chomsky'ego) *Każdą gramatykę bezkontekstową $G = (A, V, S, \Pi)$ taką, że $\lambda \notin L(G)$, można przekształcić do równoważnej postaci normalnej Chomsky'ego $G' = (A, V', S, \Pi')$.*

Przed przeprowadzeniem dowodu tego twierdzenia, zauważmy, że ograniczenie $\lambda \notin L(G)$ jest łatwe do obejścia. Jeśli bowiem $\lambda \in L(G)$, przekształćmy wstępnie G do postaci λ-wolnej G_0, usuwając z niej wszystkie puste produkcje, tak jak opisaliśmy to wcześniej. Skoro teraz $\lambda \notin L(G_0)$, dla G_0 znajdziemy postać Chomsky'ego, którą ostatecznie uzupełnimy o specjalną produkcję $S \to \lambda$.

Dowód. Rozpoczynamy od uproszczenia gramatyki G przez wyeliminowanie z niej λ-produkcji, a także produkcji jednostkowych i bezużytecznych. Można więc przyjąć, że wszystkie jej produkcje są postaci

$$X \to y_1 y_2 \dots y_m\,, \qquad y_i \in A \cup V\,, \tag{4.7}$$

przy czym jeśli $m = 1$, a więc $X \to y_1$, wówczas na pewno $y_1 \in A$. W tym wypadku dodajemy taką produkcję do nowego zbioru Π'. Ponadto, gdy $m \geqslant 2$, dla wszystkich $a \in A$ występujących wśród y_i do Π' dodajemy reguły $X_a \to a$, gdzie X_a jest nowym, unikalnym dla każdego a nieterminalem. Każdą produkcję (4.7) z $m \geqslant 2$ przepiszemy teraz w postaci

$$X \to Q_1 Q_2 \dots Q_m\,, \tag{4.8}$$

gdzie $Q_i = y_i$ jeśli $y_i \in V$ lub $Q_i = X_a$ jeśli $y_i = a \in A$. W ten sposób prawe strony (4.7) z $m \geqslant 2$ zbudowane są już z samych nieterminali. Oczywiste jest przy tym, że operacje te nie zmieniają języka generowanego przez G. Następnie, tworząc nowe symbole nieterminalne R_i, każdą z reguł (4.8) przekształcamy w równoważny ciąg produkcji w postaci Chomsky'ego

$$\begin{aligned} X &\to Q_1 R_1 \\ R_1 &\to Q_2 R_2 \\ &\vdots \\ R_{m-2} &\to Q_{m-1} Q_m \end{aligned} \tag{4.9}$$

i dodajemy je do zbioru Π'. Biorąc jako V' stary zbiór V powiększony o wszystkie dodane symbole X_a oraz R_i, otrzymujemy gramatykę G' w normalnej postaci Chomsky'ego oraz $L(G) = L(G')$. □

PRZYKŁAD 4.6 Przekształcimy gramatykę

$$\begin{aligned} S &\to abXY \mid YXY \\ X &\to bXY \mid \lambda \\ Y &\to YXa \mid X \mid \lambda \end{aligned}$$

do postaci normalnej Chomsky'ego.

Standardowo rozpoczynamy od eliminacji λ-produkcji. W zbiorze W umieszczamy nieterminale X i Y, ponieważ G zawiera produkcje $X \to \lambda$ i $Y \to \lambda$. W kolejnej iteracji do W trafia też S z uwagi na obecność produkcji $S \to YXY$. Mamy więc $W = \{X, Y, S\}$ i możemy przystąpić do tworzenia nowego zbioru reguł. Z pierwszej produkcji przez wszystkie kombinacje podstawienia λ pod X i Y otrzymujemy

$$S \to ab \mid abX \mid abY \mid abXY \mid YXY \mid XY \mid YX \mid YY \mid X \mid Y\,.$$

Podobnie z pozostałych dwóch produkcji mamy

$$\begin{aligned} X &\to b \mid bX \mid bY \mid bXY \\ Y &\to a \mid Xa \mid Ya \mid YXa \mid X \end{aligned}$$

Ponieważ $S \in W$, należy pamiętać o dodaniu do końcowej postaci Chomsky'ego naszej gramatyki dodatkowej produkcji $S \to \lambda$.

Następny krok polega na usunięciu produkcji jednostkowych z G. Zależności jednostkowe to $S \to X$ oraz $S \to Y \to X$, a ich tranzytywne domknięcie nie dodaje żadnych nowych relacji. Nowy zbiór Π obejmuje więc wszystkie dotychczasowe niejednostkowe produkcje

$$\begin{aligned} S &\to ab \mid abX \mid abY \mid abXY \mid YXY \mid XY \mid YX \mid YY \\ X &\to b \mid bX \mid bY \mid bXY \\ Y &\to a \mid Xa \mid Ya \mid YXa \end{aligned}$$

oraz produkcje zastępujące dotychczasowe jednostkowe $S \to X \mid Y$ i $Y \to X$:

$$\begin{aligned} S &\to b \mid bX \mid bY \mid bXY \mid a \mid Xa \mid Ya \mid YXa \\ Y &\to b \mid bX \mid bY \mid bXY\,. \end{aligned}$$

W końcu łatwo stwierdzamy, że nasza gramatyka nie zawiera bezużytecznych nieterminali, ponieważ X i Y są osiągalne z S oraz każdy z nich wyprowadza napisy końcowe. Możemy więc przystąpić do przekształcenia produkcji do postaci normalnej. Na początek tworzymy dwie produkcje dla terminali

$$X_a \to a\,, \qquad X_b \to b\,, \tag{4.10}$$

a oprócz tego nasz zbiór Π' zawiera końcowe produkcje z gramatyki

$$S \to a \mid b\,, \quad X \to b\,, \quad Y \to a \mid b\,. \tag{4.11}$$

Z kolei przepisujemy przy pomocy nieterminali i porządkujemy cały zbiór produkcji o dwu- lub więcej znakowych prawych stronach. Wyodrębnijmy na początek te, które już są w postaci Chomsky'ego

$$\begin{aligned} S &\to X_aX_b \,|\, XY \,|\, YX \,|\, YY \,|\, X_bX \,|\, X_bY \,|\, XX_a \,|\, YX_a \\ X &\to X_bX \,|\, X_bY \\ Y &\to XX_a \,|\, YX_a \,|\, X_bX \,|\, X_bY\,. \end{aligned} \tag{4.12}$$

Widać teraz rolę jaką odgrywają pozornie niepotrzebne symbole X_a i X_b, dublujące już obecne w gramatyce reguły (4.11). Nie można użyć np. symbolu S w roli a w podstawieniach po prawej stronie produkcji, bowiem prowadziłoby to do utworzenia iteracyjnych reguł nieobecnych w pierwotnej gramatyce, np. z $S \to Xa$ otrzymalibyśmy $S \to XS$, a to najprawdopodobniej zmieniłoby generowany język.

Następnie zapisujemy produkcje o 3- i 4-znakowych prawych stronach

$$\begin{aligned} S &\to X_aX_bX \,|\, X_aX_bY \,|\, X_aX_bXY \,|\, YXY \,|\, X_bXY \,|\, YXX_a \\ X &\to X_bXY \\ Y &\to YXX_a \,|\, X_bXY \end{aligned}$$

i konwertujemy je do postaci Chomsky'ego przez wprowadzenie nowych symboli pomocniczych. Niech

$$R_1 \to X_bX\,, \quad R_2 \to X_bY\,, \quad R_3 \to XX_a\,, \quad R_4 \to XY \tag{4.13}$$

oraz

$$R_5 \to X_bR_4\,. \tag{4.14}$$

Stąd mamy

$$\begin{aligned} S &\to X_aR_1 \,|\, X_aR_2 \,|\, X_aR_5 \,|\, YR_4 \,|\, X_bR_4 \,|\, YR_3 \\ X &\to X_bR_4 \\ Y &\to YR_3 \,|\, X_bR_4\,. \end{aligned} \tag{4.15}$$

Zbiór nieterminali liczy więc 10 elementów $V' = \{S, X, Y, X_a, X_b, R_1, \ldots R_5\}$, natomiast zbiór Π' składa się z produkcji (4.10–4.15). Powinniśmy uzupełnić go jeszcze o wyjątkową produkcję $S \to \lambda$ by zachować zgodność z pierwotnym językiem. W następnym rozdziale przekonamy się, że mimo widocznej nieprzejrzystości, gramatyka w tej postaci umożliwia efektywne weryfikowanie czy dowolnie zadane słowa znajdują się w języku $L(G')$.

Opiszemy jeszcze jedną, pożyteczną postać kanoniczną gramatyki bezkontekstowej. Ograniczymy się tym razem do przytoczenia podstawowych faktów bez dowodów. Zainteresowany czytelnik odnajdzie bardziej kompletną dyskusję w literaturze.

DEFINICJA 4.7 *Bezkontekstową gramatykę $G = (A, V, S, \Pi)$ nazywamy gramatyką w postaci normalnej Greibach,*[4] *gdy zawiera wyłącznie produkcje w formie*

$$X \to aY_1Y_2 \ldots Y_m\,, \tag{4.16}$$

gdzie $a \in A$, $m \geqslant 0$ oraz $Y_i \in V$.

[4]Sheila Greibach jest emerytowanym profesorem informatyki Uniwersytetu Kalifornijskiego w Los Angeles. Konstrukcja postaci normalnej gramatyk pochodzi 1965 r.

Podstawowy wynik jest bliźniaczo podobny do Twierdzenia 4.2.

TWIERDZENIE 4.3 (Postać normalna Greibach) *Każdą gramatykę bezkontekstową* $G = (A, V, S, \Pi)$ *taką, że* $\lambda \notin L(G)$*, można przekształcić do równoważnej postaci normalnej Greibach* $G' = (A, V', S, \Pi')$.

Jeśli gramatyka ma np. postać

$$S \;\to\; aaSb \,|\, aab\,,$$

wówczas, wzorem konstrukcji dla postaci Chomsky'ego, wprowadzając pomocnicze produkcje $X_a \to a$, $X_b \to b$, możemy łatwo uzyskać postać Greibach:

$$S \;\to\; aX_aSX_b \,|\, aX_aX_b\,.$$

Nie jest to jednak metoda uniwersalna, bowiem ta sama technika nie sprawdzi się, jeśli np. w gramatyka zawiera produkcję $S \to Sab$. Postać Greibach *wymaga*, aby pierwszym znakiem prawej strony produkcji był zawsze znak terminalny. Ogólna konstrukcja jest w tym wypadku znacznie bardziej skomplikowana.

4 Algorytm "CYK"

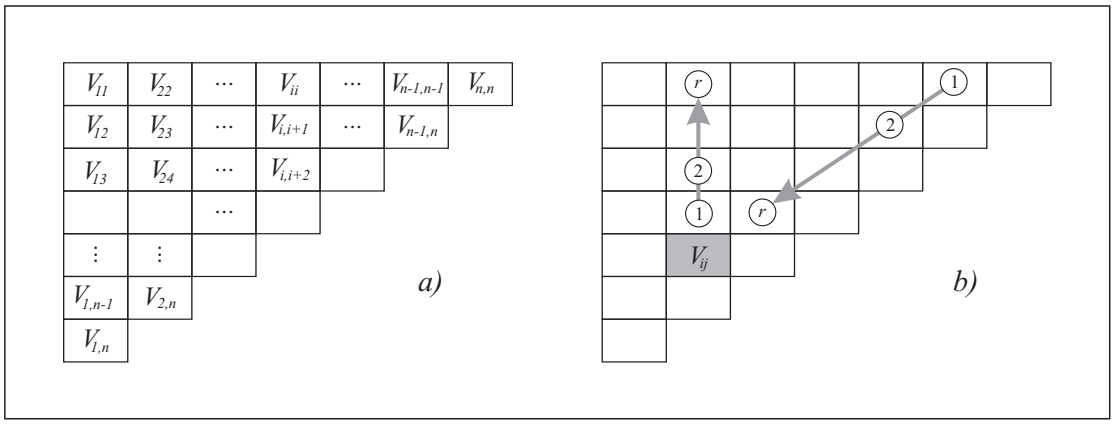
Omówimy obecnie efektywną metodę analizy składniowej dla słów języka opisanego przez gramatykę bezkontekstową w postaci Chomsky'ego. Algorytm bierze swój kryptonim od nazwisk trzech informatyków pracujących nad nim niezależnie w końcu lat 60-tych — Cooke'a, Youngera i Kasamiego. Jest to pouczający przykład zastosowania techniki programowania dynamicznego.

Niech $G = (A, V, S, \Pi)$ będzie gramatyką bezkontekstową w postaci normalnej Chomsky'ego. Zadanie polega na tym, aby dla danego słowa $\alpha = a_1a_2\ldots a_n$ rozstrzygnąć czy $\alpha \in L(G)$, czy też nie. Algorytm bada systematycznie podsłowa $\alpha_{ij} = a_i \ldots a_j$ dla $1 \leqslant i \leqslant j \leqslant n$, tworząc przy tym zbiory

$$V_{ij} \;=\; \{X \in V :\; X \Rightarrow^* \alpha_{ij}\}\,.$$

Zapamiętajmy: V_{ij} to zbiór wszystkich tych nieterminali gramatyki, z których wyprowadzić można podnapis α_{ij}. Wówczas $\alpha \in L(G)$ wtedy i tylko wtedy, gdy $S \in V_{1n}$. Okazuje się, że konstrukcja zbiorów V_{ij} może być przeprowadzona w czasie proporcjonalnym do n^3. Bierze się to stąd, że w postaci Chomsky'ego wszystkie produkcje oprócz końcowych mają dwuliterowe prawe strony. Jeśli np. wyprowadzenie $X \Rightarrow^* \alpha_{ij}$ zaczyna się od zastosowania produkcji $X \to YZ$, a więc $X \Rightarrow YZ \Rightarrow^* \alpha_{ij}$, mamy stąd $Y \Rightarrow^* \alpha_{ik}$ oraz $Z \Rightarrow^* \alpha_{k+1,j}$ dla pewnego $i \leqslant k < j$. Innymi słowy konstrukcję zbiorów V_{ij} można prowadzić dynamicznie, uzyskując V_{ij} na podstawie analizy zawartości wcześniej utworzonych zbiorów V_{ik} i $V_{k+1,j}$, uwzględniając wszystkie punkty podziału k takie, że $i \leqslant k < j$.

Zaczynamy od zbudowania n zbiorów V_{ii}. Ponieważ produkcje gramatyki G są nieskracające, łatwo zauważyć, że $X \in V_{ii}$ jedynie wtedy, gdy G zawiera produkcję $X \to a_i$. V_{ii} wyznaczamy więc przez przegląd zbioru produkcji Π. By utworzyć następnie zbiory V_{ij} dla $i < j$, postępujemy systematycznie tak jak opisaliśmy to



Rys. 4.2: a) Tabela tworzona przez algorytm CYK. b) Kolejne pary zbiorów w tabeli wyznaczające V_{ij}.

przed chwilą. Do V_{ij} trafiają te nieterminale X, które są lewymi stronami produkcji $X \to YZ$ o prawej stronie zbudowanej z dwóch symboli odpowiednio ze zbiorów V_{ik} i $V_{k+1,j}$ dla pewnego $i \leqslant k < j$. Prowadzi to nas do konstrukcji będącej podstawowym krokiem iteracyjnym algorytmu:

$$V_{ij} \;=\; \bigcup_{i \leqslant k < j} \Big\{ X :\; X \to YZ \in \Pi \quad \text{i} \quad Y \in V_{i,k},\; Z \in V_{k+i,j} \Big\}. \tag{4.17}$$

Kolejność wyznaczania zbiorów V_{ij} jest więc następująca: najpierw $V_{11}, \ldots, V_{nn}$, następnie $V_{12}, V_{23}, \ldots, V_{n-1,n}$, z kolei V_{13} $\ldots, V_{n-2,n}$ itd. Gdy otrzymamy V_{1n}, wystarczy sprawdzić czy S jest jego elementem.

By oszacować złożoność tej metody zauważmy, że w pierwszej iteracji konstruujemy n zbiorów V_{ii}, w drugiej $n-1$ zbiorów $V_{i,i+1}$ itd., a więc w sumie powstają $n(n+1)/2$ zbiory. Dla każdego z nich formuła (4.17) wymaga wyznaczenia co najwyżej n składników sumy mnogościowej. Stąd złożoność $O(n^3)$.

Przykład 4.7 Przeanalizujemy działanie algorytmu CYK na przykładzie gramatyki języka poprawnie zagnieżdżonych nawiasów $S \to ( ) \,|\, SS \,|\, (S)$ Przekształcamy gramatykę do postaci Chomsky'ego, wprowadzając pomocnicze produkcje dla terminali $L \to (\;$ oraz $\; R \to)$:

$$\begin{aligned} S &\to LR \,|\, SS \,|\, LQ \\ Q &\to SR\,. \end{aligned}$$

Przekonajmy się, że 8-znakowy napis $(( )( ))( )$ jest poprawny w sensie tej gramatyki.

Do wyznaczenia zbiorów V_{ij} wygodnie jest posłużyć się trójkątną tabelą, wypełnianą dynamicznie przez przez algorytm, Rys. 4.2. Kolejne wiersze odpowiadają coraz to dłuższym fragmentom słowa α, $\alpha_{ij} = a_i a_{i+1} \ldots a_j$. Tak więc zbiory $V_{11}, \ldots, V_{nn}$ wypełniają 1-szy wiersz, zbiory $V_{12}, \ldots, V_{n-1,n}$ — drugi, itd. Jedyny element ostatniego wiersza to zbiór V_{1n}.

Mechanizm tworzenia kolejnych zbiorów pokazuje Rys. 4.2 b). Np. by wygenerować V_{ij} posługujemy się kolejnymi zbiorami $V_{i,k}$ w kolumnie nad V_{ij} dobierając do

nich do pary kolejne zbiory $V_{k+1,j}$ z przekątnej, na której leży V_{ij}. Ponieważ tworzenie każdego ze zbiorów V_{ij} wymaga przeglądu listy produkcji gramatyki w poszukiwaniu tych, których prawe strony YZ są kombinacjami symboli Y z $V_{i,k}$ oraz Z z $V_{k+1,j}$, celowe jest zakodowanie zbioru produkcji w postaci specjalnej tabeli podglądu, co maksymalnie usprawni proces wyszukiwania:

$$\Gamma[Y,Z] = \{X \in V : \ X \to YZ \in \Pi\}\,.$$

W naszym przykładzie tabela Γ ma bardzo prostą postać:

	S	L	R	Q
S	S	$\emptyset$	Q	$\emptyset$
L	$\emptyset$	$\emptyset$	S	S
R	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
Q	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

przy czym dla czytelności piszemy w tabeli np. S zamiast $\{S\}$.

Rozpoczynamy od wypisania badanego napisu bezpośrednio nad pierwszym wierszem tabeli, który następnie wypełniamy odpowiednimi nieterminalami przeglądając terminalne produkcje gramatyki $L \to ($ oraz $R \to)$:

(	(	)	(	)	)	(	)
L	L	R	L	R	R	L	R

Dla czytelności zaniedbujemy nawiasy zbiorowe { } w klatkach tabeli.[5] Z kolei V_{12}, zgodnie z (4.17), zawiera wszystkie nieterminale będące lewymi stronami produkcji o prawych stronach zbudowanych z pary symboli z V_{11} i V_{22}, znajdujących się w tabeli bezpośrednio nad V_{12} oraz wzdłuż przekątnej, a więc będących postaci LL. Ponieważ $\Gamma[L,L] = \emptyset$, także $V_{12} = \emptyset$. Postępując podobnie znajdujemy pozostałe elementy drugiego wiersza tabeli:

(	(	)	(	)	)	(	)
L	L	R	L	R	R	L	R
$\emptyset$	S	$\emptyset$	S	$\emptyset$	$\emptyset$	S	

Wyznaczenie V_{13} wymaga uwzględnienia dwóch par zbiorów, zgodnie z (4.17) oraz Rys. 4.2 b). Ponieważ $V_{12} = \emptyset$, nie istnieje forma YZ, w której $Y \in V_{12}$, a zatem przyczynek od pary V_{12} i V_{33} do zbioru V_{13} jest pusty. Druga para V_{11} i V_{23} generuje formę LS, która nie jest prawą stroną żadnej z produkcji, $\Gamma[L,S] = \emptyset$, a więc także ten przyczynek jest pusty. Stąd $V_{13} = \emptyset$. Podobnie uzupełniamy pozostałe elementy

[5] Pamiętajmy jednak, że elementy V_{ii} są w ogólności *zbiorami* i mogą zawierać wiele elementów jeśli proces redukcji gramatyki do postaci Chomsky'ego wygeneruje kilka produkcji terminalnych $X_i \to a$ dla tej samej litery a. Chwila zastanowienia powinna doprowadzić czytelnika do wniosku, że próba uproszczenia takiej gramatyki przez zastąpienie wielu symboli X_i jednym może prowadzić do niepożądanych efektów.

trzeciego wiersza tabeli. Okazuje się przy tym, że jedynie V_{46} jest niepusty i zawiera nieterminal Q, ponieważ para V_{45} i V_{66} generuje formę SR, dla której $\Gamma[S,R] = Q$. Kontynuujemy tę procedurę zgodnie z Rys. 4.2 b) dla kolejnych wierszy.

Ostateczna postać tabeli jest następująca:

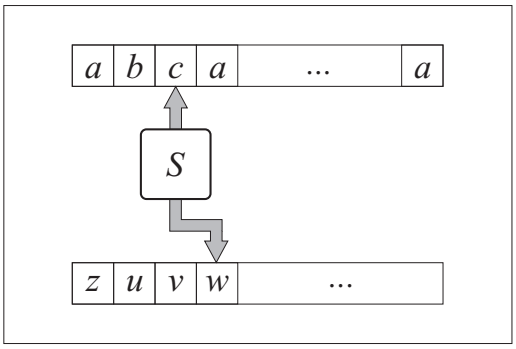
(	(	)	(	)	)	(	)
L	L	R	L	R	R	L	R
$\emptyset$	S	$\emptyset$	S	$\emptyset$	$\emptyset$	S	
$\emptyset$	$\emptyset$	$\emptyset$	Q	$\emptyset$	$\emptyset$		
$\emptyset$	S	$\emptyset$	$\emptyset$	$\emptyset$			
$\emptyset$	Q	$\emptyset$	$\emptyset$				
S	$\emptyset$	$\emptyset$					
$\emptyset$	$\emptyset$						
S							

Ostatni krok oblicza $V_{18} = \{S\}$: ponieważ $V_{16} = V_{78} = \{S\}$, a więc S trafia do V_{18} za sprawą produkcji $S \to SS$. Widzimy stąd, że napis (( )( ))( ) jest poprawny. Zachęcamy czytelnika do wykonania podobnej analizy dla błędnego tym razem napisu (( )( ).

Warto wspomnieć, że na bazie algorytmu CYK powstały ulepszone metody, których złożoność poprawia się dla pewnych typów gramatyk i np. spada do $O(n^2)$ dla gramatyk jednoznacznych. Jedną z takich metod jest tzw. algorytm Earley'a (1970).

IV.2 Automaty ze stosem

Automaty ze stosem pełnią dla języków bezkontekstowych taką samą rolę, jaką spełniają skończone automaty w odniesieniu do języków regularnych. Analogia w tym przypadku nie jest jednak idealna, bowiem istnieje istotna różnica między deterministycznymi a niedeterministycznymi automatami ze stosem. Jak sugeruje nazwa, automaty te są wyposażone w roboczą pamięć działającą na zasadzie stosu:



1 Deterministyczne i niedeterministyczne automaty ze stosem

Przytoczymy na początku precyzyjną definicję automatu ze stosem.

DEFINICJA 4.8 *Niedeterministyczny automat ze stosem (NAS) opisany jest przez siedem obiektów*

$$M \;=\; \Big(A, V, z, \Sigma, s_0, S_F, \pi\Big)\,,$$

gdzie:

A *jest alfabetem wejściowym*
V *jest alfabetem stosu*
$z \in V$ *jest wyróżnionym symbolem początkowym stosu*
Σ *jest zbiorem stanów automatu,* $\sigma = \{s_0, s_1, \ldots, s_q\}$
$s_0 \in \Sigma$ *jest wyróżnionym stanem startowym*
$S_F \subseteq \Sigma$ *jest zbiorem stanów końcowych (akceptujących)*
π *jest relacją przejścia,* $\pi : \Sigma \times (A \cup \{\lambda\}) \times V \;\to\; \mathcal{F}(\Sigma \times V^*)\,.$

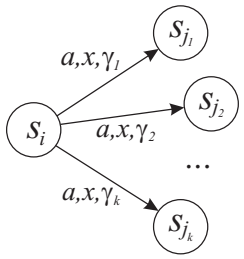
Działanie automatu ze stosem jest następujące. Skanuje on kolejny symbol na taśmie wejściowej (przesuwając przy tym głowicę czytającą o jedną pozycję w prawo), odczytuje wierzchni symbol ze stosu (usuwając go stamtąd) i zależnie od aktualnego stanu przechodzi niedeterministycznie do nowego stanu zapisując przy tym określony ciąg $\gamma \in V^*$ na wierzch stosu. Zwróćmy przy tym uwagę, że w określeniu relacji π zastrzegamy, że danej trójce "stan–litera–symbol na stosie" odpowiadać może jedynie *skończenie wiele* alternatywnych ruchów "stan–zapis γ na stosie" (ponieważ zbiór V^* jest nieskończony, zwykły zbiór potęgowy $\mathcal{P}(\Sigma \times V^*)$ zawiera jako elementy także nieskończone podzbiory, stąd też użycie $\mathcal{F}$ w tym miejscu). Możliwe jest także puste przejście $\pi(s, \lambda, x)$ polegające na tym, że w stanie s odczytywany jest wierzchni symbol stosu x, bez odczytywania taśmy wejściowej i bez ruchu jej głowicy. W każdej jednak sytuacji działanie automatu wymaga by w stosie znajdował się choć jeden symbol. Jeśli stos jest pusty, relacja przejścia nie jest określona: automat zatrzyma się nie mogąc wykonać kolejnego ruchu. Widać teraz jaką rolę pełni symbol początkowy stosu z: znajduje się on standardowo w stosie w chwili gdy automat rozpoczyna działanie.

Napis wejściowy zostaje rozpoznany, jeśli po przeczytaniu ostatniej jego litery automat znajdzie się w jednym ze swoich stanów finalnych. Końcowa zawartość stosu nie ma przy tym znaczenia. Możliwe jest także zatrzymanie automatu w stanie różnym od finalnego lub zatrzymanie przed osiągnięciem końca taśmy wejściowej. W przypadku automatu deterministycznego jest to równoznaczne z odrzuceniem słowa, natomiast automat niedeterministyczny nie dyskwalifikuje takiego napisu, może bowiem istnieć inna sekwencja ruchów kończąca się jego rozpoznaniem.

Dla automatów ze stosem, tak samo jak dla automatów skończonych, istnieje wygodna reprezentacja grafowa. Przykładowo, jeśli relacja przejścia ma postać

$$\pi(s_i, a, x) \;=\; \Big\{\,(s_{j_1}, \gamma_1),\; (s_{j_2}, \gamma_2),\; \ldots,\; (s_{j_k}, \gamma_k)\,\Big\}\,,$$

w grafie obecne będą krawędzie



Tym razem etykieta krawędzi, oprócz litery wejściowej a (lub symbolu λ dla pustych przejść), zawiera także symbol ze szczytu stosu x oraz napis γ, który zapisany będzie na stosie po usunięciu x.

Jeszcze innym, alternatywnym sposobem opisu automatu ze stosem jest relacja przejścia, którą automat indukuje w zbiorze $\left(A \cup \Sigma \cup V\right)^*$ zgodnie z regułą:

$$a_1 \ldots a_{k-1} s_i a_k \ldots a_n \, x_1 x_2 \ldots x_p \;\Rightarrow_M\; a_1 \ldots a_k s_j a_{k+1} \ldots a_n \, y_1 \ldots y_q x_2 \ldots x_p$$

wtedy i tylko wtedy, gdy

$$\pi(s_i, a_k, x_1) \;\ni\; (s_j, y_1 \ldots y_q)$$

oraz analogicznie dla pustych przejść $\pi(s_i, \lambda, x_1) \;\ni\; (s_j, y_1 \ldots y_q)$

$$a_1 \ldots a_{k-1} s_i a_k \ldots a_n \, x_1 x_2 \ldots x_p \;\Rightarrow_M\; a_1 \ldots a_{k-1} s_j a_k \ldots a_n \, y_1 \ldots y_q x_2 \ldots x_p \,.$$

Napisy postaci $\alpha_1 s \alpha_2 \gamma \in (A \cup \Sigma \cup V)^*$ nazywamy konfiguracjami i interpretujemy je jako wyczerpujące opisy stanów automatu M podczas skanowania słów wejściowych: po przeczytaniu części α_1 napisu wejściowego $\alpha = \alpha_1 \alpha_2$ automat znalazł się w stanie s, natomiast zawartość stosu w tym momencie to napis γ. Przyjmujemy przy tym konwencję, że szczytowym symbolem w stosie jest pierwszy znak γ. Zwróćmy uwagę, że nie ma potrzeby wprowadzania dodatkowego znaku separującego napis wejściowy od napisu na stosie pod warunkiem, że alfabety A i V są rozłączne — takie założenie można zawsze zrobić, ponieważ postać znaków w V nie wpływa na język akceptowany przez automat. Jak wcześniej, symbolem $\Rightarrow_M^*$ oznaczamy tranzytywne domknięcie relacji przejścia $\Rightarrow_M$.

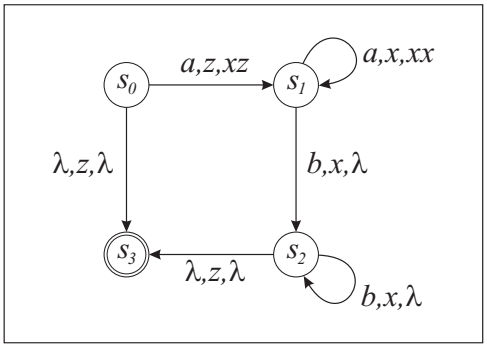
DEFINICJA 4.9 *Językiem $L(M)$ akceptowanym przez niedeterministyczny automat ze stosem M nazywamy zbiór słów $\alpha \in A^*$, dla których*

$$s_0 \alpha z \;\Rightarrow_M^*\; \alpha s_f \gamma$$

dla pewnego stanu $s_f \in S_F$ oraz dowolnego $\gamma \in V^*$. Klasę tych języków oznaczamy symbolem $\mathcal{L}_{\mathrm{Ndet}}$.

Innymi słowy, język $L(M)$ to zbiór wszystkich takich słów α, dla których *istnieje* co najmniej jedna sekwencja ruchów automatu osiągająca konfigurację akceptującą.

W praktyce gdy chcemy ustalić czy określone słowo α jest akceptowane przez zadany automat, staramy się przede wszystkim znaleźć sekwencję ruchów doprowadzającą głowicę czytającą na koniec α, gdyż zatrzymanie automatu w dowolnym innym miejscu nie produkuje konfiguracji akceptującej.



Rys. 4.3: Automat ze stosem akceptujący język $\{a^n b^n : \ n \geqslant 0\}$ z Przykładu 4.8.

PRZYKŁAD 4.8 Zbudujemy automat akceptujący język bezkontekstowy

$$L = \{a^n b^n : \ n \geqslant 0\}\,.$$

Alfabet stosu zawiera 2 symbole $V = \{x, z\}$, przy czym z jest standardowym symbolem początkowym stosu, natomiast x używane będzie do rejestrowania liczby przeczytanych liter a. Graf automatu przedstawia Rys. 4.3. Puste przejście $s_0 \to s_3$ pozwala zaakceptować pusty napis λ, który jest poprawnym słowem języka L. Jeśli taśma wejściowa nie jest pusta, przejście to nie skutkuje akceptacją zapisanego na niej słowa, bowiem głowica czytająca nie przesunie się na jego koniec. Pojawienie się na wejściu ciągu liter a powoduje, że automat przejdzie do stanu s_1 zapisując w stosie symbol x dla każdego kolejnego a. Ponieważ każdy odczyt stosu usuwa z niego wierzchni symbol, trzeba go powtórzyć przy zapisie jeśli intencją naszą jest dodanie nowego napisu na stos (np. przejście $s_0 \to s_1$ odczytuje symbol z i zapisuje w jego miejsce xz). Pojawienie się litery b powoduje usunięcie ze stosu pojedynczego x (zapisujemy pusty ciąg w jego miejsce). Ponowne osiągnięcie symbolu z na dnie stosu pozwala wykonać puste przejście $s_2 \to s_3$, które będzie ruchem akceptującym jeśli na taśmie wejściowej nie pozostaną już żadne litery.

Przyjrzyjmy się procesowi akceptacji słowa *aabb*, opisanemu ciągiem relacji przejścia. Mamy kolejno:

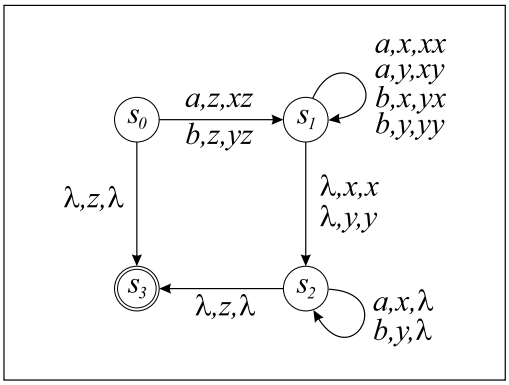
$$s_0aabbz \ \Rightarrow \ as_1abbxz \ \Rightarrow \ aas_1bbxxz \ \Rightarrow \ aabs_2bxz \ \Rightarrow \ aabbs_2z \ \Rightarrow \ aabbs_3\lambda\,.$$

Z kolei dla słowa *aabbb* sekwencja przejść może być następująca:

$$s_0aabbbz \ \Rightarrow \ as_1abbbxz \ \Rightarrow \ aas_1bbbxxz \ \Rightarrow \ aabs_2bbxz \ \Rightarrow \ aabbs_2bz \ \Rightarrow \ aabbs_3b\lambda\,,$$

jednak, jak widać, nie udaje się tą drogą osiągnąć końca napisu wejściowego. Dla słowa *aab*, po przeczytaniu b na stosie pozostaje jeden symbol x i automat nie potrafi opuścić stanu s_2.

PRZYKŁAD 4.9 Niech teraz $L = \left\{ \alpha\overleftarrow{\alpha} : \alpha \in \{a, b\}^* \right\}$. W pierwszej fazie automat będzie rejestrował na stosie symbole x i y odpowiadające wczytanym literom a i b napisu α, w drugiej zaś będzie porównywał kolejne litery $\overleftarrow{\alpha}$ z zawartością stosu. Natomiast jedynym problemem pozostaje rozpoznanie miejsca, w którym kończy się α, a zaczyna jego lustrzane odbicie. W tym miejscu automat musi przełączyć się z



Rys. 4.4: Automat ze stosem akceptujący język z Przykładu 4.9.

trybu rejestracji liter do trybu ich porównywania ze stosem. Ponieważ ma on do dyspozycji tylko jeden przebieg przez dane wejściowe, nie można wyznaczyć tego punktu przez liczenie znaków. Rozwiązanie uzyskamy dzięki niedeterminizmowi automatu: fazę rejestracji liter można przerwać w dowolnej chwili pustym przejściem do stanu s_2 rozpoczynającego porównywanie, Rys. 4.4. Jeśli przejście to zostanie wykonane w niewłaściwym momencie, automat nie osiągnie konfiguracji akceptującej, ważne jest natomiast, że dla poprawnego słowa z L (i tylko dla takiego słowa) przejście wykonane dokładnie po odczytaniu ostatniej litery α pozwoli na ukończenie całego procesu w stanie finalnym. A więc dla poprawnych strukturalnie słów *istnieje* akceptująca sekwencja ruchów automatu.

Definicja deterministycznego automatu ze stosem wymaga, aby relacja przejścia π była jednoznacznym częściowym odwzorowaniem.

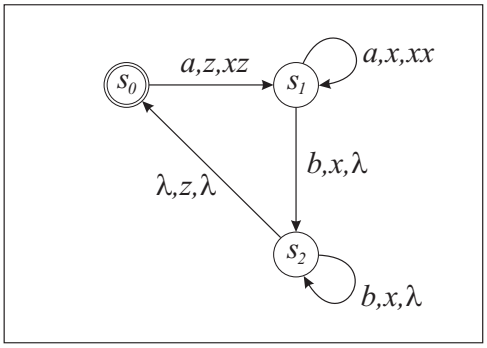
DEFINICJA 4.10 *Automat ze stosem opisany w Definicji 4.8 jest deterministyczny (DAS) jeśli relacja przejścia π jest częściową funkcją, a więc posiada następujące własności:*

(i) dla każdej trójki $(s, a, x) \in \Sigma \times A \times V$ istnieje co najwyżej jedna para $(s', \gamma) \in \Sigma \times V^*$ taka, że $\pi(s, a, x) = (s', \gamma)$;

(ii) jeśli puste przejście $\pi(s, \lambda, x)$ jest określone dla pewnych s i x, wtedy $\pi(s, a, x)$ nie może być określone dla jakiejkolwiek litery a.

Tak więc, o ile (ii) dopuszcza istnienie λ-przejść w deterministycznym automacie ze stosem (przypomnijmy, że puste przejścia były zabronione w przypadku skończonych automatów deterministycznych), dzieje się to pod warunkiem, że aktualny stan s i wierzchni symbol stosu x nie pozwalają na żadne inne przejście z ruchem głowicy czytającej. W ten sposób w każdej sytuacji automat ma co najwyżej jedno dostępne przejście, z ruchem głowicy czytającej albo bez.

Automat z Przykładu 4.8 można zamienić na równoważny deterministyczny, por. Rys. 4.3 i 4.5. Wystarczy zlikwidować stan s_4 i przyjąć s_0 jako stan początkowy i końcowy zarazem. Można natomiast udowodnić, że nie istnieje równoważny automat deterministyczny dla automatu z Przykładu 4.9 (por. jednak Zad. 19 na końcu obecnego rozdziału). Okazuje się bowiem, że klasa języków akceptowanych



Rys. 4.5: Deterministyczny automat ze stosem akceptujący język $\{a^n b^n : n \geqslant 0\}$ z Przykładu 4.8.

przez deterministyczne automaty ze stosem $\mathcal{L}_{\mathrm{Det}}$ jest węższa niż klasa języków rozpoznawanych przez automaty niedeterministyczne,

$$\mathcal{L}_{\mathrm{Det}} \subsetneq \mathcal{L}_{\mathrm{Ndet}} .$$

Oto inny sugestywny przykład języka klasy $\mathcal{L}_{\mathrm{Ndet}}$. Niech

$$L = L_1 \cup L_2 = \{a^n b^n : n \geqslant 0\} \cup \{a^m b^{2m} : m \geqslant 0\} .$$

Mając dwa deterministyczne automaty ze stosem akceptujące L_1 i L_2 o stanach startowych $s_0^{(1)}$ i odpowiednio $s_0^{(2)}$, tworzymy nowy automat sumując rozłączne z założenia zbiory Σ_1 i Σ_2, V_1 i V_2 oraz funkcje przejścia π_1 i π_2, a następnie dodając nowy stan startowy s_0 i niedeterministyczne przejścia $\pi(s_0, \lambda, z) = \{(s_0^{(1)}, z), (s_0^{(2)}, z)\}$. Jest to więc niedeterministyczny automat ze stosem akceptujący L. Problem ze stworzeniem dla niego wersji deterministycznej polega na tym, że niezależnie od wybranej metody najpóźniej w chwili gdy pod głowicą pojawi się pierwsza litera b trzeba zdecydować się na określony sposób operowania danymi zapisanymi w stosie — czy do każdej wczytanej wcześniej litery a dobierać jedną czy też dwie litery b. Od decyzji tej zależy poprawność akceptacji słów jeśli założymy deterministyczne działanie automatu. Ponieważ jednak nie możemy wiedzieć w tym momencie ile będzie liter b, decyzja taka musi być podjęta niedeterministycznie.

Zakończymy naszą dyskusję ważnym twierdzeniem.

TWIERDZENIE 4.4 *Klasa języków bezkontekstowych jest identyczna z klasą języków akceptowanych przez niedeterministyczne automaty ze stosem,*

$$\mathcal{L}_2 = \mathcal{L}_{\mathrm{Ndet}} .$$

Dowód tego twierdzenia pomijamy, wspominając jedynie, że podobnie jak w przypadku dualizmu "gramatyka–automat" dla języków regularnych, istnieje konstruktywna metoda przepisania produkcji gramatyki bezkontekstowej na przejścia automatu i odwrotnie. Wymagane jest jednak wcześniejsze przekształcenie gramatyki do postaci normalnej Greibach, a sama konstrukcja jest dużo bardziej złożona.

Języki rozpoznawane przez deterministyczne automaty ze stosem, zwane także *bezkontekstowymi językami deterministycznymi* opiszemy pokrótce w następnym podrozdziale.

2 Deterministyczne języki bezkontekstowe i ich gramatyki

Nie jest znana jednolita postać gramatyk generujących bezkontekstowe języki deterministyczne. Języki te są natomiast na tyle ważne z punktu widzenia zastosowań, w szczególności w konstrukcji kompilatorów języków programowania, że warto pokusić się przynajmniej o częściową ich charakteryzację przez czytelne gramatyki. Nietrudno domyślić się, że determinizm jest własnością ułatwiającą budowanie algorytmów parsingu o znacznie większej efektywności niż w przypadku języków niedeterministycznych. Zauważmy, że już zwykła symulacja automatu jest o wiele prostsza w przypadku deterministycznym.

Determinizm w przypadku gramatyki to cecha, która umożliwia jednoznaczny wybór produkcji w kolejnych krokach wyprowadzenia, którego celem jest wygenerowanie zadanego słowa: zamiast odtwarzania całego drzewa wyprowadzeń lub pewnej jego części, konstruujemy jedynie określoną ścieżkę prowadzącą do tego słowa. Algorytm parsingu może więc działać bez nawrotów. Najprostszym przykładem są wyprowadzenia top-down w tzw. s-gramatykach (s- od słowa "simple").

DEFINICJA 4.11 *G jest s-gramatyką, jeśli wszystkie jej produkcje mają postać*

$$X \to a\,Y_1 \ldots Y_m, \qquad \textit{gdzie} \quad a \in A, \quad Y_i \in V, \quad m \geqslant 0\,, \tag{4.18}$$

przy czym dla każdej pary X i a istnieje co najwyżej jedna taka produkcja.

Zauważmy, że (4.18) to postać normalna Greibach gramatyki, do której, jak już wiemy z Twierdzenia 4.3, przekształcić można dowolną gramatykę bezkontekstową, natomiast istotny jest tu warunek unikalności produkcji. Jeśli próbujemy wygenerować zadane słowo wejściowe $\alpha = a_1 a_2 \ldots a_n$, tworzymy wyprowadzenie lewostronne zaczynając od symbolu S. Pierwszy krok wyznaczony jest jednoznacznie przez parę symboli S i a_1: istnieje bowiem co najwyżej jedna produkcja postaci $S \to a_1 Y_1 \ldots Y_m$. Tak więc $S \Rightarrow a_1 Y_1 \ldots Y_m$. Kolejny krok to zastąpienie Y_1 przy pomocy odpowiedniej produkcji. Tu ponownie podgląd pary Y_1 i a_2 z napisu wejściowego jednoznacznie ją określa i kolejna forma zdaniowa ma przykładową postać $a_1 a_2 Z_1 \ldots Z_k Y_2 \ldots Y_m$. Jeśli nie istnieje taka produkcja, możemy odrzucić α jako nienależące do języka $L(G)$. W przeciwnym wypadku kontynuujemy proces z Z_1 i a_3 i tak dalej do końca wyprowadzenia.

Widzimy więc, że dla s-gramatyk podgląd kolejnych liter słowa wejściowego steruje deterministycznie procesem wyprowadzenia, przerywając je w przypadku napotkania pierwszej niezgodności. Można łatwo zaimplementować taki proces posługując się tablicą podglądu produkcji, indeksowaną przez pary symboli X i a, np. $T[X, a] = Y_1 \ldots Y_m$ jeśli parze X i a odpowiada produkcja $X \to aY_1 \ldots Y_m$.

S-gramatyki stanowią jednak zbyt ograniczoną kategorię, bowiem generowane przez nie języki tworzą w sumie niewielką rodzinę wśród języków deterministycznych. Bardziej ogólną konstrukcją są tzw. gramatyki typu $LL(k)$. Różnica polega na tym, że oprócz dostępu do aktualnie skanowanej litery zezwalamy tym razem na podgląd $k-1$ symboli napisu wejściowego na prawo od punktu skanowania, a więc na podgląd "wyprzedzający". Skrót LL to określenie "left-left", które oznacza, że symbole napisu wejściowego czytane są od lewej do prawej strony i odpowiednie

ich grupy sterują przebiegiem lewostronnych wyprowadzeń. Jeśli np. aktualnie skanowaną literą napisu wejściowego jest a_i oraz aktualnie zastępowanym symbolem nieterminalnym w lewostronnym wyprowadzeniu jest X, wówczas podgląd dalszych liter $a_{i+1}, \ldots, a_{i+k-1}$ (lub mniejszej ich liczby jeśli skanujemy w pobliżu końca słowa) pozwala jednoznacznie wybrać właściwą produkcję $X \to \beta$ lub stwierdzić, że taka nie istnieje i tym samym odrzucić analizowane słowo. Wszystkie s-gramatyki są z definicji typu $LL(1)$. Precyzyjna definicja gramatyki $LL(k)$ jest następująca:

DEFINICJA 4.12 *Gramatyka bezkontekstowa $G = (A, V, S, \Pi)$ jest typu $LL(k)$, jeśli dla każdej pary wyprowadzeń lewostronnych postaci*

$$\begin{aligned} S &\Rightarrow^* \alpha_1 X \omega_1 \Rightarrow \alpha_1\beta_1\omega_1 \Rightarrow^* \alpha_1\alpha_2 \\ S &\Rightarrow^* \alpha_1 X \omega_2 \Rightarrow \alpha_1\beta_2\omega_2 \Rightarrow^* \alpha_1\alpha_3\,, \end{aligned}$$

gdzie $\alpha_i \in A^*$, $\beta_i, \omega_i \in (A \cup V)^*$, równość co najwyżej k początkowych liter w α_2 i α_3 pociąga za sobą równość $\beta_1 = \beta_2$.

Tak więc wybór jednej z możliwych produkcji $X \to \beta_i$ jest jednoznacznie określony przez nie więcej niż k liter występujących w α tuż za jego początkową, przetworzoną już częścią α_1. Ujmując to inaczej, podgląd kolejnych k liter napisu wejściowego "steruje" procesem wyprowadzenia, wybierając właściwą ścieżkę w drzewie parsingu top-down.

PRZYKŁAD 4.10 Gramatyka $S \to aSb \,|\, ab$ nie jest s-gramatyką, bowiem prawe strony obydwu produkcji zastępujących symbol S rozpoczynają się literą a (pomijając mniej istotny i łatwy do naprawienia defekt, że zamiast b powinniśmy użyć pomocniczego symbolu B i dodatkowej produkcji $B \to b$). Jest ona natomiast gramatyką typu $LL(2)$: jeśli skanowany symbol to a i następny to także a, wybieramy produkcję $S \to aSb$. Jeśli natomiast następny symbol to b, wybieramy $S \to ab$. Proponujemy czytelnikowi przeprowadzenie wyprowadzenia napisu $aaabbb$ przy użyciu sterującej tabelki

	aa	ab	ba	bb
S	aSb	ab	$\emptyset$	$\emptyset$

której kolumny wskazują jakiej produkcji należy użyć w wyprowadzeniu do zastąpienia symbolu S, gdy symbole wejściowe a_i oraz a_{i+1} to odpowiednio aa, ab itd. Symbol $\emptyset$ oznacza brak odpowiedniej produkcji i sygnał do odrzucenia słowa.

Istnieje również s-gramatyka dla rozważanego tu języka:

$$\begin{aligned} S &\to aA \\ A &\to aAB \,|\, b \\ B &\to b \end{aligned}$$

Nie należy jednak sądzić, że jest to sytuacja typowa: fakt, że dla danego języka znamy gramatykę typu $LL(k)$ nie oznacza, że istnieje dla niego równoważna gramatyka typu $LL(k-1)$ lub niższego. Ogólny obraz scharakteryzujemy w dalszej części tego podrozdziału.

PRZYKŁAD 4.11 Gramatyka języka poprawnie zagnieżdżonych nawiasów $S \to ab \,|\, aSb \,|\, SS$ nie jest typu $LL(k)$ dla żadnego k. Przy analizowaniu napisów o długości

przekraczającej 2 mamy do wyboru dwie produkcje: $S \to aSb$ oraz $S \to SS$. Skanowany aktualnie symbol nie podpowiada, której z nich powinniśmy użyć, podobnie jak podgląd dowolnej liczby symboli wejściowych. Weźmy bowiem słowa

$$a^n b^n \qquad \text{oraz} \qquad a^n b^n ab\,. \tag{4.19}$$

Wyprowadzenie pierwszego z nich polega na iteracyjnym użyciu produkcji $S \to aSb$, natomiast wyprowadzenie drugiego *musi* zaczynać się od $S \to SS$. Decyzję co do tego, której produkcji użyć w wyprowadzeniu jako pierwszej można podjąć na podstawie podglądu *nie mniej* niż $2n$ początkowych symboli wejściowych. Ponieważ jednak w naszej gramatyce możemy wyprowadzać takie napisy dla dowolnie dużych n, nie może być ona typu $LL(k)$ dla jakiegokolwiek ustalonego k.

Nie znaczy to jednak, że język poprawnie zagnieżdżonych nawiasów nie może być generowany przez gramatykę typu LL. Oto bowiem niemal równoważna gramatyka typu $LL(1)$ dla tego języka:

$$S \to aSbS \,|\, \lambda\,.$$

Niemal, ponieważ generuje ona dodatkowo słowo puste, którego nie włączyliśmy do oryginalnego języka. Zajmiemy się tym drobnym defektem za chwilę. Analizując przykład słowa *aabbab* zauważmy, że skanowanie wejściowej litery a nakazuje jednoznacznie wybrać produkcję $S \to aSbS$, natomiast w przypadku litery b lub końca napisu — produkcję $S \to \lambda$. Oto odnośne wyprowadzenie, w którym nad symbolami $\Rightarrow$ podajemy aktualnie skanowaną literę wejściową:

$$S \overset{a}{\Rightarrow} aSbS \overset{a}{\Rightarrow} aaSbSbS \overset{b}{\Rightarrow} aabSbS \overset{b}{\Rightarrow} aabbS \overset{a}{\Rightarrow} aabbaSbS \overset{b}{\Rightarrow} aabbabS \overset{\lambda}{\Rightarrow} aabbab\,.$$

By zagwarantować pełną równoważność z pierwotną gramatyką, pozostaje wyeliminować z generowanego języka słowo puste przez wymuszenie, by każde wyprowadzenie rozpoczynało się od zastosowania produkcji $S \to aSbS$. Skorygowana gramatyka przedstawia się następująco:

$$S \to aXbX\,, \qquad X \to aXbX \,|\, \lambda\,.$$

Przykład 4.12 Innym ważnym z punktu widzenia zastosowań przykładem jest język prostych wyrażeń arytmetycznych. Gramatyka (4.6) nie jest gramatyką typu LL z uwagi na obecność par produkcji $S \to T$ oraz $S \to T{+}S$ i podobnie $T \to F$ oraz $T \to F * T$. Zarówno składnik T jak i czynnik F mogą rozwijać się w konkretnym wyprowadzeniu do podwyrażeń o dowolnej długości, a więc podgląd wyprzedzający ustalonej liczby k znaków wejściowych nie w każdym przypadku pozwoli nam stwierdzić czy oczekiwać np. znaku + po T, a więc czy użyć produkcji $S \to T + S$ czy też $S \to T$. Zauważmy, że jest to schemat identyczny z tym, który analizowaliśmy w poprzednim przykładzie dla napisów (4.19). Podobnie jak poprzednio, gramatykę (4.6) można przekształcić do równoważnej postaci posiadającej już własność $LL(1)$:

$$\begin{array}{lcllcl} S & \to & TR & R & \to & +TR \,|\, \lambda \\ T & \to & FQ & Q & \to & *FQ \,|\, \lambda \\ F & \to & x \,|\, y \,|\, z \,|\, (S)\,. & & & \end{array} \tag{4.20}$$

Proces parsingu sterowany jest tabelą podglądu podobną do tej, której użyliśmy w Przykładzie 4.10. Jednak tym razem omówimy ten proces bardziej szczegółowo. Algorytm deterministyczny bazujący na gramatyce typu $LL(k)$ z jednej strony skanuje napis wejściowy z odpowiednim wyprzedzającym podglądem, z drugiej zaś operuje na formie zdaniowej wyprowadzonej do tej pory z symbolu S, dokonując w niej stosownych podstawień na podstawie produkcji. By zrealizować proces lewostronnego wyprowadzania wygodnie jest wykorzystać stos do przechowania formy zdaniowej, w którym jego wierzchni symbol odpowiada początkowemu znakowi formy. Jeśli jest to nieterminal, będzie on zdjęty ze stosu i zastąpiony prawą stroną właściwej produkcji (wyznaczonej jednoznacznie przez tabelę podglądu), przy czym jej znaki wpisywane są do stosu od ostatniego, by zachować wspomnianą wyżej konwencję "wierzch stosu = początek formy". Jeśli zaś wierzchni element stosu to terminal, zdejmujemy go jeśli tylko zgadza się ze skanowanym aktualnie symbolem wejściowym. W tym wypadku przesuwamy także punkt skanowania. Brak zgodności terminala na stosie z symbolem wejściowym oznacza błąd. Proces rozpoczyna się od umieszczenia w pustym stosie symbolu S i ustawieniu punktu skanowania na pierwszej literze. W przypadku gdy napis wejściowy jest gramatycznie poprawny, powinniśmy dojść do konfiguracji z pustym stosem i punktem skanowania za ostatnią literą. Zwracamy uwagę na pewne podobieństwo tego algorytmu z opisem działania automatu ze stosem — to właśnie użycie stosu w procesie rozpoznawania słów jest charakterystyczną cechą języków bezkontekstowych.

Poniżej podajemy tabelę podglądu dla gramatyki (4.20). Kolumny indeksowane są pojedynczymi (zgodnie z typem $LL(1)$) symbolami wejściowymi, wiersze zaś symbolami na szczycie stosu. Tabela zawiera prawe strony odpowiednich produkcji, które należy zastosować zastępując nieterminal na wierzchu stosu indeksujący dany wiersz, a także symbole akcji "pop" oznaczające usunięcie symbolu (terminalnego) ze stosu i przesunięcie punktu skanowania napisu wejściowego. Puste pole w tabeli oznacza konfigurację błędną. Stan pustego stosu oznaczyliśmy symbolem Ø.

	x	y	z	$+$	$*$	$($	$)$	λ
S	TP	TP	TP			TP		
P				$+TP$			λ	λ
T	FQ	FQ	FQ			FQ		
Q				λ	$*FQ$		λ	λ
F	x	y	z			(S)		
x	pop							
y		pop						
z			pop					
$+$				pop				
$*$					pop			
$($						pop		
$)$							pop	
Ø								accept

Dla przykładu przeprowadzimy analizę wyrażenia $x * (y + z)$ sterowaną powyższą tabelą. Oto kolejne kroki wyprowadzenia (stany stosu), podjęte akcje i stany napisu wejściowego od punktu skanowania:

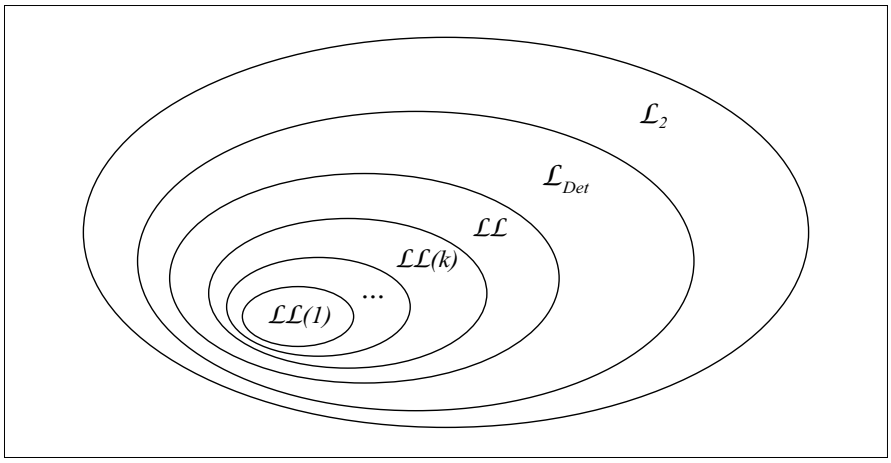
Stos	Akcja	Wejście
S	$S \to TP$	$x*(y+z)$
TP	$T \to FQ$	$x*(y+z)$
FQP	$F \to x$	$x*(y+z)$
xQP	pop	$x*(y+z)$
QP	$Q \to *FQ$	$*(y+z)$
$*FQP$	pop	$*(y+z)$
FQP	$F \to (S)$	$(y+z)$
$(S)QP$	pop	$(y+z)$
$S)QP$	$S \to TP$	$y+z)$
$TP)QP$	$T \to FQ$	$y+z)$
$FQP)QP$	$F \to y$	$y+z)$
$yQP)QP$	pop	$y+z)$
$QP)QP$	$Q \to \lambda$	$+z)$
$P)QP$	$P \to +TP$	$+z)$
$+TP)QP$	pop	$+z)$
$TP)QP$	$T \to FQ$	$z)$
$FQP)QP$	$F \to z$	$z)$
$zQP)QP$	pop	$z)$
$QP)QP$	$Q \to \lambda$	$)$
$P)QP$	$P \to \lambda$	$)$
$)QP$	pop	$)$
QP	$Q \to \lambda$	λ
P	$P \to \lambda$	λ
$\emptyset$	accept	λ

Zakończymy niniejszy rozdział omówieniem kilku ważnych faktów dotyczących języków deterministycznych. Zdefiniujmy klasę $\mathcal{LL}(k)$ języków bezkontekstowych jako zbiór tych L, dla których istnieje gramatyka G typu $LL(k)$ generująca ten język, $L = L(G)$. Ponieważ gramatyka typu $LL(k)$ jest trywialnie także typu $LL(k+1)$ (wystarczy jeśli zignorujemy ostatni z podglądanych symboli wejściowych), mamy

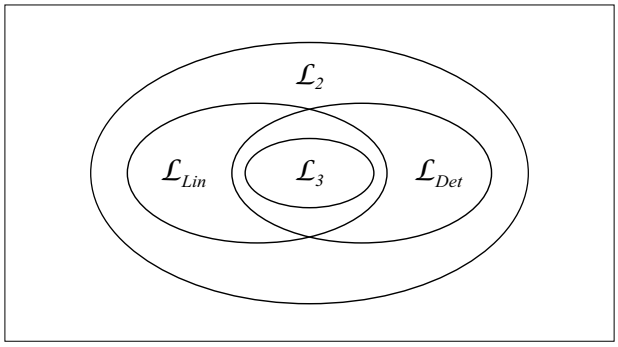
$$\mathcal{LL}(k) \subseteq \mathcal{LL}(k+1)\,.$$

Mniej oczywisty jest fakt, że inkluzja ta jest właściwa, $\mathcal{LL}(k) \subsetneq \mathcal{LL}(k+1)$. W literaturze znane są przykłady języków klasy $\mathcal{LL}(k)$, które nie są klasy $\mathcal{LL}(k-1)$, por. np. [1], vol. II. Ponadto istnieją przykłady języków deterministycznych, które nie należą do żadnej z klas $\mathcal{LL}(k)$. Tak więc ogólną klasyfikację języków deterministycznych w klasie języków bezkontekstowych przedstawić można diagramem z Rys. 4.6. W uzupełnieniu dodajmy, że także zawarta w $\mathcal{L}_2$ klasa języków liniowych $\mathcal{L}_{\mathrm{Lin}}$, czyli języków generowanych przez gramatyki liniowe, nie jest porównywalna z żadną z klas $\mathcal{LL}(k)$ ani też z $\mathcal{L}_{\mathrm{Det}}$. Dla czytelności, osobny diagram przedstawiony na Rys. 4.7 ilustruje wzajemne relacje między językami regularnymi, liniowymi i deterministycznymi.

Oprócz gramatyk typu LL rozważa się także inne ich rodziny, które umożliwiają deterministyczny parsing. Na szczególną uwagę zasługują gramatyki tzw. typu



Rys. 4.6: Hierarchia $\mathcal{LL}(k)$ wewnątrz klasy języków deterministycznych i bezkontekstowych. Symbolem $\mathcal{LL}$ oznaczyliśmy sumę wszystkich klas $\mathcal{LL}(k)$. Czytelnik zechce samodzielnie uzasadnić, że języki regularne stanowią podklasę $\mathcal{LL}(1)$, por. Zad. 20 na końcu rozdziału.



Rys. 4.7: Relacje między językami regularnymi, liniowymi i deterministycznymi. Przykładem nieliniowego języka deterministycznego jest $L = \{\omega : \ |\omega|_a = |\omega|_b\}$, natomiast język $L = \{a^n b^n\} \cup \{a^n b^{2n}$ jest liniowy lecz niedeterministyczny.

$LR(k)$, w których deterministyczny wybór produkcji możliwy jest jak poprzednio na podstawie wyprzedzającego podglądu kolejnych $k-1$ symboli wejściowych. Tym razem jednak analiza oparta jest na prawostronnych wyprowadzeniach. Gramatyki te stanowią bazę dla efektywnych algorytmów parsingu "bottom-up", w których wyprowadzenia przeprowadzane są w przeciwną stronę i polegają na redukcji fragmentów formy zdaniowej do pojedynczych nieterminali, a więc na użyciu produkcji w odwrotny sposób: odnaleziona w formie zdaniowej prawa strona produkcji zastępowana jest przez symbol z lewej strony. W końcowym efekcie ciągu redukcji powinniśmy otrzymać pojedynczy symbol S. Parsing działa tu w wielu aspektach dokładnie odwrotnie do metody opartej na gramatykach $LL(k)$: w wersji LL stos początkowo zawiera symbol S a na końcu jest pusty, natomiast w wersji LR na odwrót, najpierw jest pusty, a na końcu powinien zawierać symbol S. Gramatyki tego typu i szczegółowe metody parsingu omówione są wyczerpująco we wspomnianej książce [1].

IV.3 Własności języków bezkontekstowych

Omówimy obecnie kilka ważnych własności języków bezkontekstowych. Zajmiemy się w szczególności zamkniętością $\mathcal{L}_2$ ze względu na operacje mnogościowe i inne algebraiczne operacje na językach. Okazuje się, że inaczej niż w przypadku języków regularnych, wiele takich operacji wyprowadza poza klasę $\mathcal{L}_2$. Także lista pozytywnie rozstrzygalnych problemów decyzyjnych jest uboższa dla języków bezkontekstowych. W szczególności istotny problem rozpoznawania równoważności gramatyk nie jest w klasie $\mathcal{L}_2$ rozstrzygalny.

Rozpoczniemy od przestawienia dwóch lematów o pompowaniu: dla języków liniowych i dla ogólnych języków bezkontekstowych. Wnioski z tych lematów pozwolą nam sformułować ważne negatywne wyniki dotyczące operacji na językach w $\mathcal{L}_2$.

W dalszej części rozdziału omówimy inne, mniej znane charakteryzacje języków bezkontekstowych. Jednym z nich jest interesujące twierdzenie Parikha, które stanowi swego rodzaju arytmetyczną charakteryzację języków z $\mathcal{L}_2$, odwołującą się do liczby wystąpień symboli terminalnych w ich słowach.

1 Lematy o pompowaniu dla języków bezkontekstowych

Rozpocznijmy od przedstawienia lematu o pompowaniu w wersji dla języków liniowych. Przypomnijmy, że język liniowy generowany jest przez gramatykę bezkontekstową, w której prawe strony produkcji zawierają co najwyżej po jednym symbolu nieterminalnym.

LEMAT 4.1 (O pompowaniu dla języków liniowych) *Niech L będzie nieskończonym językiem liniowym. Istnieje wówczas stała $m > 0$ (zależna jedynie od L) taka, że dowolne słowo $\omega \in L$ o długości co najmniej m można rozłożyć na części $\omega = \alpha\beta\gamma\delta\eta$ w taki sposób, że*

(i) $|\alpha\beta\delta\eta| \leqslant m$,

(ii) $|\beta\delta| \geqslant 1$,

a przy tym wszystkie bez wyjątku słowa $\omega_k = \alpha\beta^k\gamma\delta^k\eta$ dla $k = 0, 1, 2, \ldots$ są także elementami L.

Dowód tego lematu pomijamy, por. np. [7]. Jest on w swojej strukturze podobny do dowodu ogólniejszego lematu o pompowaniu dla języków bezkontekstowych, który naszkicujemy niżej.

PRZYKŁAD 4.13 Zastosujemy powyższy lemat do pokazania, że język $L = \{\omega \in \{a, b\}^* : |\omega|_a = |\omega|_b\}$ z Przykładu 2.5 nie jest liniowy. Załóżmy nie wprost, że istnieje generująca go liniowa gramatyka. Zgodnie z Lematem 4.1, dla L istnieje stała m wyznaczająca długość słów ω, dla których zawsze możliwe jest pompowanie. Weźmy zatem jako ω słowo postaci $a^m b^{2m} a^m$. Lemat gwarantuje dalej istnienie podziału $\omega = \alpha\beta\gamma\delta\eta$, dla którego nierówność (i) oznacza w obecnej sytuacji, że części α, β, δ i η zbudowane są wyłącznie z liter a, natomiast (ii) zapewnia, że pompowane części β i δ zawierają co najmniej jedną literę a. Część centralna słowa ω zbudowana z liter b ukryta jest w podciągu γ i nie podlega pompowaniu. Tak więc liczba liter

a w słowach ω_k wzrasta, podczas gdy liczba liter b pozostaje niezmieniona. Tym samym pompowanie tworzy słowa spoza L. Otrzymaliśmy sprzeczność z tezą lematu, a zatem założenie o liniowości języka L musiało być fałszywe.

Ostatni przykład pokazuje, że $\mathcal{L}_{\mathrm{Lin}} \subsetneq \mathcal{L}_2$. Przejdźmy do sformułowania analogicznego lematu dla ogólnych języków bezkontekstowych.

LEMAT 4.2 (O pompowaniu dla jęz. bezkontekstowych) *Niech L będzie nieskończonym językiem bezkontekstowym. Istnieje wówczas stała $m > 0$ (zależna jedynie od L) taka, że dowolne słowo $\omega \in L$ o długości co najmniej m można rozłożyć na części $\omega = \alpha\beta\gamma\delta\eta$ w taki sposób, że*

(i) $|\beta\gamma\delta| \leqslant m$,

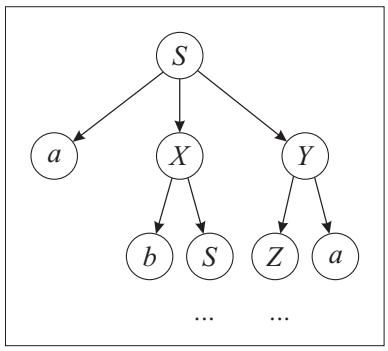
(ii) $|\beta\delta| \geqslant 1$,

a przy tym wszystkie bez wyjątku słowa $\omega_k = \alpha\beta^k\gamma\delta^k\eta$ dla $k = 0, 1, 2, \ldots$ są także elementami L.

Zauważmy, że różnica między obydwoma wersjami lematu jest raczej subtelna: ogranicza się ona do innej postaci nierówności (i). W przypadku języków liniowych warunek (i) gwarantuje, że części β i δ podlegające pompowaniu znajdują się w obrębie m znaków od końców napisu ω, a więc "blisko jego brzegów". Rozdzielająca je część "centralna" γ może być dowolnie długa. Widać więc celowość takiego a nie innego wyboru słowa ω w Przykładzie 4.13: skoro naszym celem jest obalenie przypuszczenia o liniowości L, wybieramy słowo ω tak, by pompowanie generowało słowa spoza L, w szczególności by powielało tylko litery a, nie zmieniając liczby liter b i naruszając tym samym warunek $|\omega|_a = |\omega|_b$ definiujący L.

Odwrotnie, w wersji lematu dla języków bezkontekstowych części podlegające pompowaniu leżą "niedaleko" od siebie, zawsze w obrębie m znaków niezależnie od wyboru słowa ω, a części początkowa i końcowa α i η mogą tym razem być długie. Ta obserwacja także pomoże nam wybierać celowo słowa ω, które łatwo prowadzą do obalenia hipotezy o bezkontekstowości analizowanego przy pomocy lematu języka.

SZKIC DOWODU. Skoro język L jest klasy $\mathcal{L}_2$, posiada generującą go gramatykę bezkontekstową G. Załóżmy bez zmniejszenia ogólności, że gramatyka ta nie zawiera pustych ani też jednostkowych produkcji. Tym razem przedstawimy wyprowadzenie $S \Rightarrow^* \omega$ w postaci drzewa, w którym wierzchołki reprezentują symbole terminalne i nieterminalne, a rozgałęzienia do kolejnych pięter odpowiadają użytym produkcjom, np.



Zastosowane w tym przykładzie produkcje to $S \to aXY$, $X \to bS$ i $Y \to Za$. Niech q oznacza największą długość prawej strony produkcji w G. Jeśli m jest długością słowa $\omega \in L$, wówczas $\log_q m$ jest dolnym oszacowaniem wysokości drzewa reprezentującego wyprowadzenie $S \Rightarrow^* \omega$. Rzeczywiście, gdyby w każdym kroku wyprowadzenia używano produkcji z q-literową prawą stroną, pierwsze piętro drzewa zawierałoby q znaków, drugie q^2 znaków itd., a więc l-te q^l znaków. Biorąc $q^l \geqslant m$, otrzymujemy $l \geqslant \log_q m$.

Ponieważ L jest z założenia nieskończony, długości słów są w nim nieograniczone. Wybierzmy więc słowo ω o dostatecznie dużej długości m, tak aby $\log_q m > p$, gdzie p jest liczbą nieterminali w G. Zatem wysokość l drzewa wyprowadzenia ω spełnia nierówność $l > p$. Istnieje więc ścieżka w tym drzewie o kolejnych wierzchołkach nieterminalnych $S = X_1, X_2, \ldots, X_l$, a ponieważ $l > p$, pewien nieterminal pojawia się w niej więcej niż raz. Niech X będzie takim powtarzającym się nieterminalem, $X = X_i = X_j$, $j > i$. W zwykłym zapisie wyprowadzenia mamy więc

$$S \Rightarrow^* \alpha X \eta \Rightarrow^* \alpha\beta X \delta\eta \Rightarrow^* \alpha\beta\gamma\delta\eta \;=\; \omega\,,$$

przy czym szczegółowe kroki tego wyprowadzenia zawsze przeprowadzić można w takiej kolejności, żeby α, β, δ i η były napisami terminalnymi. Oznacza to, że $X \Rightarrow^* \beta X \delta$ oraz $X \Rightarrow^* \gamma$. Legalne są zatem także następujące wyprowadzenia:

$$S \Rightarrow^* \alpha X \eta \Rightarrow^* \alpha\gamma\eta$$

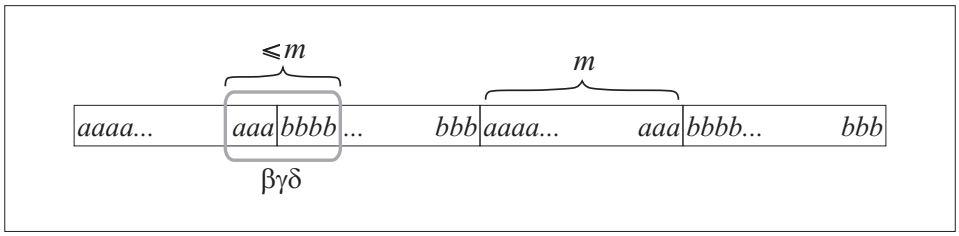
oraz

$$S \Rightarrow^* \alpha X \eta \Rightarrow^* \alpha\beta^k X \delta^k \eta \Rightarrow^* \alpha\beta^k\gamma\delta^k\eta$$

dla $k = 1, 2, \ldots$, a więc wszystkie słowa $\omega_k = \alpha\beta^k\gamma\delta^k\eta$ dla $k = 0, 1, 2, \ldots$ są elementami L. Oszacowanie (i) w lemacie uzasadnić można przez bardziej precyzyjny wybór powtarzającego się nieterminala (chodzi o taki zawsze wykonalny wybór $X = X_i = X_j$, aby w poddrzewach zakorzenionych w X_i i X_j nie występowały już inne powtórzenia nieterminali). Nie będziemy jednak wchodzić w szczegóły tego technicznego argumentu. Nierówność (ii) jest konsekwencją faktu, że gramatyka G nie zawiera ani pustych ani też jednostkowych produkcji: skoro $X \Rightarrow^* \beta X \delta$, zatem $|\beta\delta| > 0$. □

Zilustrujemy kilkoma przykładami przydatność lematu o pompowaniu do obalania przypuszczeń co do bezkontekstowości pewnych języków.

Przykład 4.14 Standardowym przykładem języka, który nie spełnia tezy Lematu 4.2 jest $L = \{a^n b^n c^n : \; n \geqslant 0\}$. Załóżmy, że jest to język bezkontekstowy. Istnieje zatem stała m gwarantująca, że słowo $\omega = a^m b^m c^m$ dzięki swojej długości może być podzielone na segmenty $\omega = \alpha\beta\gamma\delta\eta$ w sposób zgodny z (i) oraz (ii) tak, że zastosowana do niego operacja pompowania $\omega_k = \alpha\beta^k\gamma\delta^k\eta$ generuje inne słowa z L. Jednak warunek (ii) $|\beta\gamma\delta| \leqslant m$ dla naszego słowa oznacza, że segment ten może być zbudowany z powtórzeń *co najwyżej dwóch* liter, a więc może być jednej z postaci a^r, $a^s b^t$, b^r, $b^s c^t$ lub ostatecznie c^r, gdzie $r \leqslant m$ lub $s + t \leqslant m$. Wówczas jednak pompowanie powiela tylko jedną lub dwie spośród liter a, b, c, produkując słowa spoza języka L. Stąd wniosek, że język ten nie może być bezkontekstowy.



Rys. 4.8: Ilustracja do Przykładu 4.15. Segment $\beta\gamma\delta$ ze względu na ograniczoną długość nie większą niż m może mieścić się w obrębie co najwyżej dwóch sąsiednich bloków liter a i b. W każdym z możliwych położeń pompowanie niszczy symetrię słów języka L.

PRZYKŁAD 4.15 Niech z kolei $L = \{\mu\mu : \ \mu \in \{a,b\}^*\}$. Gdyby język ten był bezkontekstowy, zgodnie z lematem istniałaby dla niego odpowiednia stała m. Jako robocze słowo $\omega \in L$ weźmy tym razem $a^m b^m a^m b^m$. Wybór tego słowa jak zwykle nie jest przypadkowy, chcemy bowiem ułatwić sobie analizę struktury napisów ω_k budowanych przez pompowanie. Warunek (i) oznacza, że jakkolwiek nie wyglądałby podział $\omega = \alpha\beta\gamma\delta\eta$, segment $\beta\gamma\eta$, ze względu na swoją długość nieprzekraczającą m, może być zlokalizowany w obrębie co najwyżej dwóch bloków liter a lub b, por. Rys. 4.8. Na podstawie tego rysunku widać dlaczego pompowanie w każdym możliwym położeniu bloku $\beta\gamma\delta$ niszczy strukturę $\mu\mu$ słów języka L. Formalny dowód jest bardzo prosty lecz powinien być przeprowadzony dla wszystkich szczegółowych przypadków. Np. gdy segment $\beta\gamma\delta$ mieści się w całości w obrębie jednego z czterech bloków liter a lub b, pompowanie generuje słowa postaci

$$\omega_k = a^{m+ks} b^m a^m b^m\,, \quad a^m b^{m+ks} a^m b^m\,, \quad a^m b^m a^{m+ks} b^m \quad \text{lub} \quad a^m b^m a^m b^{m+ks}\,,$$

gdzie $s > 0$ i $k = 1, 2, \ldots$. Żadne z nich nie należy do L. Natomiast konfiguracje takie jak przedstawiona na Rys. 4.8 wymagają osobnej, wariantowej ze względu na różne możliwe położenia β i δ, analizy.

PRZYKŁAD 4.16 Jako ostatni z przykładów rozważmy język $L = \{a^{n^2} : \ n \geqslant 0\}$, który analizowaliśmy już wcześniej, por. Przykład 3.17, uzasadniając dlaczego nie jest on regularny. Czytelnik zechce przekonać się, że dokładnie *tego samego* sposobu argumentacji użyć można również teraz, w kontekście lematu o pompowaniu dla języków bezkontekstowych, wnioskując, że L także nie jest klasy $\mathcal{L}_2$. Zbieżność ta nie jet przypadkowa: w podrozdziale IV.3.4 przekonamy się, że języki nad jednoliterowym alfabetem, jeśli nie są regularne, nie mogą być także bezkontekstowe.

2 Zamkniętość klasy języków bezkontekstowych

Jak wspomnieliśmy, wyniki dotyczące zamkniętości klasy $\mathcal{L}_2$ ze względu na operacje na językach są bardziej skromne niż w wypadku klasy $\mathcal{L}_3$.

LEMAT 4.3 *Klasa $\mathcal{L}_2$ jest zamknięta ze względu na sumę mnogościową, konkatenację i domknięcie konkatenacyjne. Jest także zamknięta ze względu na odbicie lustrzane i dowolny homomorfizm.*

DOWÓD. Załóżmy, że $L_1, L_2 \in \mathcal{L}_2$ są generowane przez gramatyki bezkontekstowe G_1 i G_2 o rozłącznych alfabetach nieterminalnych V_1 i V_2. Niech ponadto S_1 i S_2 będą ich symbolami startowymi. Wówczas, jak łatwo sprawdzić, następujące gramatyki bezkontekstowe generują odpowiednio języki $L_1 \cup L_2$, $L_1 L_2$ oraz L_1^*:

$$\begin{aligned} G_\cup &= \Big(A_1 \cup A_2,\ V_1 \cup V_2 \cup \{S\},\ S,\ \Pi_1 \cup \Pi_2 \cup \{S \to S_1 \,|\, S_2\}\Big), \\ G_{\text{kon}} &= \Big(A_1 \cup A_2,\ V_1 \cup V_2 \cup \{S\},\ S,\ \Pi_1 \cup \Pi_2 \cup \{S \to S_1 S_2\}\Big), \\ G_* &= \Big(A_1\ V_1 \cup \{S\},\ S,\ \Pi_1 \cup \{S \to S_1 S \,|\, \lambda\}\Big). \end{aligned}$$

Konstrukcja gramatyki generującej język $\overleftarrow{L}$ jest także bardzo prosta. Wystarczy zastąpić każdą produkcję $X \to \alpha$ produkcją "lustrzaną" $X \to \overleftarrow{\alpha}$. Fakt, że gramatyka taka generuje lustrzane odbicia słów $L(G)$ jest intuicyjnie oczywisty, gdybyśmy jednak chcieli podać bardziej formalny dowód, należałoby posłużyć się postacią Chomsky'ego dla G i przeprowadzić prosty argument indukcyjny względem długości wyprowadzenia.

W końcu, dla homomorfizmu $h : A \to B^*$ wystarczy, podobnie jak wyżej, użyć postaci Chomsky'ego gramatyki i zastąpić produkcje terminalne $X \to a$ produkcjami $X \to h(a)$. Jest oczywiste, że zmodyfikowany w ten sposób układ produkcji generuje język $h(L(G))$. □

Podobnie jak w przypadku języków regularnych, istnieje inny prosty sposób zwięzłego uzasadnienia zamkniętości klasy $\mathcal{L}_2$ ze względu na kilka spośród wymienionych wyżej operacji. W tym celu przywołamy opisaną w Rozdziale II.3, str. 31, operację podstawienia $\chi(L)$ indukowanego przez odwzorowanie $\chi(a) = L_a$. Nasze poprzednie twierdzenie jest w rzeczywistości w znacznej części prostą konsekwencją następującego, ogólniejszego faktu:

LEMAT 4.4 *Jeśli język L oraz wszystkie języki L_a, $a \in A$, są klasy $\mathcal{L}_2$, jest nim także język $\chi(L)$.*

DOWÓD. Przyjmijmy, że L generowany jest przez gramatykę G, a języki L_a — przez gramatyki G_a, wszystkie o rozłącznych zbiorach nieterminali i symbolach początkowych S oraz odp. S_a. Gramatykę G_χ dla podstawienia $\chi(L)$ uzyskamy zastępując w produkcjach G wszystkie symbole terminalne a przez odpowiednie symbole startowe S_a. Tak zmodyfikowany zbiór produkcji G uzupełniamy o produkcje ze wszystkich gramatyk G_a. □

Przejdźmy teraz do wyników negatywnych.

LEMAT 4.5 *Klasa $\mathcal{L}_2$ nie jest zamknięta ze względu na przekroje i dopełnienia języków, nie jest tym samym zamknięta ze względu na różnice i różnice symetryczne.*

DOWÓD. Poniższy prosty przykład pokazuje, że przekrój języków bezkontekstowych na ogół nie musi być językiem tej samej klasy. Niech

$$L_1 = \{a^n b^n c^m : m, n \geqslant 0\} \qquad \text{oraz} \qquad L_2 = \{a^m b^n c^n : m, n \geqslant 0\}.$$

L_1 i L_2 są oczywiście językami z $\mathcal{L}_2$, natomiast $L_1 \cap L_2 = \{a^n b^n c^n : n \geqslant 0\}$ nie jest językiem tej klasy, por. Przykład 4.13.

Fakt, że dopełnienie języka bezkontekstowego nie musi być typu $\mathcal{L}_2$ wynika nie wprost z tożsamości $(L_1^c \cup L_2^c)^c = L_1 \cap L_2$. Gdyby bowiem było inaczej, lewa jej strona musiałaby być językiem bezkontekstowym, co przeczyłoby uzyskanemu właśnie negatywnemu wynikowi dla przekrojów. Podobnie z równości

$$L_1 - (L_1 - L_2) \;=\; L_1 \cap L_2 \qquad \text{oraz} \qquad L_1 \oplus L_2 \oplus (L_1 \cup L_2) = L_1 \cap L_2$$

(por. Zad. 2 z Rozdziału I) wnioskujemy, że zarówno różnica jak i różnica symetryczna na ogół wyprowadzają poza klasę $\mathcal{L}_2$. □.

Kilka innych faktów dotyczących operacji na językach bezkontekstowych pojawi się w formie zadań na końcu rozdziału. Przytoczymy tu jeszcze jeden użyteczny rezultat.

LEMAT 4.6 *Jeśli L jest klasy $\mathcal{L}_2$ natomiast L_r jest regularny, wówczas $L \cap L_r$ jest językiem bezkontekstowym. Innymi słowy klasa $\mathcal{L}_2$ jest zamknięta ze względu na przekroje z językami regularnymi.*

DOWÓD. Uzasadnienie lematu uzyskujemy przez konstrukcję podobną do tej, której używamy do budowy automatu deterministycznego dla przekroju języków regularnych, por. (3.6), str. 68. Niech $M = (A, V, z, \Sigma, s_0, S_F, \pi)$ będzie niedeterministycznym automatem ze stosem dla L oraz niech $M_r = (A, \Sigma_r, r_0, R_F, \pi_r)$ będzie skończonym automatem deterministycznym akceptującym L_r. Utworzymy automat ze stosem $\hat{M}$ symulujący równoległe działanie M i M_r. Oznaczmy komponenty $\hat{M}$ jako

$$\hat{M} \;=\; \left(A, V, z, \hat{\Sigma}, \hat{s}_0, \hat{S}_F, \hat{\pi}\right).$$

Podstawą konstrukcji jest użycie odpowiednich iloczynów kartezjańskich

$$\hat{\Sigma} = \Sigma \times \Sigma_r\,, \qquad \hat{S}_F = S_F \times R_F\,, \qquad \hat{s}_0 = (s_0, r_0)\,.$$

Następnie definiujemy ruchy automatu $\hat{M}$:

$$\Big((s_k, r_l), \gamma\Big) \in \hat{\pi}((s_i, r_j), a, x) \qquad \text{jeśli} \qquad (s_k, \gamma) \in \pi(s_i, a, x) \quad \text{i} \quad \pi_r(r_j, a) = r_l\,.$$

Ponadto dla pustych przejść $(s_k, \gamma) \in \pi(s_i, \lambda, x)$ mamy

$$\Big((s_k, r_j), \gamma\Big) \in \hat{\pi}((s_i, r_j), \lambda, x)\,.$$

Tak więc pustym przejściom w M towarzyszy brak ruchu automatu M_r. Stosując indukcję można łatwo udowodnić, że

$$(s_0, r_0)\alpha z \;\Rightarrow^*_{\hat{M}}\; \alpha(s_f, r_f)\gamma \qquad \text{dla} \qquad s_f \in S_F \quad \text{i} \quad r_f \in R_F$$

wtedy i tylko wtedy, gdy

$$s_0 \alpha z \;\Rightarrow^*_M\; \alpha s_f \gamma \qquad \text{oraz} \qquad r_0 \alpha \;\Rightarrow^*_{M_r}\; \alpha\, r_f\,.$$

Stąd napis α jest akceptowany przez automat $\hat{M}$ wtedy i tylko wtedy, gdy jest jednocześnie akceptowany przez automaty M i M_r, a więc $L(\hat{M}) = L \cap L_r$. □

Poniższe przykłady ilustrują sposób wykorzystania ostatniego lematu.

PRZYKŁAD 4.17 Rozważmy język $L_{99} = \{a^n b^n : \ n \geqslant 0, \ n \neq 99\}$. Choć można byłoby zbudować odpowiedni automat ze stosem lub gramatykę bezkontekstową dla tego języka, konstrukcja taka byłaby dość żmudna. By przekonać się szybko, że L_{99} jest językiem bezkontekstowym, rozumujemy następująco. Jako L weźmy język $\{a^n b^n : \ n \geqslant 0\}$, natomiast jako L_r język jednoelementowy $\{a^{99}b^{99}\}$. Ponieważ L_r^c jest regularny, na podstawie Lematu 4.6 wnioskujemy, że $L_{99} = L \cap L_r^c$ jest bezkontekstowy.

PRZYKŁAD 4.18 Pokażemy teraz, że język

$$L \ = \ \left\{\alpha \in \{a,b,c\}^* : \ \ |\alpha|_a = |\alpha|_b = |\alpha|_c \right\}$$

nie jest bezkontekstowy. Jest to natychmiastowy wniosek z Lematu 4.6 jeśli zauważyć, że

$$L \ \cap \ L(a^*b^*c^*) \ = \ \{a^n b^n c^n : \ n \geqslant 0\}\,.$$

Gdyby bowiem L był klasy $\mathcal{L}_2$, na podstawie Lematu powyższy przekrój także byłby językiem bezkontekstowym.

3 Rozstrzygalność w klasie $\mathcal{L}_2$

Z dyskusji przy końcu podrozdziału IV.1 wiemy już, że problem czy dane słowo jest czy też nie jest elementem języka opisanego przez gramatykę bezkontekstową jest rozstrzygalny. Wiemy także, że istnieją ogólne, obliczeniowo wydajne metody (algorytm CYK) rozwiązujące ten problem, a w przypadku języków deterministycznych są to wręcz metody o liniowej złożoności względem długości analizowanego słowa. Opiszemy pokrótce, które z typowych problemów posiadają efektywne rozwiązania dla języków bezkontekstowych.

LEMAT 4.7 *Dla danej gramatyki bezkontekstowej pytanie czy generowany przez nią język jest pusty oraz pytanie czy jest on nieskończony są rozstrzygalne.*

DOWÓD. By przekonać się o tym, czy $L(G) = \emptyset$ czy też nie, poddajmy gramatykę operacjom eliminującym puste produkcje, a następnie wyznaczmy wszystkie symbole bezużyteczne (por. Rozdział IV.1.3). Jeśli symbol startowy okaże się bezużyteczny, język $L(G)$ jest pusty. By stwierdzić czy jest on nieskończony, podobnie rozpoczynamy od uproszczenia gramatyki przez eliminację pustych, jednostkowych i bezużytecznych produkcji, a następnie tworzymy graf zależności nieterminali: X łączy się skierowaną krawędzią z Y jeśli G zawiera produkcję postaci $X \to \alpha Y \beta$. Łatwo sprawdzić, że L jest nieskończony wtedy i tylko wtedy, gdy graf zależności zawiera cykle. Z kolei sprawdzenie czy dany graf skierowany zawiera cykle jest standardowym efektywnie wykonalnym testem w teorii grafów. □

Bez szczegółowych dowodów podajemy niżej listę niektórych problemów nierozstrzygalnych w klasie $\mathcal{L}_2$. Niech G i H oznaczają zadane gramatyki bezkontekstowe.

1. Rozstrzygnąć czy $L(G) = L(H)$.
2. Rozstrzygnąć czy $L(G) \subseteq L(H)$.
3. Zbadać czy $L(G) \cap L(H) = \emptyset$.
4. Zbadać czy $L(G) = A^*$.
5. Rozstrzygnąć czy $L(G)$ jest językiem regularnym.
6. Rozstrzygnąć czy gramatyka G jest niejednoznaczna.
7. Zbadać czy dany język bezkontekstowy L jest wewnętrznie niejednoznaczny.

Zauważmy, że badanie czy $L(G) \oplus L(H) = \emptyset$ jako równoważnik zagadnienia 1, skuteczne w przypadku języków regularnych, nie może być tym razem zastosowane, gdyż ani przekrój, ani też dopełnienie nie są jak już wiemy wykonalne w klasie $\mathcal{L}_2$. Z tego samego powodu nieskuteczna byłaby próba redukcji problemu 4 do pytania czy $L^c(G) = \emptyset$.

Zwróćmy także uwagę na pewien subtelny szczegół. Nie należy utożsamiać pytania 5 z problemem badania czy *konkretny* zadany język jest regularny. Mamy wszak do tego wygodne narzędzia — twierdzenie Myhilla i lemat o pompowaniu. W problemie 5 chodzi o coś zupełnie innego: chodzi o to, czy istnieje *jeden ogólny algorytm*, który czytając jako dane opis dowolnej gramatyki bezkontekstowej odpowiada na pytanie czy generowany przez nią język jest regularny. Zastosowanie twierdzenia Myhilla takim algorytmem nie jest, bowiem na ogół używa odmiennych metod, a czasem całkiem niestandardowych sztuczek, zależnie od tego jaki język analizujemy przy jego pomocy. Nierozstrzygalny jest także nieco prostszy problem, będący wariantem pytania 1: dla danych gramatyk G bezkontekstowej i H regularnej rozstrzygnąć czy $L(G) = L(H)$.

Problem 1 jest rozstrzygalny dla gramatyk $LL(k)$, natomiast nie wiadomo nic o rozstrzygalności tego zagadnienia w całej klasie języków deterministycznych. Wykazano natomiast natomiast rozstrzygalność problemu 4 w tej klasie.

Zainteresowany ta tematyką czytelnik znajdzie więcej informacji w książkach [1] i [4], por. także ostatni rozdział niniejszego skryptu.

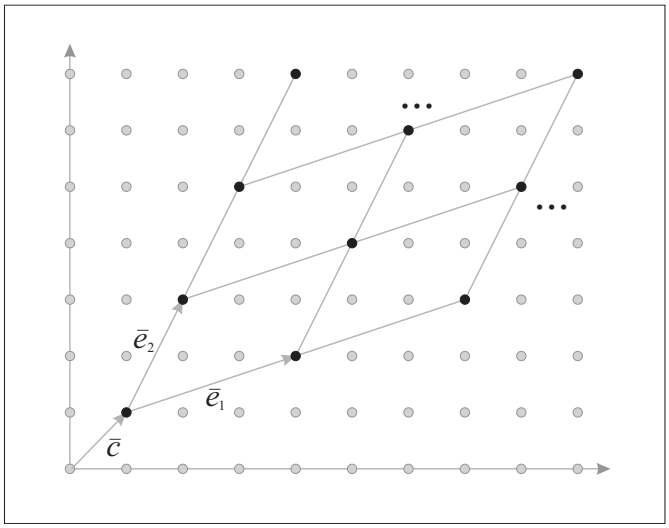
4 Twierdzenie Parikha i inne charakteryzacje języków klasy $\mathcal{L}_2$

Twierdzenia Parikha jest kolejnym rezultatem charakteryzującym języki bezkontekstowe, tym razem przez liczby wystąpień poszczególnych liter terminalnych w ich słowach. Zdefiniujmy najpierw dwa niezbędne pojęcia.

DEFINICJA 4.13 *Dla danego alfabetu $A = \{a_1, a_2, \ldots, a_n\}$ odwzorowaniem Parikha nazywamy funkcję $\Psi : A^* \to \mathbb{N}^n$ określoną następująco:*

$$\Psi(\omega) \;=\; \Big(|\omega|_{a_1}, |\omega|_{a_2}, \ldots, |\omega|_{a_n}\Big)\,. \tag{4.21}$$

Tak więc $\Psi(\omega)$ jest wektorem całkowitoliczbowym, którego składowe określają liczby wystąpień symboli a_i w ω. Zauważmy przy tym, że kolejność numeracji liter alfabetu a_i nie jest tu istotna, ważne jedynie by była na początku ustalona, aby wiadomo było do której litery odnosi się i-ta składowa $\Psi(\omega)$. Np. jeśli $A = \{a, b, c\}$, wówczas $\Psi(aacabb) = (3, 2, 1)$ itp.



Rys. 4.9: Przykład zbioru liniowego w $\mathbb{N}^2$.

DEFINICJA 4.14 *Podzbiór $K \subseteq \mathbb{N}^n$ nazywamy liniowym, jeśli istnieją wektory $\overline{c}$ i $\overline{e}_1, \ldots, \overline{e}_m \in \mathbb{N}^n$, $m \leqslant n$, dla których*

$$K \;=\; \left\{\overline{c} \;+\; l_1\overline{e}_1 + l_2\overline{e}_2 + \cdots + l_m\overline{e}_m \;:\; \; l_1, l_2, \ldots l_m \in \mathbb{N}\right\}.$$

Zbór K nazywamy semiliniowym, jeśli jest on sumą mnogościową skończonej liczby zbiorów liniowych.

Zbiór liniowy przypomina więc rozmaitość liniową znaną z geometrii afinicznej, z tą różnicą, że w przypadku rozmaitości współczynniki l_i powinny być elementami ciała liczbowego, którym $\mathbb{N}$ nie jest. Punkty zbioru liniowego tworzą regularną siatkę w zbiorze $\mathbb{N}^n$, por. Rys. 4.9. Zbiór liniowy można uważać za wielowymiarowe uogólnienie ciągu arytmetycznego, $a_n = a_0 + nr$, gdzie $\overline{c}$ pełni rolę wyrazu początkowego a_0, a wektory $\overline{e}_i$ są wielowymiarowymi przyrostami odpowiadającymi r. Mówiąc obrazowo, zbiory liniowe i semiliniowe w $\mathbb{N}^n$ to zbiory o bardzo prostej, regularnej strukturze.

Możemy teraz sformułować twierdzenie Parikha [8, 11]. Dowód pomijamy.

TWIERDZENIE 4.5 *Jeśli język L jest bezkontekstowy, wówczas jego obrazem w odwzorowaniu Parikha $\Psi(L)$ jest zbiór semiliniowy.*

Zauważmy, że bardzo różne języki mogą mieć identyczną reprezentację w odwzorowaniu Parikha jako podzbiory $\mathbb{N}^n$. Na przykład $L = \{a^n b^n : \; n \geqslant 0\}$ i język poprawnie zagnieżdżonych nawiasów przekształcane są przez Ψ w zbiór $\{(n, n) : n \in \mathbb{N}\}$. Ma zatem sens następujące określenie: mówimy, że języki L i L' są *literowo* albo też *permutacyjnie równoważne* jeśli $\Psi(L) = \Psi(L')$.

Podamy teraz prosty do udowodnienia fakt.

LEMAT 4.8 *Każdy zbiór semiliniowy w $\mathbb{N}^n$ jest obrazem pewnego języka regularnego nad n-literowym alfabetem.*

DOWÓD. Udowodnimy lemat dla $n = 2$, uogólnienie na więcej wymiarów będzie oczywiste. Niech K będzie zbiorem liniowym, dla którego $\overline{c} = (c_1, c_2)$ oraz $\overline{e}_1 = (p, q)$ i $\overline{e}_2 = (r, s)$. Tworzymy na tej podstawie wyrażenie regularne

$$\rho = a^{c_1} b^{c_2} (a^p b^q)^* (a^r b^s)^* .$$

Łatwo sprawdzić, że $\Psi(L(\rho)) = K$. Jeśli K jest semiliniowy, $K = K_1 \cup \ldots \cup K_m$, utwórzmy podobnie wyrażenia regularne ρ_i dla K_i a następnie połóżmy $\rho = \rho_1 + \cdots + \rho_m$. $L(\rho)$ jest poszukiwanym językiem regularnym. □

Prowadzi to nas do wniosku, który czasami podawany jest jako właściwe sformułowanie twierdzenia Parikha.

WNIOSEK 4.1: *Każdy język bezkontekstowy jest permutacyjnie równoważny z pewnym językiem regularnym.*

Różnica między językami bezkontekstowymi a regularnymi zasadza się więc w uporządkowaniu liter, nie zaś w arytmetycznych relacjach między ich liczebnościami w różnych słowach. W szczególnym wypadku języków nad jednoliterowym alfabetem uporządkowanie liter przestaje mieć znaczenie i otrzymujemy

WNIOSEK 4.2: *Każdy język bezkontekstowy nad jednoliterowym alfabetem jest regularny.*

Zapowiedzią tego wniosku był rozważany przez nas Przykład 4.16, $L = \{a^{n^2} : n \geqslant 0\}$. Jeśli jednoliterowy język zostanie rozpoznany jako nieregularny, wiadomo od razu, że nie może on być bezkontekstowy.

Zauważmy jednak, że istnieją bardziej skomplikowane języki, których obrazy Parikha mogą nadal mieć prostą strukturę zbioru semiliniowego. Np. $L = \{a^n b^n c^n : n \geqslant 0\}$ mimo, że jak już wiemy nie jest bezkontekstowy, posiada bardzo prosty liniowy obraz $\{(n, n, n) : n \geqslant 0\}$. Zatem nieprawdziwe jest stwierdzenie odwrotne do Twierdzenia 4.5, że jeśli obraz $\Psi(L)$ jest semiliniowy, to L musi być bezkontekstowy.

Wspomnimy na koniec o jeszcze jednym ważnym twierdzeniu formułującym algebraiczną charakteryzację języków bezkontekstowych. Przedstawia ono język poprawnie zagnieżdżonych nawiasów jako swego rodzaju prototyp wszystkich języków bezkontekstowych. W istocie potrzebny jest nieco bardziej ogólny prototyp, by poradzić sobie z językami nad bogatszymi niż dwuliterowe alfabetami.

DEFINICJA 4.15 *Rozważmy alfabet zawierający parzystą liczbę powiązanych w pary liter $A_n = \{l_1, r_1, l_2, r_2, \ldots, l_n, r_n\}$. Językiem Dycka rzędu n, D_n, nazywamy język bezkontekstowy generowany przez gramatykę o produkcjach*

$$S \rightarrow l_1 S r_1 \,|\, l_2 S r_2 \,|\, \ldots \,|\, l_n S r_n \,|\, SS \,|\, \lambda .$$

Język Dycka jest niczym innym jak językiem poprawnie zagnieżdżonych nawiasów n różnych typów l_1, r_1 l_2, r_2 itd., tak jak np. w napisie $[()()]\{\}$. Oto wspomniane twierdzenie.

TWIERDZENIE 4.6 *(Chomsky-Schützenberger) Każdy język bezkontekstowy L jest homomorficznym obrazem przekroju pewnego języka regularnego R z językiem Dycka,*

$$L \;=\; h(R \cap D_n)\,.$$

Można więc powiedzieć, że pod względem porządku liter każdy język bezkontekstowy L naśladuje strukturę poprawnie zagnieżdżonych nawiasów, zaś przekrój z językiem R pozwala skorygować liczebność poszczególnych liter w słowach zgodnie z twierdzeniem Parikha, natomiast homomorfizm odpowiedzialny jest za ewentualną zmianę alfabetu.

IV.4 Zadania do Rozdziału IV

Zadanie 1

Znaleźć gramatyki bezkontekstowe dla następujących języków:

a) $L = \{a^m b^n :\ m \leqslant n+3\}$

b) $L = \{a^m b^n :\ m \neq n-1\}$

c) $L = \{a^m b^n :\ m \neq 2n\}$

d) $L = \{a^m b^n :\ 2m \leqslant n \leqslant 3m\}$

e) $L = \{\omega \in \{a,b\}^* :\ |\omega|_a \neq |\omega|_b\}$

f) $L = \{a^k b^m c^n :\ k = m \quad \text{lub} \quad m \leqslant n\}$

g) $L = \{a^k b^m c^n :\ k = m \quad \text{lub} \quad m \neq n\}$

h) $L = \{a^{m+n} b^m c^n :\ m, n \geqslant 0\}$

i) $L = \{a^k b^m c^{|k-m|} :\ k, m \geqslant 0\}$

j) $L = \{a^k b^m c^n :\ n \neq k+m\}$

k) $L = \{\omega \in \{a,b,c\}^* :\ |\omega|_a + |\omega|_b \neq |\omega|_c\}$

l) $L = \{\omega \in \{a,b,c\}^* :\ |\omega| = 3|\omega|_a\}$

Zadanie 2

Podać gramatykę bezkontekstową opisującą język złożony z poprawnych wyrażeń logicznych wzgl. zmiennych p, q, r z operacjami $\wedge$, $\vee$, $\neg$ oraz z nawiasami ( ).

Zadanie 3

Znaleźć gramatykę bezkontekstową generującą język poprawnych wyrażeń regularnych nad alfabetem $\{a, b\}$.

Zadanie 4

Pokazać, że dopełnienie języka $L = \{a^n b^n :\ n \geqslant 0\}$ jest językiem bezkontekstowym.

Zadanie 5

Pokazać, że dopełnienie języka $L = \{\omega\overleftarrow{\omega} : \ \omega \in \{a,b\}^*\}$ jest językiem bezkontekstowym.

Zadanie 6

Pokazać, że gramatyka z produkcjami

$$S \to AB \,|\, aaB\,, \quad A \to a \,|\, Aa\,, \quad B \to b$$

jest niejednoznaczna.

Zadanie 7

Skonstruować jednoznaczną gramatykę równoważną tej z poprzedniego zadania.

Zadanie 8

Znaleźć jednoznaczną gramatykę równoważną gramatyce otrzymanej w Zadaniu 3.

Zadanie 9

Pokazać, że każda s-gramatyka jest jednoznaczna.

Zadanie 10

Pokazać, że język poprawnie zagnieżdżonych nawiasów nie jest wewnętrznie niejednoznaczny.

Zadanie 11

Uprościć gramatykę

$$S \to AB|CAb\,, \quad A \to aaA|\lambda\,, \quad B \to Bb|\lambda\,, \quad C \to CS|CaB$$

usuwając z niej wszystkie λ-produkcje, produkcje jednostkowe i produkcje bezużyteczne.

Zadanie 12

$$\begin{aligned} S &\to aScc \,|\, SZc \,|\, X \\ Z &\to YcZ \,|\, ZY \\ Y &\to \lambda \,|\, bZ \\ X &\to bXc \,|\, Y \end{aligned}$$

a) Uprościć gramatykę usuwając z niej kolejno λ-produkcje, produkcje jednostkowe i produkcje bezużyteczne.

b) Przekształcić uproszczoną gramatykę do postaci normalnej Chomsky'ego.

Zadanie 13

Przekształcić następującą gramatykę do postaci Chomsky'ego:

$$S \to aaB\,, \quad A \to bBb \,|\, \lambda\,, \quad B \to Aa\,.$$

Zadanie 14

Znaleźć postać Chomsky'ego dla gramatyk otrzymanych w Zadaniach 1 a), b), d), h) oraz l).

Zadanie 15

Korzystając z gramatyki uzyskanej w poprzednim zadaniu zasymulować działanie algorytmu CYK dla języka $L = \{a^m b^n : \ m \neq n-1\}$ i słów *aaabbb* oraz *aabbb*.

Zadanie 16

Skonstruować niedeterministyczne automaty ze stosem akceptujące następujące języki:

a) $L = \{a^n b^{2n} : \ n \geqslant 0\}$

b) $L = \{ab(ab)^n b(ba)^n : \ n \geqslant 0\}$

c) $L = \{a^m b^{m+n} c^n : \ m \geqslant 0,\, n \geqslant 1\}$

d) $L = \left\{\omega \in \{a,b,c\}^* : \ \ |\omega|_a + 2|\omega|_b = |\omega|_c\right\}$

e) $L = \left\{\omega \in \{a,b\}^* : \ \ 2|\omega|_a \leqslant |\omega|_b \leqslant 3|\omega|_a\right\}$

Zadanie 17

Pokazać, że język z Zadania 16 a) jest językiem deterministycznym.

Zadanie 18

Pokazać, że język $L = \{\omega \in \{a,b\}^* : \ \ |\omega|_a \neq |\omega|_b\}$ jest deterministyczny.

Zadanie 19

Pokazać, że $L = \{\alpha c \overleftarrow{\alpha} : \ \alpha \in \{a,b\}^*\}$ jest deterministyczny.

Zadanie 20

Uzasadnić, że języki regularne stanowią podklasę $\mathcal{LL}(1)$.

Zadanie 21

Znaleźć gramatyki typu LL dla następujących języków bezkontekstowych:

a) $L = \{a^m b^n c^{m+n} : \ m, n \geqslant 0\}$

b) $L = \{\omega \in \{a,b\}^* : \ \ |\omega|_a < |\omega|_b\}$

c) $L = \{\omega \in \{a,b,c\}^* : \ \ |\omega|_a + |\omega|_b \neq |\omega|_c\}$

Zadanie 22

Stosując lemat o pompowaniu dla języków bezkontekstowych pokazać, że następujące języki nie są klasy $\mathcal{L}_2$:

a) $L = \{a^k b^l c^{kl},\ \ k, l \geqslant 1\}$

b) $L = \{a^m b^m c^n :\ \ m \neq n\}$

c) $L = \{\omega\, \overleftarrow{\omega}\, \omega :\ \ \omega \in \{a, b\}^*\}$

d) $L = \{a^n b^{2^n} :\ \ n \geqslant 0\}$

e) $L = \{\omega \in \{a, b, c\}^* :\ \ |\omega|_a < |\omega|_b < |\omega|_c\}$

f) $L = \{a^p :\ \ p$ jest liczbą pierwszą$\}$

Zadanie 23

Stosując lemat o pompowaniu dla języków liniowych pokazać, że następujące języki nie są tej klasy:

a) $L = \{a^m b^m a^n b^n :\ \ m, n \geqslant 0\}$

b) $L = \{\omega \in \{a, b\}^* :\ \ |\omega|_a \leqslant |\omega|_b\}$

c) $L = \{\omega \in \{a, b, c\}^* :\ \ |\omega|_a + |\omega|_b = |\omega|_c\}$

Zadanie 24

Pokazać, że klasa $\mathcal{L}_{\text{Lin}}$ jest zamknięta ze względu na sumę mnogościową, lecz nie jest zamknięta ze względu na konkatenację.

Zadanie 25

Pokazać, że klasa języków deterministycznych nie jest zamknięta ze względu na sumę mnogościową.

Zadanie 26

Pokazać, że jeśli język L_1 jest liniowy, a L_2 regularny, wówczas $L_1 L_2$ jest liniowy.

Zadanie 27

Podać przykład języka bezkontekstowego, którego dopełnienie nie jest tej klasy.

Zadanie 28

Zaproponować algorytm, który dla danej gramatyki bezkontekstowej G rozstrzyga czy $\lambda \in L(G)$.

Zadanie 29

Pokazać, że następujący problem jest rozstrzygalny: mając daną gramatykę bezkontekstową G oraz liczbę $n > 0$, rozstrzygnąć czy $L(G)$ zawiera słowa ω o długości nie przekraczającej n.

Zadanie 30

Pokazać, że istnieje algorytm, który dla danych języków bezkontekstowego i regularnego rozstrzyga czy ich przekrój jest niepusty.

Zadanie 31

Znaleźć języki regularne permutacyjnie równoważne językom a), b), d), h), i) oraz l) z Zadania 1.

Zadanie 32

Znaleźć odpowiednie języki R, D_n i homomorfizm h, o których mowa w Twierdzeniu 4.6 dla

a) $L = \{a^n b^n : \ n \geqslant 0\}$

b) $L = \{a^n b^{2n} : \ n \geqslant 0\}$

c) $L = \{\omega \in \{a, b\}^* : \ |\omega|_a = |\omega|_b\}$

d) języków z Zadania 1 pkt. a), b), c) oraz h).

Rozdział V

Gramatyki i języki kontekstowe

W Rozdziale II podaliśmy ogólną definicję gramatyki i definiowanego przez nią języka, por. Definicje 2.4 i 2.5. Dalej w Rozdziale IV, por. str. 85, argumentowaliśmy, że badanie, czy dane słowo jest elementem generowanego przez gramatykę języka, oparte na systematycznej analizie drzewa wyprowadzeń może być skuteczne jedynie wtedy, gdy wyprowadzenia są nieskracające. Przypomnijmy pokrótce ten argument, gdyż będzie on bardzo istotny w dalszej dyskusji.

Na podstawie opisu gramatyki G staramy się rozstrzygnąć dla dowolnego słowa α czy $\alpha \in L(G)$. Bez dodatkowych założeń o postaci gramatyki musimy zdać się na ogólną metodę systematycznej konstrukcji drzewa wszystkich możliwych wyprowadzeń w G, w którego k-tym piętrze znajdą się wszystkie formy zdaniowe osiągalne w k krokach z symbolu startowego S, por. Rys. 4.1. Liczba elementów k-tego piętra drzewa ograniczona jest przez p^k, gdzie p jest liczbą produkcji w G. Jeśli $\alpha \in L(G)$, słowo to na pewno pojawi się w pewnym momencie w drzewie wyprowadzeń, jeśli zaś $\alpha \notin L(G)$, wówczas w ogólnym przypadku systematyczny przegląd nie jest w stanie potwierdzić tego faktu w jakiejkolwiek skończonej liczbie kroków. Dzieje się tak, gdy dopuścimy wyprowadzenia skracające formy zdaniowe, a więc w szczególności dla gramatyk z produkcjami $X \to \lambda$. Jeśli przeciwnie długość form zdaniowych wzdłuż ścieżek w drzewie wyprowadzeń rośnie monotonicznie, wówczas znając długość α, możemy przerwać przegląd po osiągnięciu poziomu w drzewie, poniżej którego znajdują się już w nim wszystkie formy zdaniowe o tej długości i tym samym rozstrzygnąć czy $\alpha \in L(G)$.

W przypadku gramatyk bezkontekstowych obecność λ-produkcji naruszałaby własność nieskracania. Jak już wiemy, dla tych gramatyk istnieje ogólna metoda eliminowania takich produkcji bez naruszania generowanego języka. Inaczej ma się sprawa dla bardziej ogólnych gramatyk: w ich przypadku własność nieskracania musi być zagwarantowana w definicji, w przeciwnym bowiem razie gramatyczny opis języka byłby nieefektywny.

Omawiane w tym rozdziale gramatyki kontekstowe definiowane są właśnie w ten sposób: wymagamy aby wszystkie ich produkcje były nieskracające. Opiszemy także własności generowanych przez nie języków kontekstowych i zbadamy czy wyczerpują one kategorię wszystkich języków, dla których problem należenia słowa jest efektywnie rozstrzygalny.

V.1 Gramatyki kontekstowe i ich postać normalna

Rozpocznijmy od definicji gramatyk kontekstowych i języków przez nie generowanych.

DEFINICJA 5.1 *$G = (A, V, S, \Pi)$ nazywamy gramatyką kontekstową lub równoważnie gramatyką typu 1, jeśli wszystkie jej produkcje spełniają warunek nieskracania:*

$$\gamma \to \eta \in \Pi \qquad \textit{wtedy i tylko wtedy, gdy} \qquad |\gamma| \leqslant |\eta| \,. \tag{5.1}$$

W zbiorze Π wyjątkowo wystąpić może produkcja postaci $S \to \lambda$, pod warunkiem jednak, że wówczas symbol startowy S nie pojawia się w prawych stronach innych produkcji.

Języki generowane przez gramatyki typu 1 nazywamy językami kontekstowymi. Tworzą one klasę oznaczoną w hierarchii Chomsky'ego symbolem $\mathcal{L}_1$.

Przypomnijmy, że zgodnie z Definicją 2.4 lewa strona każdej produkcji musi zawierać co najmniej jeden symbol nieterminalny.

W Przykładzie 2.6 opisaliśmy gramatykę dla języka $L = \{a^n b^n a^n : \; n \geqslant 0\}$. Nie spełnia ona jednak powyższej definicji, ponieważ zawiera produkcję postaci $AX \to a$. Oto równoważna gramatyka, tym razem kontekstowa.

PRZYKŁAD 5.1 Gramatyka G z produkcjami

$$\begin{array}{rcl} S & \to & aXba \,|\, aba \,|\, \lambda \\ aX & \to & aaXbX \,|\, aabX \\ bXb & \to & bbX \\ bXa & \to & baa \end{array}$$

generuje język L jak wyżej. Istotnie, mamy np.

$$\begin{aligned} S \;\Rightarrow\; aXba \;\Rightarrow\; aaXbXba \;\Rightarrow\; aaabXbXba \;\Rightarrow\; aaabXbbXa \\ \Rightarrow\; aaabXbbaa \;\Rightarrow\; aaabbXbaa \;\Rightarrow\; aaabbbXaa \;\Rightarrow\; aaabbbaaa \end{aligned}$$

Przez prostą modyfikację można zamienić G w gramatykę dla pokrewnego języka $L = \{a^n b^n c^n : \; n \geqslant 0\}$.

PRZYKŁAD 5.2 Niech z kolei G zawiera następujące produkcje

$$\begin{array}{rcl} S & \to & a \,|\, aa \,|\, AXXA \\ AX & \to & AXYX \\ YX & \to & XXY \\ YA & \to & XA \\ A & \to & a \\ aX & \to & aa \end{array}$$

Gramatyka ta generuje język $L = \{a^{2^n} : \; n \geqslant 0\}$. Wyprowadzenia słów a, aa i $aaaa$ są oczywiste. Przeanalizujmy wyprowadzenie rozpoczynające się od $S \Rightarrow AXXA \Rightarrow AXYXXA$. Zauważmy, że oprócz zamiany symboli A i X na terminal a

kończącej wyprowadzenie, użycie w drugim kroku produkcji $AX \to AXYX$ generującej zmienną Y jest jedyną alternatywą dostępną na tym etapie. Rola zmiennej Y polega na tym, że poruszając się w prawo przez kolejne symbole X podwaja ich liczbę za sprawą produkcji $YX \to XXY$. Ponadto rozpoczynająca ten cykl produkcja $AX \to AXYX$ "podwaja" także symbol A do postaci AX. By pozbyć się zmiennej Y, której nie można zamienić bezpośrednio w terminal, należy przesunąć ją aż do końcowego symbolu A. Po osiągnięciu tego punktu symbol Y znika "podwajając" przy tym końcowe A do XA. Tak więc jeden przebieg, przesuwający zmienną Y od lewej do prawej strony, powoduje podwojenie długości napisu $AX \ldots XA$. Zamiana A i X na literę a kończy wyprowadzenie kolejnego słowa a^{2^n}, natomiast kontynuacja wyprowadzenia przez kolejne wygenerowanie Y przy początkowym A rozpoczyna podobny proces tworzenia słowa $a^{2^{n+1}}$.

Przykład 5.3 W kolejnym przykładzie omówimy konstrukcję gramatyki dla języka $L = \{\omega\omega :\ \omega \in \{a,b\}^*\}$. Idea tej konstrukcji polega na wstępnym wygenerowaniu słowa $\omega\overleftarrow{\omega}$ i następującym po nim odwróceniu $\overleftarrow{\omega}$ do ω. Oczywiście słowo $\overleftarrow{\omega}$ musi być zapisane na tym etapie przy pomocy nieterminali, powiedzmy A, B, by była możliwa jego dalsza modyfikacja. Spróbujmy opisać więc proces odwracania kolejności symboli w dowolnym słowie $\Theta \in \{A,B\}^*$ przy pomocy kontekstowych produkcji. Oto jeden z wielu możliwych scenariuszy. Ułatwimy sobie zadanie przyjmując, że przed pierwszą literą Θ stoi dodatkowy symbol P (zadbamy później o to, by w pierwszym etapie taki symbol się w tym miejscu pojawił). Rola P jako znacznika aktualnego początku przekształcanego słowa Θ polega na wywołaniu kontekstowej zamiany sąsiadującej z nim litery A lub B odpowiednio w symbol X lub Y, który następnie będzie przesuwany w prawo, a gdy znajdzie się na końcu słowa Θ, zamieni się w odpowiadający mu symbol terminalny. Produkcje realizujące ten proces to:

PA	$\to$	PX,	PB	$\to$	PY	zamiana początkowego A w X lub B w Y
XA	$\to$	AX,	XB	$\to$	BX	przesuwanie X w prawo przez A lub B
YA	$\to$	AY,	YB	$\to$	BY	przesuwanie Y w prawo przez A lub B
X	$\to$	a,	Y	$\to$	b	zamiana X i Y w terminale

Przekonajmy się jak działa ten algorytm:

$$PABB \Rightarrow PXBB \Rightarrow PBXB \Rightarrow PBBX \Rightarrow PBBa \Rightarrow PYBa \Rightarrow PBYa \Rightarrow PBba \Rightarrow PYba \Rightarrow Pbba.$$

Zauważmy, że zbyt wczesna zamiana symbolu X lub Y na terminal (to jest przed osiągnięciem przez niego końca słowa Θ) spowoduje zablokowanie możliwości dalszego przekształcania znajdujących się na prawo od niego liter A i B. W tym wypadku wyprowadzenie zakończy się na napisie zawierającym nieterminale. Z drugiej strony zamiana X i Y na końcu Θ ustawia blokadę dla przesunięć w prawo kolejnych symboli X i Y i początkowa kolejność A i B w Θ ulega stopniowemu odwróceniu. Należy jeszcze pozbyć się symbolu P, jednak chcąc zachować kontekstowość gramatyki nie możemy w tym celu użyć nielegalnej produkcji $P \to \lambda$. Zamiast tego, zadbamy by P było nieterminalem w wyprowadzeniu początkowego słowa ω.

Uzupełnijmy więc naszą gramatykę o produkcje realizujące pierwszą fazę wyprowadzenia:

$$S \to aSA \,|\, bSB \,|\, PA \,|\, QB$$

generują "wzorce" słów $\omega\overleftarrow{\omega}$, przy czym ostatnia litera w ω zakodowana jako P dla a lub Q dla b pełni jednocześnie rolę znacznika początku części $\overleftarrow{\omega}$ dla potrzeb fazy drugiej, odwracającej $\overleftarrow{\omega}$. Dla kompletności gramatyki pozostaje dodać produkcje usuwające P i Q:

$$P \to a\,, \qquad Q \to b\,.$$

Podobnie jak poprzednio, przedwczesne usunięcie P lub Q uniemożliwia eliminację symboli A i B na prawo od niego.

W końcu by zaliczyć do języka L także słowo puste, stosujemy standardowe rozszerzenie G o nowy symbol startowy S' i produkcje

$$S' \to S \mid \lambda\,.$$

Opiszemy obecnie szczególnie dogodną postać gramatyki kontekstowej.

TWIERDZENIE 5.1 *Dowolną gramatykę $G = (A, V, S, \Pi)$ typu 1 można przekształcić do równoważnej postaci, w której wszystkie produkcje mają formę*

$$\alpha X \beta \;\to\; \alpha\gamma\beta\,, \tag{5.2}$$

gdzie $X \in V$, $\alpha, \beta, \gamma \in (A \cup V)^*$ oraz $\gamma \neq \lambda$, z możliwym wyjątkiem produkcji $S \to \lambda$ pod warunkiem jednak, że S nie występuje wówczas w prawych stronach innych produkcji.

Opisywana tu postać gramatyki tłumaczy pochodzenie terminu "kontekstowa". Produkcja (5.2) pozwala zastąpić symbol X napisem γ jedynie wówczas, gdy występuje on w otoczeniu, czyli w *kontekście* napisów α i β. Za sprawą założenia, że $\gamma \neq \lambda$ produkcje tego typu są oczywiście nieskracające. Gdy jednocześnie α i β są puste, produkcja redukuje się do postaci bezkontekstowej. Tak więc każda gramatyka bezkontekstowa jest szczególnym przypadkiem gramatyki kontekstowej, a więc $\mathcal{L}_2 \subseteq \mathcal{L}_1$. Zauważmy ponadto, że na podstawie Przykładów 4.15 i 4.15 oraz omówionych przed chwilą 5.1 i 5.3 inkluzja ta jest właściwa.

SZKIC DOWODU. Niech G będzie gramatyką o nieskracających produkcjach. Załóżmy, że terminale występują jedynie w produkcjach postaci $A \to a$ (zakładamy więc, że każdy terminal posiada reprezentujący go symbol nieterminalny). Produkcje mają zatem postać

$$X_1 X_2 \ldots X_m \;\to\; Y_1 Y_2 \ldots Y_n\,, \qquad 2 \leqslant m \leqslant n\,. \tag{5.3}$$

Każdą z nich można zamienić na równoważny ciąg produkcji postaci

$$\begin{aligned}
X_1 X_2 \ldots X_m &\to Z_1 X_2 \ldots X_m \\
Z_1 X_2 X_3 \ldots X_m &\to Z_1 Z_2 X_3 \ldots X_m \\
&\vdots \\
Z_1 \ldots Z_{m-1} X_m &\to Z_1 \ldots Z_{m-1} Z_m Y_{m+1} \ldots Y_n \\
Z_1 Z_2 \ldots Z_m Y_{m+1} \ldots Y_n &\to Y_1 Z_2 \ldots Z_m Y_{m+1} \ldots Y_n \\
&\vdots \\
Y_1 \ldots Y_{m-1} Z_m Y_{m+1} \ldots Y_n &\to Y_1 \ldots Y_{m-1} Y_m Y_{m+1} \ldots Y_n
\end{aligned} \tag{5.4}$$

gdzie Z_j, $j = 1, \ldots m$ są nowymi symbolami nieterminalnymi, unikalnymi dla każdej przetwarzanej produkcji (5.3). Zauważmy, że każda z produkcji (5.4) jest już postaci (5.2). Użycie w wyprowadzeniu produkcji (5.3) jest równoważne użyciu całego ciągu (5.4), przy czym elementów tego ciągu nie można wykorzystać inaczej niż wspólnie z pozostałymi. □

W Rozdziale IV poznaliśmy postać normalną Chomsky'ego gramatyk bezkontekstowych, która używana jest między innymi w algorytmie CYK. Gramatyki kontekstowe również posiadają specjalne, proste formy. Najbardziej istotne jest to, że istnieje uniwersalny algorytm transformujący dowolną gramatykę kontekstową do zadanej postaci normalnej. Oto dwie takie postaci.

DEFINICJA 5.2 *Mówimy, że gramatyka kontekstowa dana jest w postaci normalnej Kurody, jeżeli każda z jej produkcji przyjmuje jedną z trzech dopuszczalnych postaci:*

$$A \to a\,, \qquad A \to BC\,, \qquad AB \to CD\,.$$

Ponadto gramatyka jest w jednostronnej postaci normalnej Kurody, jeśli trzeci z podanych wyżej wzorców produkcji ma postać

$$AB \to AC\,.$$

Zachodzi następujące twierdzenie:

TWIERDZENIE 5.2 *Każdą gramatykę kontekstową można efektywnie przekształcić do postaci normalnej Kurody, jak również do postaci jednostronnej.*

Dowód (dla postaci normalnej Kurody). Zakładamy standardowo, że symbole terminalne obecne są jedynie w produkcjach typu $A \to a$. Drugi rodzaj produkcji to produkcje bezkontekstowe w postaci Chomsky'ego. Jeśli gramatyka zawiera produkcje bezkontekstowe, można je przekształcić znaną z Rozdziału IV metodą do tej właśnie postaci (gramatyka kontekstowa jest z definicji λ-wolna, a ewentualne jednostkowe produkcje można usunąć z niej dokładnie tak, jak opisaliśmy to w Rozdziale IV). Każdą produkcję postaci (5.3) dla $2 \leqslant m < n$ (gdy $m = n = 2$, produkcja ta ma już postać $AB \to CD$) przekształcamy w równoważny ciąg produkcji

$$\begin{aligned}
X_1X_2 &\to Y_1Z_2\\
Z_2X_3 &\to Y_2Z_3\\
&\vdots\\
Z_{m-1}X_m &\to Y_{m-1}Z_m\\
Z_m &\to Y_mZ_{m+1}\\
Z_{m+1} &\to Y_{m+1}Z_{m+2}\\
&\vdots\\
Z_{n-1} &\to Y_{n-1}Y_n
\end{aligned}$$

gdzie Z_j, $j = 2, \ldots, n-1$ są nowymi nieterminalami, unikalnymi dla każdej przekształcanej produkcji (5.3). □

Przytoczymy na koniec dwa rezultaty, których ważność polega na tym, że pokazują explicite w czym zasadza się różnica między gramatykami kontekstowymi a ogólną, nieograniczoną postacią gramatyk typu 0, zdefiniowaną przez nas w Rozdziale II, Definicja 2.4.

TWIERDZENIE 5.3 *Każdą gramatykę typu 0 można doprowadzić do równoważnej postaci, w której oprócz produkcji pustych $A \to \lambda$ wszystkie reguły mają formę kontekstową (5.2).*

WNIOSEK. *Każdą gramatykę typu 0 można przekształcić do postaci równoważnej, zawierającej oprócz reguł $A \to \lambda$ wyłącznie produkcje w postaci normalnej Kurody.*

V.2 Własności języków kontekstowych

Wszystkie języki, które pojawiły się do tej pory w naszym skrypcie w formie przykładów są językami kontekstowymi. Okazuje się bowiem, że standardowe algorytmy obliczeniowe, których użyć można w definicji języka, prowadzą z reguły do języków klasy $\mathcal{L}_1$. Mamy tu na myśli języki takie, jak np.

$$\{a^n b^{n^2} : n \geqslant 0\}, \quad \{a^p : p \text{ jest liczbą pierwszą}\}, \quad \{a^m b^n : NWP(m,n) = 7\}$$

itp.[1] Co więcej, wiele własności składniowo-semantycznych języków naturalnych można skutecznie opisać przy pomocy gramatycznych reguł kontekstowych. Natomiast znalezienie przykładu języka spoza klasy $\mathcal{L}_1$ jest bardziej skomplikowane niż mogłoby się to wydawać, choć oczywiście istnieje bardzo wiele takich języków.

Spostrzeżenie to jest pewnym intuicyjnym wyjaśnieniem faktu, że klasa $\mathcal{L}_1$ w odróżnieniu od $\mathcal{L}_2$ jest zamknięta ze względu na większość operacji. W tym rozdziale ograniczymy się do podania najważniejszych rezultatów tej kategorii bez ich dowodzenia.

TWIERDZENIE 5.4 *Następujące operacje nie wyprowadzają poza klasę $\mathcal{L}_1$:*

(i) operacje mnogościowe: suma, przekrój i dopełnienie (a więc także różnica i różnica symetryczna)

(ii) konkatenacja i domknięcie konkatenacyjne

(iii) odbicie lustrzane

(iv) λ-wolne podstawienie (a więc także λ-wolny homomorfizm).

Podstawienie, czyli odwzorowanie przyporządkowujące każdej literze a alfabetu A pewien język L_a nad alfabetem B_a jest λ-wolne jeśli żaden z języków L_a nie zawiera napisu pustego λ. Stąd też λ-wolny homomorfizm h spełnia warunek $h(a) \neq \lambda$ dla wszystkich $a \in A$. Podstawienie, o którym mowa w punkcie (iv) to odwzorowanie $\chi : A \to \mathcal{L}_1$, które standardowo rozszerzamy na zbiór napisów A^* za pomocą formuły

[1] Zaznaczmy przy tym, że tworzenie gramatyk kontekstowych dla tego typu języków jest na ogół dość trudne. Zwykle prostszym zadaniem jest konstrukcja akceptującego automatu specjalnej klasy. O tych automatach mówić będziemy w kolejnych podrozdziałach.

$\chi(a_1 \dots a_n) = \chi(a_1) \dots \chi(a_n)$. Interesuje nas oczywiście obraz $\chi(L)$ dla $L \in \mathcal{L}_1$. Dowód zamkniętości $\mathcal{L}_1$ ze względu na takie podstawienia polega w pierwszej fazie na wyprowadzeniu słowa $\alpha \in L$ w postaci $A_1 A_2 \dots A_n$, gdzie nieterminale A_i reprezentują litery terminalne a_i (gramatyka G dla L zawiera z założenia produkcje $A_i \to a_i$). Jeśli teraz utożsamić A_i z symbolem startowym gramatyki G_{a_i} języka $\chi(a_i)$, ze słowa $A_1 A_2 \dots A_n$ wyprowadzić można wszystkie napisy w $\chi(\alpha)$. Skoro $\lambda \notin \chi(a_i)$, G_{a_i} jako gramatyka kontekstowa nie zawiera produkcji $A_i \to \lambda$. Dlatego wyprowadzenia z $A_1 \dots A_n$ są nieskracające, a zatem $\chi(L)$ jest językiem kontekstowym.

Dowolne podstawienie, jeśli nie jest λ-wolne, sprawia, że warunek nieskracania zostanie złamany. Stąd ogólny wniosek:

TWIERDZENIE 5.5 *Klasa $\mathcal{L}_1$ nie jest zamknięta ze względu na dowolne podstawienia i dowolne homomorfizmy.*

Zakończymy ten podrozdział opisem, które z zagadnień są efektywnie rozstrzygalne w klasie $\mathcal{L}_1$.

TWIERDZENIE 5.6 *Problem należenia słowa $\alpha \in L$ jest efektywnie rozstrzygalny dla języków kontekstowych, natomiast nierozstrzygalne są dla nich następujące problemy:*

(i) czy $L = \emptyset$, czy L jest skończony oraz czy L jest nieskończony

(ii) czy $L(G_1) = L(G_2)$ oraz czy $L(G_1) \subseteq L(G_2)$

(iii) czy L jest regularny lub czy jest bezkontekstowy.

Wspomnijmy jeszcze, że istnieją podklasy gramatyk kontekstowych, dla których problem (iii) jest łatwo rozstrzygalny, [10].

LEMAT 5.1 *Jeśli wszystkie produkcje w G mają postać $\alpha X \beta \to \alpha \gamma \beta$, gdzie $\alpha, \beta \in A^*$, $\gamma \in (A \cup V)^*$, wówczas istnieje gramatyka bezkontekstowa równoważna G.*

V.3 Języki rekursywne

Przyjmijmy następującą roboczą definicję, której nadamy bardziej precyzyjny wyraz w Rozdziale VI.

DEFINICJA 5.3 *Język $L \subseteq A^*$ nazywamy językiem rekursywnym, jeśli istnieje algorytm, który w skończonej liczbie kroków rozstrzyga o dowolnym słowie $\alpha \in A$ czy $\alpha \in L$, czy też $\alpha \notin L$. Klasę języków rekursywnych oznaczamy symbolem $\mathcal{L}_{\mathrm{Rec}}$.*

Oczywiście każda gramatyka nieskracająca, a więc każda kontekstowa, generuje język tej klasy. Mamy więc $\mathcal{L}_1 \subseteq \mathcal{L}_{\mathrm{Rec}}$. Pojawia się natychmiast standardowe pytanie o to, czy klasy te są równe. Oto dość zaskakujące rozstrzygniecie.

TWIERDZENIE 5.7 *Istnieje język rekursywny, który nie jest kontekstowy.*

Istnieją więc języki, dla których algorytmy rozstrzygające czy $\alpha \in L$ bazują na własnościach istotnie słabszych niż monotoniczność wyprowadzeń.

DOWÓD. Argument wykorzystuje metodę przekątniową, o której pisaliśmy w Rozdziale I, por. Przykład 1.4. Rozważmy wszystkie kontekstowe gramatyki dla dwuliterowego alfabetu $A = \{a, b\}$. Załóżmy, że w każdej z nich symbole nieterminalne oznaczone są standardowo przez X_i, $V = \{X_0, X_1, \ldots\}$, przy czym zawsze X_0 pełni rolę symbolu startowego. Każda z gramatyk jest wtedy wyczerpująco opisana przez ciąg swoich produkcji, który możemy zapisać jako pojedyncze słowo nad alfabetem $A \cup V \cup \{\rightarrow, ;\}$

$$\alpha_1 \rightarrow \beta_1;\ \alpha_2 \rightarrow \beta_2;\ \ldots\ \alpha_m \rightarrow \beta_m\,.$$

Zakodujmy to słowo jako ciąg binarny, używając następującego homomorfizmu

$$h(a) = 010,\ \ h(b) = 0110,\ \ h(\rightarrow) = 01110,\ \ h(;) = 011110,\ \ h(X_i) = 01^{i+5}0\,.$$

Tak więc każda gramatyka kontekstowa ma jednoznaczną reprezentację w postaci słowa w języku regularnym $L((011^*0)^*)$. Ponadto ta reprezentacja jest odwracalna, bowiem każdemu takiemu słowu binarnemu odpowiada nie więcej niż jedna gramatyka kontekstowa (żadna, jeśli słowo nie dekoduje się sensownie w ciąg produkcji).

Uporządkujmy zbiór $\{0,1\}^*$ leksykograficznie — każdemu ciągowi binarnemu przyporządkowana jest w ten sposób jego liczba porządkowa. To uporządkowanie indukuje numerację gramatyk kontekstowych: jeśli słowo o numerze i, ω_i, reprezentuje gramatykę kontekstową G, nadajemy jej numer i, a więc $G = G_i$. Możemy teraz zdefiniować następujący język nad $\{0,1\}$:

$$L \;=\; \Big\{\omega_i :\ \ \omega_i \text{ definiuje gramatykę kontekstową } G_i \quad \text{oraz} \quad \omega_i \notin L(G_i)\Big\}\,. \tag{5.5}$$

Przekonajmy się, że tak zdefiniowany język jest rekursywny. Weźmy bowiem dowolne $\omega_i \in \{0,1\}^*$ i sprawdźmy czy reprezentuje ono kontekstową gramatykę. Jeśli nie, wówczas $\omega_i \notin L$, zaś w przeciwnym wypadku — mając rozkodowany opis G_i — sprawdźmy czy $\omega_i \in L(G_i)$. Ten ostatni test jest oczywiście wykonalny, ponieważ język $L(G_i)$ jest z założenia kontekstowy, a więc rekursywny. Jeśli więc $\omega_i \in L(G_i))$, wówczas $\omega_i \notin L$. Natomiast gdy przeciwnie $\omega_i \notin L(G_i))$, wtedy $\omega_i \in L$. Widać więc, że dla dowolnego ω_i opisany tu algorytm weryfikuje efektywnie czy $\omega_i \in L$ czy też nie.

Okazuje się jednak, że sam język L nie jest kontekstowy. Załóżmy nie wprost, że jest on klasy $\mathcal{L}_1$. Posiada więc generującą go gramatykę kontekstową, która musi być ujęta w opisanej wyżej numeracji. Niech więc n będzie liczbą taką, że $L = L(G_n)$. Można zadać wówczas pytanie czy słowo binarne ω_n kodujące tę gramatykę jest elementem języka L. Gdyby było $\omega_n \in L$, wówczas $\omega_n \in L(G_n)$ skoro $L = L(G_n)$. Ale to oznacza na podstawie definicji (5.5) języka L, że $\omega_n \notin L$, mamy zatem sprzeczność. Gdy przeciwnie założyć, że $\omega_n \notin L$, a więc $\omega_n \notin L(G_n)$, wówczas jak poprzednio na podstawie (5.5) otrzymujemy $\omega_n \in L$, ponownie sprzeczność. Widzimy zatem, że przypuszczenie jakoby $L \in \mathcal{L}_1$ musi być fałszywe. W konsekwencji $\mathcal{L}_1 \subsetneq \mathcal{L}_{\text{Rec}}$. □

Dodajmy na koniec, że własności zamkniętości $\mathcal{L}_{\text{Rec}}$ ze względu na operacje na językach są identyczne jak w przypadku klasy $\mathcal{L}_1$, por. Twierdzenie 5.4.

V.4 Przestrzeń robocza i automaty liniowo ograniczone

Naszym celem jest teraz opis klasy automatów akceptujących języki z $\mathcal{L}_1$. Okazuje się, że pojęciem, które prowadzi do adekwatnego ograniczenia na tryb operowania pamięcią przez automat jest tzw. przestrzeń robocza. Zdefiniujemy ją dla dowolnej gramatyki typu 0 (por. Definicję 2.4).

DEFINICJA 5.4 *Niech $G = (A, V, S, \Pi)$ będzie dowolną gramatyką i rozważmy wyprowadzenie słowa $\omega \in L(G)$,*

$$D: \quad S = \omega_0 \Rightarrow \omega_1 \Rightarrow \ \ldots \ \Rightarrow \omega_n = \omega\,.$$

Przestrzenią roboczą wyprowadzenia D nazywamy wielkość

$$\mathcal{W}_G(\omega, D) \stackrel{df}{=} \max\{|\omega_i| \ : \ 0 \leqslant i \leqslant n\}\,.$$

Przestrzeń robocza słowa ω zdefiniowana jest z kolei jako

$$\mathcal{W}_G(\omega) \stackrel{df}{=} \min_D \mathcal{W}_G(\omega, D)\,.$$

Jeśli G jest gramatyką kontekstową, z uwagi na fakt, że wyprowadzenia są z definicji nieskracające, dla dowolnego D mamy $\mathcal{W}_G(\omega, D) = |\omega|$, a więc także $\mathcal{W}_G(\omega) = |\omega|$. W przypadku dowolnych gramatyk formy zdaniowe mogą w zasadzie rosnąć nieograniczenie, jeśli jednak potrafimy sensownie oszacować ten wzrost przez długość wyprowadzanego słowa, okazuje się, że generowany język jest kontekstowy. Oto odnośny wynik.

TWIERDZENIE 5.8 *Jeśli G jest dowolną gramatyką typu 0 oraz jeśli istnieje stała $M > 0$ taka, że*

$$\mathcal{W}_G(\omega) \ \leqslant \ M|\omega|$$

dla wszystkich niepustych słów $\omega \in L(G)$, wówczas $L(G)$ jest językiem kontekstowym.

Zauważmy, że M jest wielkością związaną z gramatyką G, wspólną dla wszystkich słów $\alpha \in L(G)$.

Z twierdzenia tego wynika w szczególności, że w przypadku gdy G opisuje język spoza klasy $\mathcal{L}_1$, wówczas dla każdej, nieważne jak dużej liczby M da się znaleźć w $L(G)$ słowo ω, którego przestrzeń robocza spełnia nierówność $\mathcal{W}_G(\omega) > M|\omega|$. Innymi słowy wyprowadzenia w gramatykach generujących języki spoza $\mathcal{L}_1$ mogą tworzyć pośrednie formy zdaniowe o bardzo wielkiej długości, mającej się nijak do długości generowanego słowa.

Twierdzenie powyższe zawiera sugestię co do ograniczeń jakie należy nałożyć na sposób wykorzystania pamięci roboczej przez automat, by jego "moc obliczeniowa" odpowiadała jak najlepiej "mocy generatywnej" gramatyk kontekstowych. Na tej bazie utworzona jest definicja automatu liniowo ograniczonego.

DEFINICJA 5.5 *Niedeterministyczny automat liniowo ograniczony (ALO) jest opisany jako*

$$M = (A, V, [,], \Sigma, s_0, S_F, \{L, R\}, \pi)\,,$$

gdzie:

A *jest alfabetem wejściowym*
V *jest alfabetem taśmy roboczej*
$[,] \in V$ *są znacznikami końców taśmy roboczej*
Σ *jest zbiorem stanów automatu,* $\sigma = \{s_0, s_1, \ldots, s_q\}$
$s_0 \in \Sigma$ *jest wyróżnionym stanem startowym*
$S_F \subseteq \Sigma$ *jest zbiorem stanów końcowych (akceptujących)*
L, R *są oznaczeniami kierunku ruchu głowicy automatu na taśmie roboczej*
π *jest relacją przejścia,* $\pi : \Sigma \times (A \cup \{\lambda\}) \times V \rightarrow \mathcal{F}(\Sigma \times V \times \{L, R\})$.

Działanie automatu liniowo ograniczonego jest podobne do działania automatu ze stosem, z tą różnicą, że taśma robocza może być zapisywana (po jednym znaku na ruch) i odczytywana w dowolnej kolejności, a jej głowica może przesuwać się o jedną pozycję w lewo lub w prawo w każdym ruchu. Jednak przestrzeń na taśmie roboczej jest ograniczona przez nieusuwalne znaczniki [oraz] i zawiera dokładnie tyle pozycji, ile jest liter napisu wejściowego. W świetle Twierdzenia 5.8 spodziewalibyśmy się raczej ograniczenia rozmiaru przestrzeni roboczej przez wielkość typu $M|\alpha|$. Zauważmy jednak, że standardowa wartość $M = 1$ nie jest w istocie żadnym ograniczeniem, bowiem można efektywnie powiększyć pojemność pamięci roboczej przez użycie bogatszego alfabetu V. Ważne jest jednak, że alfabet ten (a więc pojemność roboczej pamięci) jest ustalony dla automatu czyli dla akceptowanego języka, nie jest więc zależny od napisu wejściowego.

W stanie początkowym s_0 głowica taśmy wejściowej jest standardowo ustawiona na pierwszej literze badanego napisu, natomiast głowica taśmy roboczej skanuje pierwszy symbol na prawo od znacznika [. Zakładamy przy tym, że taśma ta wypełniona jest symbolami "pustymi", np. O.

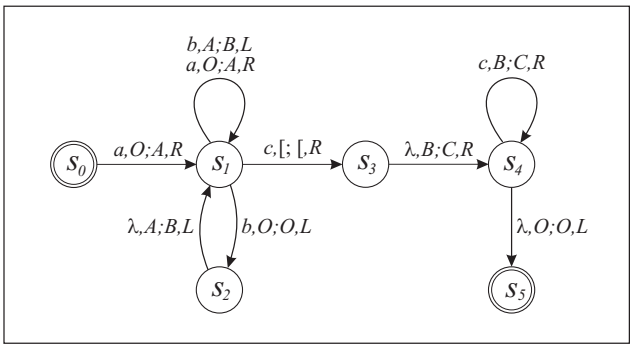
Słowo wejściowe zostaje zaakceptowane, jeśli automat osiągnie jego prawy koniec i znajdzie się w jednym ze stanów końcowych z S_F. W każdym innym wypadku słowo zostaje odrzucone.

Podsumowując, relacja przejścia ALO opisuje alternatywne ruchy, np.

$$\pi(s, a, X) \ni (s', Y, L)$$

interpretowane następująco: jeśli automat znajduje się w stanie s, czytając literę a z taśmy wejściowej, natomiast skanowanym aktualnie symbolem na taśmie roboczej jest X, automat przechodzi do stanu s', zapisując Y w miejscu X i przesuwając głowicę na taśmie roboczej o jedną pozycję w lewo. Głowica taśmy wejściowej przesuwa się zawsze o jedną pozycję w prawo z wyjątkiem ruchów "pustych" postaci $\pi(s, \lambda, X)$, gdy następuje zmiana stanu i działanie na taśmie roboczej bez badania napisu wejściowego i bez przesunięcia głowicy czytającej. Przejścia, w których skanowanym symbolem na taśmie roboczej jest znak [lub] pozostawiają go bez zmian, wykonując obowiązkowo ruch w prawo lub odpowiednio w lewo.

PRZYKŁAD 5.4 Automat akceptujący język $L = \{a^n b^n c^n : n \geqslant 0\}$ skanując napis wejściowy zapisuje na taśmie roboczej symbole A dopóki czytane litery to a. Po osiągnięciu litery b automat cofa się na taśmie roboczej zamieniając symbole A w B. W



Rys. 5.1: Automat liniowo ograniczony akceptujący język $\{a^n b^n c^n : \ n \geqslant 0\}$ z Przykładu 5.4. Czytelnik zechce zwrócić uwagę na wykorzystanie pustych ruchów z s_2 i s_3 do uzgodnienia pozycji głowic na taśmie wejściowej i roboczej.

końcu po rozpoznaniu pierwszej litery c ruch głowicy na taśmie roboczej ponownie zmienia kierunek. Niezgodność liczby liter powoduje zatrzymanie automatu z braku odpowiedniego przejścia lub, np. w przypadku nadmiaru liter c, przed osiągnięciem końca napisu wejściowego. Zakładamy jak wyżej, że głowica taśmy roboczej znajduje się w pozycji bezpośrednio za znacznikiem [i że taśma ta wypełniona jest symbolami O. Rozwiązanie przedstawiamy w postaci grafu, p. Rys. 5.1. Przeanalizujmy działanie automatu na napisie wejściowym *aabbcc*. Opiszemy je przy pomocy relacji przejścia. Przypomnijmy, że konfiguracja automatu reprezentowana jest przez ciąg

$$a_1 a_2 \ldots s a_i \ldots a_n [X_1 X_2 \ldots \underline{X_k} \ldots X_n] \,,$$

gdzie pierwsza część opisuje napis wejściowy $a_1 \ldots a_n$, a druga zawartość taśmy roboczej $X_1 \ldots X_n$. Aktualny stan s wskazuje pozycję głowicy na taśmie wejściowej, a podkreślenie pozycję głowicy na taśmie roboczej. Mamy więc

$$\begin{aligned}
s_0 aabbcc[\underline{O}OOOOO] \ &\Rightarrow \ a s_1 abbcc[A\underline{O}OOOO] \ \Rightarrow \ aas_1 bbcc[AA\underline{O}OOO] \\
&\Rightarrow \ aabs_2 bcc[A\underline{A}OOOO] \ \Rightarrow \ aabs_1 bcc[\underline{A}BOOOO] \\
&\Rightarrow \ aabbs_1 cc[\underline{B}BOOOO] \ \Rightarrow \ aabbcs_3 c[\underline{B}BOOOO] \\
&\Rightarrow \ aabbcs_4 c[C\underline{B}OOOO] \ \Rightarrow \ aabbccs_4 [CC\underline{O}OOO] \\
&\Rightarrow \ aabbccs_5 [C\underline{C}OOOO] \,.
\end{aligned}$$

Standardowa definicja automatu liniowo ograniczonego opisuje go jako urządzenie niedeterministyczne. Istnieją także deterministyczne automaty tego typu, jednak problem ich równoważności bądź nierównoważności z niedeterministycznymi nie został jak dotąd rozwiązany. Jest to jeden z najbardziej znanych otwartych problemów w teorii języków formalnych.

Zakończymy obecny rozdział ważnym twierdzeniem.

TWIERDZENIE 5.9 *Klasa języków akceptowanych przez automaty liniowo ograniczone jest identyczna z klasą języków kontekstowych* $\mathcal{L}_1$.

Dowód tego twierdzenia wykorzystuje fakt, że przestrzeń robocza gramatyki kontekstowej jest dokładnie równa długości wyprowadzanego słowa. Automat liniowo ograniczony jest więc w stanie zasymulować proces takiego wyprowadzenia na taśmie roboczej, jeśli jego przejścia są odpowiednio powiązane z produkcjami gramatyki. Odwrotnie, pokazuje się, że reguły przejścia automatu liniowo ograniczonego mogą być przetworzone do nieskracających produkcji odpowiedniej gramatyki.

Automaty liniowo ograniczone posiadają także równoważną, jednotaśmową reprezentację jako standardowo zdefiniowane maszyny Turinga (p. rozdział następny), jednak z liniowym względem długości danych ograniczeniem na rozmiar wykorzystywanej pamięci. Ma to bezpośredni związek z charakteryzacją języków kontekstowych zapisaną w Twierdzeniu 5.8.

V.5 Zadania do Rozdziału V

Zadanie 1

Znaleźć gramatyki kontekstowe dla następujących języków:

a) $L = \{a^{n-1}b^n c^{n+1} : \ n \geqslant 1\}$

b) $L = \{a^n b^n a^{2n} : \ n \geqslant 0\}$

c) $L = \{a^n b^n a^n b^n : \ n \geqslant 0\}$

d) $L = \{a^{n-1}b^n c^{n+1} : \ n \geqslant 1\}$

e) $L = \{\omega \overleftarrow{\omega}\, \omega : \ \ \omega \in \{a,b\}^*\}$

f) $L = \{\omega \in \{a,b,c\}^* : \ \ |\omega|_a = |\omega|_b = |\omega|_c\}$

g) $L = \{\omega \in \{a,b,c\}^* : \ \ |\omega|_a,\ |\omega|_b,\ |\omega|_c$ są różne$\}$

h) $L = \{a^n b^{n^2} : \ n \geqslant 1\}$

Zadanie 2

Narysować grafy automatów liniowo ograniczonych akceptujących języki z poprzedniego zadania.

Zadanie 3

Załóżmy, że mamy opis automatu liniowo ograniczonego akceptującego język L. Pokazać, że można uzupełnić i zmodyfikować ten opis tak, by uzyskać automat liniowo ograniczony akceptujący język $\overleftarrow{L}$.

Stanowi to dowód faktu, że klasa $\mathcal{L}_1$ jest zamknięta ze względu na lustrzane odbicie.

Rozdział VI

Gramatyki typu 0 i maszyny Turinga

Wiemy już, że gramatyki typu 0 (p. Definicja 2.4), przez dopuszczenie wyprowadzeń skracających formy zdaniowe, stają się niedoskonałym narzędziem opisu języków, bowiem nie gwarantują możliwości w pełni efektywnego testowania dowolnych słów pod kątem ich należenia bądź nienależenia do języka. Pełnią one jednak w hierarchii Chomsky'ego bardzo ważną rolę, ponieważ są funkcjonalnie równoważne pojęciu *obliczalności*, któremu w tym rozdziale nadamy precyzyjny sens. Wyznaczają więc granice tego co intuicyjnie rozumiemy jako zagadnienia konstruktywnie rozwiązalne.

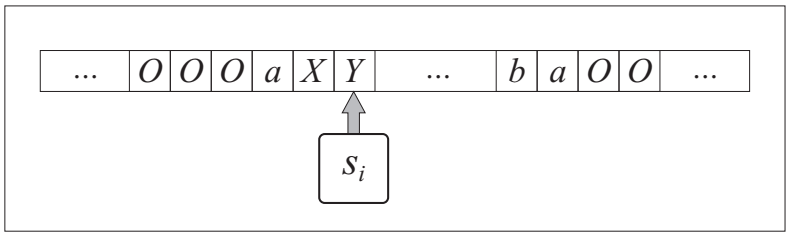
Systematyczny przegląd drzewa wyprowadzeń, w którym formy zdaniowe mogą na przemian wydłużać się i skracać, mimo wspomnianej wady, może jednak służyć jako podstawa algorytmu wypisującego systematycznie wszystkie słowa z języka: jeśli podczas przeglądu kolejnego piętra drzewa napotkamy napis terminalny, wypisujemy go. Oczywiście algorytm taki działa w nieskończonym czasie, być może wypisując niektóre słowa wielokrotnie, jednak każde słowo z języka pojawi się na tworzonej liście w skończonym, choć nieograniczonym czasie. Jest to bodaj najsłabsza, wciąż konstruktywna definicja języka jaką jesteśmy skłonni przyjąć.

VI.1 Maszyny Turinga

Maszyny Turinga są najbardziej ogólnymi automatami akceptującymi języki — jak pokażemy, równoważnymi gramatykom typu 0. Zostały zdefiniowane w 1936 roku przez brytyjskiego matematyka Alana M. Turinga [13], znanego także z badań, które doprowadziły do złamania szyfru Enigmy podczas II Wojny Światowej. Definicję Turinga probowano rozszerzać na wiele sposobów w nadziei uzyskania bardziej ogólnych modeli automatu. Jednak zawsze zabiegi te prowadziły jedynie do konstrukcji urządzeń o możliwościach obliczeniowych dokładnie takich samych, jak oryginalne maszyny Turinga. Na podstawie tej obserwacji amerykański matematyk Alonzo Church sformułował słynną hipotezę mówiącą o tym, że maszyny Turinga stanowią najbardziej ogólną formalną definicję intuicyjnego pojęcia efektywnej procedury, [3].

1 Klasyczna definicja

Rozpocznijmy od standardowej definicji deterministycznej maszyny Turinga.



Rys. 6.1: Klasyczna jednotaśmowa maszyna Turinga.

DEFINICJA 6.1 *Deterministyczną maszyną Turinga nazywamy automat określony przez*

$$M = (A, V, O, \Sigma, s_0, S_F, \{L, R\}, \pi)\,,$$

gdzie:

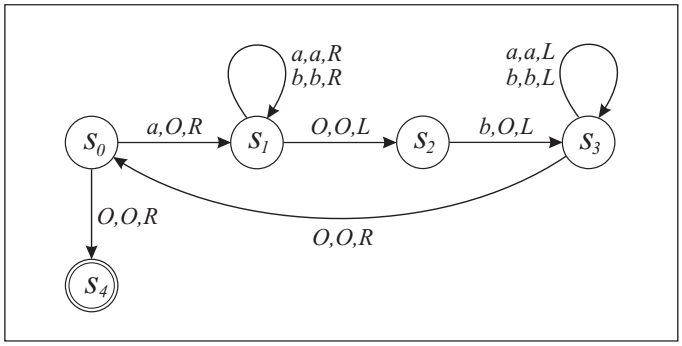
A	*jest alfabetem wejściowym*
V	*jest alfabetem taśmy,* $A \subseteq V$
$O \in V - A$	*jest symbolem pustej klatki na taśmie*
Σ	*jest zbiorem stanów automatu,* $\sigma = \{s_0, s_1, \ldots, s_q\}$
$s_0 \in \Sigma$	*jest wyróżnionym stanem startowym*
$S_F \subseteq \Sigma$	*jest zbiorem stanów końcowych (akceptujących)*
L, R	*są oznaczeniami kierunku ruchu głowicy automatu*
π	*jest częściową funkcją przejścia,* $\pi : \Sigma \times V \nrightarrow \Sigma \times V \times \{L, R\}$.

Maszyna Turinga wyposażona jest w potencjalnie nieskończoną pamięć taśmową, jednak każde "obliczenie" wykorzystuje jedynie skończony choć a priori nieograniczony jej odcinek. Klatki taśmy standardowo zawierają symbol pusty O, za wyjątkiem odcinka, w którym zapisany jest napis wejściowy. Automat startuje w stanie s_0 z głowicą umieszczoną w pozycji pierwszej litery napisu wejściowego. Maszyna w każdym kroku wykonuje ruch opisany przez funkcję π. I tak np.

$$\pi(s, x) = (s', y, R)\,, \qquad s, s' \in \Sigma,\ \ x, y \in V$$

opisuje ruch automatu polegający na tym, że jeśli znajduje się on w stanie s, a czytanym aktualnie symbolem jest x, następuje zmiana stanu na s', symbol y jest zapisywany w miejsce x, a głowica wykonuje ruch w prawo do następnej klatki. Jeśli wartość $\pi(s, x)$ jest nieokreślona, maszyna zatrzymuje się, przy czym jeśli takie zatrzymanie następuje w stanie $s \in S_F$, traktujemy to jako akceptację napisu wejściowego, w przeciwnym zaś wypadku — jako odrzucenie.

Widzimy więc, że działanie maszyny Turinga polega na tym, że jej głowica przemierza, być może wielokrotnie, obszar zawierający napis wejściowy, zmieniając przy tym jego litery na symbole pomocnicze bądź terminalne. Może także zmieniać rozmiar obszaru roboczego na taśmie zapisując puste klatki dowolnymi symbolami lub odwrotnie. Przy okazji, widać teraz rolę założenia $O \in V - A$: symbol pustej klatki musi się różnić od liter alfabetu A, by mógł pełnić rolę znacznika początku i końca napisu wejściowego.



Rys. 6.2: Maszyna Turinga akceptująca język $L = \{a^n b^n : \ n \geqslant 0\}$, Przykład 6.1.

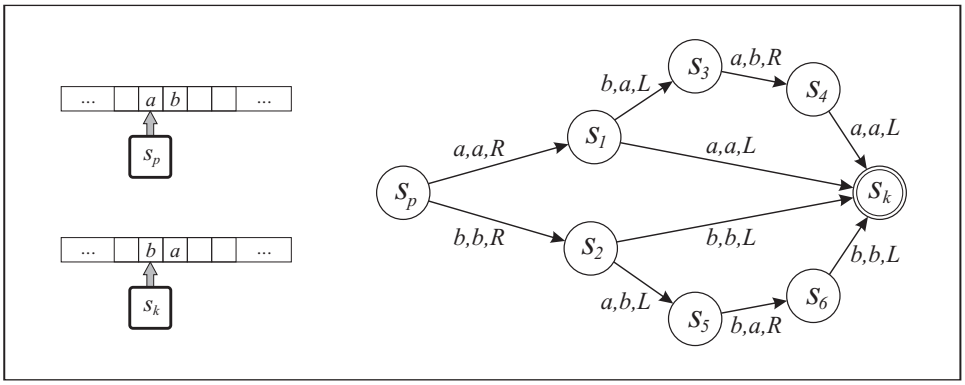
DEFINICJA 6.2 *Językiem akceptowanym przez maszynę Turinga M nazywamy zbiór tych napisów nad alfabetem A, dla których M zatrzymuje się w jednym ze swoich stanów końcowych.*

Zilustrujemy ten formalny opis konkretnym przykładem.

PRZYKŁAD 6.1 Ponownie rozważymy język $L = \{a^n b^n : \ n \geqslant 0\}$, opisując tym razem akceptującą go maszynę Turinga. Metoda, którą zastosujemy polega na tym, że czytana początkowa litera a jest zamieniana na symbol pustej klatki O, maszyna zaś przechodzi do stanu s_1, w którym przesuwa głowicę na koniec napisu, bez zmiany kolejnych liter. Po osiągnięciu symbolu O ograniczającego słowo z prawej strony, głowica cofa się o jedną klatkę (stan s_2) i wymazuje literę b, jeśli taka, zgodnie z oczekiwaniami, się tam znajduje. W stanie s_3 maszyna cofa się na początek słowa i cały opisany cykl powtarza się. W stanie s_0, gdy głowica znajduje się nad pierwszą literą nieusuniętej dotąd części słowa, określone są jedynie przejścia dla liter a i O. Pojawienie się litery b powoduje zatrzymanie maszyny, a ponieważ s_0 nie jest stanem końcowym, słowo zostanie odrzucone. Symbol pustej klatki w tym miejscu to albo puste słowo początkowe λ, albo konsekwencja wymazania wszystkich par liter a i b. Podobnie, wobec nieokreśloności przejścia ze stanu s_2 z literą a, słowo zostaje odrzucone jeśli na jego końcu pojawia się a zamiast spodziewanego b. Rys. 6.2 przedstawia graf opisanej tu maszyny Turinga. Sugerujemy czytelnikowi przeanalizowanie jej działania na przykładowych napisach *aabb*, *aab* i *abb*.

Oprócz tradycyjnej interpretacji automatu jako urządzenia rozpoznającego słowa danego języka, czyli tzw. akceptora, maszyna Turinga może być uważana także za urządzenie obliczające, a więc tzw. komputer. W pierwszym przypadku istotny jest stan, w którym nastąpiło zatrzymanie, nie jest istotna natomiast końcowa zawartość taśmy. W drugim przypadku, po zatrzymaniu w stanie końcowym, zawartość taśmy interpretowana jest jako wynik obliczeń wykonanych na danych wejściowych. Zwróćmy jednak uwagę na całkowitą równoważność obydwu tych interpretacji. Rozpoznawanie słów dowolnego języka jest wszelako niczym innym jak procesem obliczającym pewną funkcję χ_L zdefiniowaną jako

$$\chi_L(\alpha) = \begin{cases} 1 & \text{gdy } \alpha \in L \\ 0 & \text{gdy } \alpha \notin L\,. \end{cases}$$



Rys. 6.3: Operacja zamiany dwóch sąsiednich liter, Przykład 6.2.

Z drugiej strony, każdy proces obliczeniowy wyznaczający wartości $f : \mathbb{N} \to \mathbb{N}$ można zamienić w problem rozpoznawania języka[1]

$$L \;=\; \{a^n b^{f(n)} :\; n \in \mathbb{N}\}\,.$$

Ponadto przez "obliczenia" nie należy rozumieć wyłącznie procedur arytmetycznych, powinniśmy myśleć ogólniej o dowolnych operacjach na ciągach znaków. Kolejne trzy przykłady ilustrują różne elementarne przekształcenia napisu wejściowego, jedne z wielu, które mogą być zrealizowane przez maszynę Turinga.

Przykład 6.2 Rys. 6.3 przedstawia fragment grafu maszyny Turinga, którego działanie polega na zamianie miejscami dwóch kolejnych liter: tej, która aktualnie znajduje się pod głowicą i następnej. Można łatwo wyobrazić sobie umieszczenie tego fragmentu w zapętlonej części grafu, by uzyskać np. procedurę przemieszczania określonego znaku na koniec napisu. Zwróćmy uwagę na to, jak można używać stanów do chwilowego przechowania informacji o tym, jaka był wcześniejsza litera. Np. stan s_1 inicjuje kolejne kroki, gdy przeczytaną literą było a, natomiast s_2, gdy było nią b. Zakładamy, że po zamianie liter głowica wraca do pierwotnej pozycji, choć oczywiście w konkretnej sytuacji można przyjąć inne rozwiązanie.

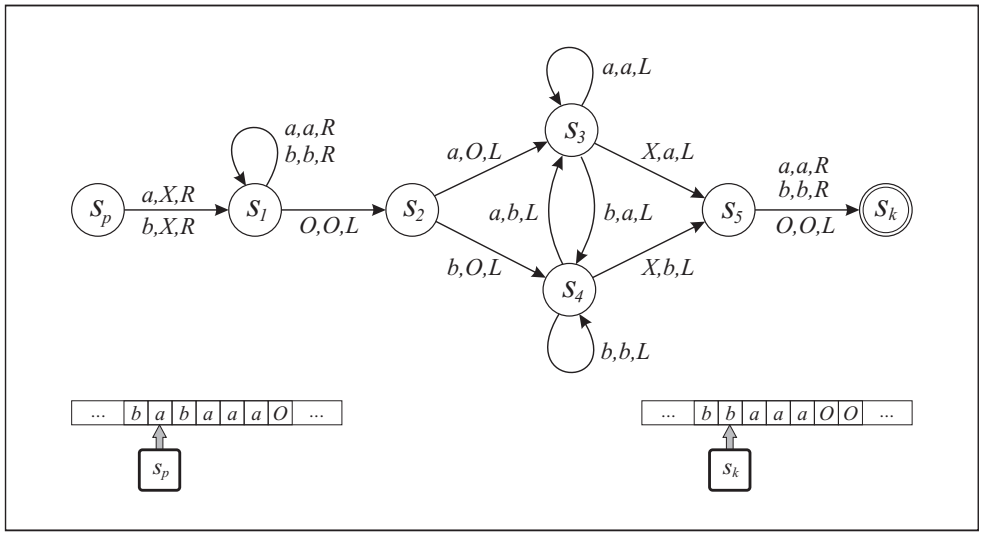
Przykład 6.3 Kolejny przykład, Rys. 6.4, to realizacja wymazania znajdującego się pod głowicą znaku, z przepisaniem całego fragmentu słowa na prawo od niego o jedną pozycję na taśmie w lewo. Symbol pomocniczy X ustawiony początkowo w pozycji usuwanego znaku pełni rolę znacznika miejsca na taśmie, w którym ma się zakończyć proces przesuwania dalszej części słowa. Po ustawieniu tego wskaźnika maszyna przemieszcza głowicę na koniec słowa, skąd, cofając się w lewo, nadpisuje kolejne litery wartościami ich następników.

Przykład 6.4 Końcowy przykład, Rys. 6.5, pokazuje jak wstawić nową klatkę na taśmie wewnątrz napisu, przepisując dalszą jego część o jedno miejsce w prawo. Symbol X pełni rolę tymczasowego wypełnienia tego miejsca (z oczywistych powodów

[1] Ogólniej $f : \mathbb{N}^k \to \mathbb{N}^l$ odpowiada językowi

$$L = \{a_1^{n_1} \dots a_k^{n_k} b_1^{f_1(n_1,\dots,n_k)} \dots b_l^{f_l(n_1,\dots,n_k)} \;:\; (n_1,\dots,n_k) \in \mathbb{N}^k\}$$

nad alfabetem $A = \{a_1,\dots,a_k,b_1,\dots,b_l\}$.



Rys. 6.4: Wymazanie wskazanego znaku ze słowa, Przykład 6.3.

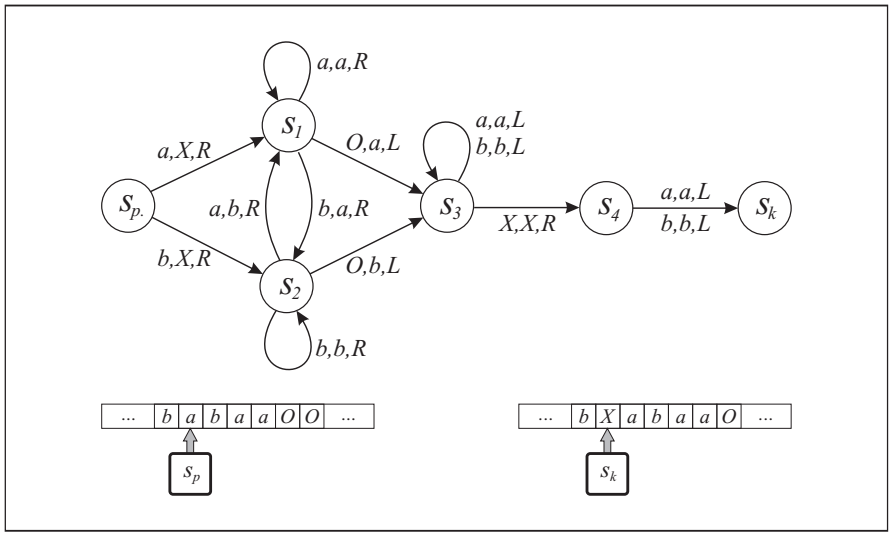
unikamy użycia symbolu O w tej roli). Zwróćmy uwagę na podobieństwo grafów automatu w tym i w poprzednim przykładzie. Są to wszakże operacje odwrotne względem siebie.

Nieco bardziej czasochłonne, choć równie nieskomplikowane są zadania, w których należy skonstruować maszynę Turinga wykonującą określone operacje arytmetyczne. Jedno z zadań na końcu rozdziału dotyczy takiej właśnie implementacji sumatora binarnego. Dysponując opisami maszyn realizujących operacje arytmetyczne, możemy użyć ich do zbudowania bardziej złożonych automatów. Jeśli np. taśma wejściowa zawiera dwie liczby binarne o wartościach m i n oddzielone znakami separatora, powiedzmy $\#$, można łatwo opisać automat, który implementuje operację

$$\textbf{if } m \leqslant n \textbf{ then } \text{oblicz } m+n \textbf{ else } \text{oblicz } 2m+1\,.$$

Rozwiązanie polegać może na obliczeniu różnicy $n-m$ i zależnie od wyniku przejścia do stanu, który inicjuje obliczanie $m+n$ lub odpowiednio $2m+1$. Zmierzamy do tego, aby uświadomić czytelnikowi, że niezwykle prosty repertuar elementarnych ruchów maszyny Turinga wystarcza do realizacji złożonych algorytmów. Oczywiście próba implementowania tą metodą nawet zwykłego algorytmu Euklidesa dla NWP jest raczej mało praktycznym zajęciem, bo potrafimy to zrobić znacznie szybciej np. przy pomocy programu w C. Istotne jest jednak to, że maszyny Turinga mogą w zasadzie realizować te same zadania, do których stosujemy w codziennej praktyce wysokopoziomowe języki programowania.

Powyższa obserwacja ma fundamentalne znaczenie dla współczesnej matematyki obliczeniowej, w tym dla samej teorii języków formalnych. "Moc obliczeniowa" maszyn Turinga pozwala bowiem symulować na nich działanie innych automatów, bądź w inny sposób formalnie opisanych procesów algorytmicznych, pokazując uniwersalność automatu Turinga. Poniżej opisujemy jeden z najważniejszych przykładów wykorzystania takiego "symulacyjnego" argumentu.



Rys. 6.5: Operacja wstawienia nowego symbolu przed wskazanym znakiem, Przykład 6.4.

Definicja niedeterministycznej maszyny Turinga różni się od deterministycznej jedynie tym, że π jest relacją, a więc, że przejścia $\pi(s,a)$ mogą być wieloznaczne,

$$\pi(s,a) = \{(s_{i_1}, X_{i_1}, Q_{i_1}), \ldots, (s_{i_k}, X_{i_k}, Q_{i_k})\}, \quad s_{i_j} \in \Sigma, \;\; X_{i_j} \in V, \;\; Q_{i_j} \in \{L, R\}.$$

Słowo $\alpha \in A^*$ jest akceptowane przez niedeterministyczną maszynę Turinga jeśli dla danych wejściowych postaci α istnieje przynajmniej jedna sekwencja jej ruchów kończąca się zatrzymaniem w stanie finalnym $s \in S_F$. Następny przykład ukazuje różnicę w działaniu maszyny deterministycznej i niedeterministycznej.

Przykład 6.5 Przeanalizujmy opisowo rozwiązania niedeterministyczne i deterministyczne dla języka $L = \{\omega\omega : \;\; \omega \in \{a,b\}^*\}$. Jak już wiemy, jest to język kontekstowy, bo znamy generującą go gramatykę kontekstową, p. Przykład 5.3. Podstawowym problemem w przypadku analizy bottom-up słów z L przy pomocy automatu jest poprawne wyznaczenie punktu podziału słowa, tak by można było następnie porównać obie jego części. Niedeterministyczne rozwiązanie może po prostu próbować "odgadnąć" właściwy punkt podziału. Niech bowiem niedeterministyczna maszyna Turinga rozpoczyna swoje działanie od przesunięcia głowicy o kilka klatek w prawo,

$$\pi(s_0,a) = \{(s_0,a,R),(s_1,a,R)\}, \qquad \pi(s_0,b) = \{(s_0,b,R),(s_1,b,R)\},$$

przy czym w dowolnym momencie może nastąpić (niedeterministycznie) wybór stanu s_1. Stan s_1 inicjuje dalej proces wstawiania nowej klatki z symbolem X w miejscu w którym znalazła się głowica, tak jak w Przykładzie 6.4, a symbol ten traktowany jest odtąd jako znacznik punktu podziału. Pozostaje więc już tylko porównać na zgodność znaki części po lewej i po prawej stronie X. Zgodnie z definicją słowo zostanie zaakceptowane, jeśli *istnieje* sekwencja ruchów maszyny prowadząca do stanu finalnego, a dla słów postaci $\omega\omega$ (i tylko dla nich!) taka seria ruchów oczywiście istnieje: wystarczy aby maszyna niedeterministycznie wstawiła znak X dokładnie na środku słowa.

W przypadku rozwiązania deterministycznego punkt podziału musi być precyzyjnie wyznaczony. Można to osiągnąć np. następującą metodą. Ustawmy najpierw po jednej literze pomocniczej C na początku i na końcu badanego słowa. Następnie wykonujmy cyklicznie przesunięcie początkowego C o jedną literę w prawo, a końcowego — o jedną w lewo. Jeśli po wykonaniu kolejnej takiej pary przestawień dojdzie do spotkania się znaków C, wyznaczyliśmy środek napisu. Pozostaje faza porównania, którą łatwo jest zrealizować deterministycznie, p. Zadanie 4.

Jak pokazuje ostatni przykład, wykorzystanie niedeterminizmu może w niektórych przypadkach istotnie uprościć konstrukcję maszyn Turinga. Każda deterministyczna maszyna jest oczywiście szczególną postacią maszyny niedeterministycznej. Zachodzi jednak następujące, bardzo ważne twierdzenie.

TWIERDZENIE 6.1 *Dla każdej niedeterministycznej maszyny Turinga istnieje równoważna (w sensie akceptowanego języka) maszyna deterministyczna.*

Konstrukcja maszyny deterministycznej na podstawie opisu niedeterministycznej, o której mowa w twierdzeniu, jest tym razem bardziej skomplikowana niż miało to miejsce to w przypadku skończonych automatów, p. Twierdzenie 3.1. Prosta metoda polegająca na utożsamieniu stanów nowego automatu z elementami zbioru potęgowego starych stanów nie może być zastosowana, bowiem relacja przejścia maszyny Turinga zwraca jako wartości także symbole $X \in V$ do zapisania na taśmie oraz określa kierunki ruchu głowicy. Nie wiadomo np. jak deterministycznie opisać niedeterministyczne przejście postaci

$$\pi(s, x) = \{(s, x, L), (s, x, R)\}\,,$$

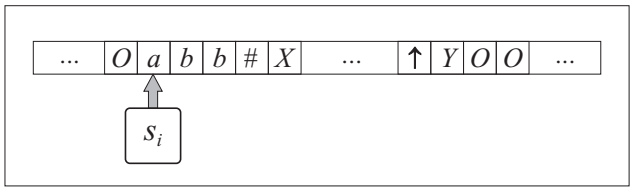
polegające na tym, że maszyna nie zmienia stanu ani zapisanej na taśmie litery, lecz niedeterministycznie przesuwa głowicę w lewo lub w prawo. Nie będziemy wchodzić w szczegóły techniczne dowodu, poprzestając jedynie na krótkim opisie idei. Można wyobrazić sobie alternatywne sekwencje kroków maszyny niedeterministycznej w postaci drzewa, w którego węzłach zapisujemy konfiguracje automatu

$$x_1 x_2 \ldots\, s_i\, x_k \ldots x_n\,, \qquad x_j \in V,\ \ s_i \in \Sigma\,.$$

Jak wcześniej, $x_1 \ldots x_n$ oznaczają litery zapisane na taśmie, a s_i — aktualny stan, wskazujący jednocześnie pozycję głowicy. Krawędzie w tym drzewie odpowiadają relacji przejścia między konfiguracjami, np.

$$x_1 x_2 \ldots\, s_i\, x_k \ldots x_n \ \Rightarrow\ x_1 x_2 \ldots y\, s_j\, x_{k+1} \ldots x_n$$
$$\text{wtedy i tylko wtedy, gdy} \qquad \pi(s_i, x_k) \ni (s_j, y, R)\,.$$

Można tak zorganizować działanie maszyny deterministycznej, by symulowała ona systematyczny przegląd wszerz drzewa relacji przejścia maszyny niedeterministycznej. Zapisana na taśmie konfiguracja startowa $s_0 a_1 \ldots a_n$ jest przekształcana na ciąg alternatywnie osiągalnych z niej konfiguracji, opisanych przez relację przejścia $\pi(s_0, a_1)$ maszyny niedeterministycznej. Elementy tego ciągu na taśmie maszyny



Rys. 6.6: Symulacja automatu dwutaśmowego przez maszynę Turinga.

deterministycznej oddzielone są specjalnymi separatorami, np. #. Jest to pierwsze piętro przeglądanego drzewa. Odczytując kolejne elementy tego ciągu maszyna buduje na taśmie obraz drugiego piętra. Gdy osiągnięta w ten sposób zostanie konfiguracja akceptująca oryginalnej maszyny, deterministyczny automat zatrzymuje się w swoim stanie finalnym, jeśli nie — działa dalej wg. tego samego schematu.

Skoro deterministyczna i niedeterministyczna wersja maszyny Turinga okazują się równoważne pod względem zdolności obliczeniowej, proponując rozwiązania różnych teoretycznych problemów możemy wybierać rozwiązania niedeterministyczne, jeśli są prostsze. Gdyby zaś chodziło o względy implementacyjne, należy pamiętać, że złożoność obliczeniowa procesu niedeterministycznego jest na ogół istotnie mniejsza niż złożoność jego deterministycznego równoważnika. Ocena złożoności będzie przedmiotem dyskusji w Rozdziale VII.

Innym przykładem ilustrującym uniwersalność maszyn Turinga jest następująca argumentacja, dowodząca, że przyjęta przez nas w Rozdziale II ogólna definicja automatu, por. Rys. 2.1, str. 36, wyposażonego w taśmę wejściową i osobną pamięć roboczą, nie jest wbrew pozorom bardziej pojemna od jednotaśmowej maszyny Turinga. Automat Turinga może bowiem, przez odpowiednią rozbudowę alfabetu pomocniczego V i uzupełnienie relacji przejścia, zasymulować działanie takiej dwutaśmowej maszyny na swojej pojedynczej taśmie. Wystarczy przyjąć, że napis wejściowy jest ograniczony z prawej strony specjalnym znakiem końca, np. #, natomiast przestrzeń taśmy na prawo od tego znaku pełni rolę pamięci roboczej. Drugi specjalny znak, np. ↑, pełni rolę znacznika pozycji głowicy na symulowanej w ten sposób taśmie roboczej. Maszyna czyta pierwszy znak z części wejściowej, zapamiętując go przy pomocy swojego stanu, przy czym natychmiast zamienia go na symbol pusty O. W ten sposób późniejszy powrót głowicy do części wejściowej zatrzyma ją na kolejnej literze, gwarantując, że odczyt napisu wejściowego odbywa się znak po znaku od lewej do prawej strony. Po przeczytaniu i zapamiętaniu znaku wejściowego maszyna przesuwa głowicę w prawo aż do znaku ↑ i wykonuje operacje wynikające z opisu symulowanego automatu, korzystając z części roboczej taśmy od wskazanego miejsca. Symulacja ruchu głowicy roboczej polega na odpowiednim przeniesieniu znaku ↑ po wykonanej operacji. Następuje powrót do części wejściowej po kolejny znak. Osiągnięcie w tej fazie znaku # oznacza, że napis wejściowy został przeczytany do końca, p. Rys. 6.6.

Warto zwrócić uwagę, że jeśli w trybie takiej symulacji maszyna Turinga nie korzysta w ogóle z roboczej części taśmy, wówczas symuluje ona skończony automat deterministyczny. Jeśli korzysta z tej części jak ze stosu, ustawiając znak ↑ zawsze przed ostatnim zapisanym tam symbolem, symulacja imituje działanie automatu ze stosem. W końcu jeśli założyć, że część robocza podczas obliczeń nie przekracza

długością części wejściowej, mamy działanie symulujące automat liniowo ograniczony. Podając opis sposobu symulacji pewnego typu automatu przez maszynę Turinga dowodzimy jednocześnie, że ten automat jest niczym więcej jak tylko szczególnym, ograniczonym typem maszyny Turinga.

2 Inne modele maszyn Turinga

W Definicji 6.1 każdemu przejściu maszyny Turinga towarzyszy obowiązkowy ruch głowicy, w lewo lub w prawo. Patrząc na omawiane przez nas przykłady można uznać to za pewną niedogodność, bowiem często potrzebny jest dodatkowy ruch cofający głowicę do poprzedniej pozycji. Można więc zastanawiać się nad rozszerzeniem definicji maszyny w taki sposób, aby dopuszczalne były przejścia bez ruchu głowicy, $\pi(s,x) = (q,y,N)$. Pytanie, które natychmiast należałoby postawić, to czy zdefiniowana w ten sposób maszyna "potrafi" istotnie więcej od klasycznego automatu Turinga? Pokażemy że obie definicje są najzupełniej równoważne.

Metoda dowodu równoważności dwóch rodzajów automatu polega zawsze na opisie symulacji działania jednego z nich przez drugi i vice versa.

Niech więc $M = (A,V,O,\Sigma,s_0,S_F,\{L,R,N\},\pi)$ będzie rozszerzoną maszyną Turinga. Nie ulega wątpliwości, że jest ona w stanie wykonać wszystkie ruchy dowolnej standardowej maszyny wprost z opisu jej funkcji przejścia π' , bowiem ruchy te po porostu nie korzystają z dyrektywy N.

By opisać symulację w odwrotna stronę, wyobraźmy sobie standardową maszynę $M' = (A,V,O,\Sigma',s_o,S_F,\{L,R\},\pi')$, gdzie $\Sigma \subset \Sigma'$ oraz π' kopiuje w pierwszej kolejności wszystkie przejścia π rozszeronej maszyny nie zawierające dyrektywy N,

$$\pi'(s,x) = \pi(s,x) = (q,y,L \text{ lub } R)\,.$$

Natomiast dla każdego z jej ruchów postaci $\pi(s,x) = (q,y,N)$, funkcja przejścia maszyny standardowej zawiera ruchy

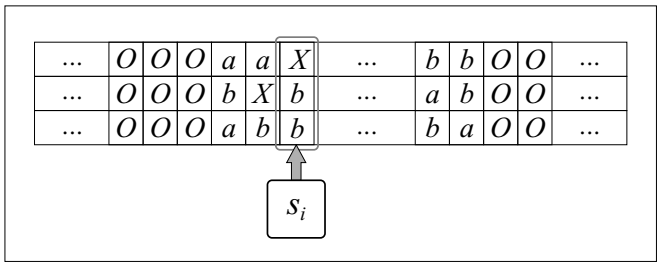
$$\pi'(s,x) = (q^{(N)},y,R) \qquad \text{oraz} \qquad \pi'(q^{(N)},z) = (q,z,L) \quad \text{dla wszystkich} \;\; z \in V\,,$$

gdzie $q^{(N)}$ jest nowym stanem dodanym do zbioru Σ' standardowej maszyny dla każdego stanu $q \in \Sigma$, który jak wyżej pojawia się w przejściu zawierającym dyrektywę N.

Innym sposobem rozszerzenia standardowej maszyny Turinga, który można spotkać w literaturze jest wyposażenie jej w wielościeżkową taśmę, p. Rys. 6.7. Posiada ona jak poprzednio jedną głowicę, natomiast taśma podzielona jest na kilka równoległych ścieżek, na każdej z których zapisywane mogą być litery z V. Odczyt i zapis w danym położeniu głowicy odbywa się jednocześnie na wszystkich ścieżkach, np.

$$\pi(s,(a,b,c)) = (q,(x,y,z),L)\,.$$

Chwila zastanowienia uświadomi nam, że model powyższy nie różni się niczym od standardowego, wystarczy bowiem przyjąć, że alfabet V' nowej maszyny zawiera symbole złożone, takie jak np. (X,b,b), a więc w przypadku maszyny trójścieżkowej jest zbiorem postaci $V \times V \times V$. Dlatego model wielościeżkowy nazywany jest czasem alternatywnie wektorową maszyną Turinga.



Rys. 6.7: Wielościeżkowa maszyna Turinga.

Mniej oczywista na pierwszy rzut oka jest natomiast równoważność standardowej wersji z tzw. *wielotaśmową* maszyną Turinga. Wyjaśnijmy, że chodzi tym razem o istotnie inny model niż ten, który opisaliśmy jako ogólny schemat automatu w Rozdziale II i który analizowaliśmy pod kątem jego relacji z klasyczną maszyną Turinga w poprzednim podrozdziale. Taśma wejściowa pełniła tam jedynie rolę pasywną, przechowując dane. Tym razem wszystkie taśmy są *aktywne*, a więc mogą być odczytywane i zapisywane niezależnie, niezależny i nieograniczony jest także ruch ich głowic. Funkcja przejścia ma więc następującą postać

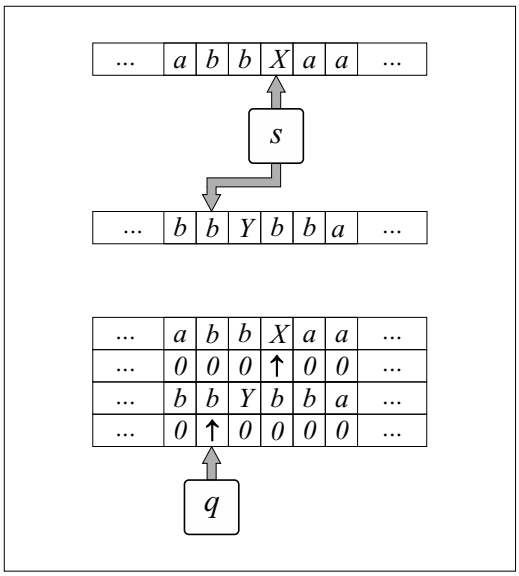
$$\pi : \Sigma \times V^n \ \rightarrow \ \Sigma \times V^n \times \{L, R\}^n ,$$

gdzie n jest liczbą taśm. Przyjrzyjmy się szczegółowo maszynie dwutaśmowej, a naszą analizę tego przypadku łatwo będzie później uogólnić na przypadek wielotaśmowy. Np. przejście $\pi(s, X, b) = (s', Z, b, L, R)$ interpretowane jest następująco: jeśli w stanie s symbolem skanowanym na pierwszej taśmie jest X, natomiast na drugiej jest to b, maszyna przechodzi do stanu s' zapisując Z w miejsce X na pierwszej taśmie i przesuwając jej głowicę w lewo, jednocześnie przesuwając drugą głowicę w prawo bez zmiany przeczytanej litery b. Ponownie przekonamy się, że maszyny wielotaśmowe w istocie nie stanowią żadnego uogólnienia klasycznych maszyn Turinga, funkcjonalnie są bowiem im równoważne.

To, że dwutaśmowa maszyna jest w stanie symulować standardowy automat Turinga nie budzi wątpliwości: może po prostu realizować jego działanie wykorzystując tylko jedną ze swoich taśm. Symulacja maszyny dwutaśmowej przez jednotaśmową jest nieco bardziej złożona. Przyjmiemy, że maszyna jednotaśmowa pracuje w trybie 4-ścieżkowym. Korzystamy tu z posiadanej już wiedzy, że maszyny z wielościeżkową taśmą są równoważne zwykłym. Idea symulacji zasadza się w sposobie, w jaki dwie taśmy reprezentowane są przy pomocy czterech ścieżek, p. Rys. 6.8. Ścieżki 1 i 3 w każdej chwili zawierają kopie zawartości obu taśm, natomiast ścieżki 2 i 4 zawierają w swoich niepustych klatkach np. cyfry 0 i pojedynczy znak $\uparrow$ służący jako wskaźnik pozycji głowicy, odpowiednio na taśmie pierwszej i trzeciej. Ruchy maszyny symulującej wykonywane są w cyklach złożonych z następujących kroków:

— ustawienie głowicy maszyny symulującej na początku użytkowego obszaru swojej taśmy w stanie reprezentującym aktualny stan (np. s) maszyny dwutaśmowej;

— odszukanie znaku $\uparrow$ na ścieżce 2;



Rys. 6.8: Reprezentacja konfiguracji dwutaśmowej maszyny Turinga na pojedynczej 4-ścieżkowej taśmie.

— zapamiętanie przy pomocy odpowiedniej zmiany stanu symbolu na ścieżce 1 w tej pozycji (np. X);

— odszukanie znacznika $\uparrow$ na ścieżce 4;

— na podstawie aktualnego stanu (przechowującego informację o znaku zlokalizowanym wcześniej na pierwszej ścieżce i wyjściowym stanie maszyny dwutaśmowej) oraz wartości symbolu w aktualnej pozycji na ścieżce 3 (np. b) wiadomo teraz jakie czynności w tej konfiguracji wykonuje maszyna dwutaśmowa: np. $\pi(s, X, b) = (s', Z, b, L, R)$;

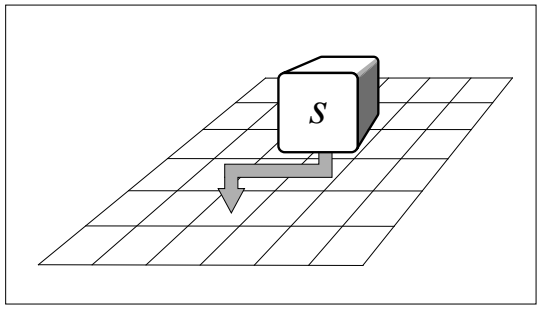
— w kolejnych krokach należy zmodyfikować odpowiednio zawartość ścieżek, a więc np. zamienić X na u na pierwszej, przesunąć znacznik $\uparrow$ o jedno miejsce w prawo na drugiej itd.

— przechowanie informacji o stanie s' i rozpoczęcie nowego cyklu.

Pomijamy tu oczywiście szereg technicznych szczegółów, mając nadzieję, że idea symulacji jest dostatecznie jasna. Istotne jest by zbiór stanów maszyny symulującej był dostatecznie liczny, by rozróżnić przy ich pomocy wszystkie możliwe kombinacje (s, x) maszyny dwutaśmowej.

Tak więc maszyny wielotaśmowe posiadają tę samą moc obliczeniową co klasyczne maszyny Turinga w następującym, precyzyjnym sensie: każdy algorytm, który może być zrealizowany przez wielotaśmowy automat, może być także wykonany przez klasyczną maszynę Turinga. Podobnie jak wcześniej, nie odnosimy się na tym etapie dyskusji do zagadnienia nakładu dodatkowej pracy obliczeniowej, niezbędnego do przeprowadzenia symulacji.

Wspomnijmy jeszcze o uogólnieniu maszyn Turinga polegającym na wprowadzeniu wielowymiarowej pamięci, np. tabelarycznej na płaszczyźnie, p. Rys. 6.9. Repertuar ruchów głowicy obejmuje tym razem także ruchy w pionie — w góre i w



Rys. 6.9: Maszyna Turinga z 2-wymiarową pamięcią.

dół, a funkcja przejścia ma postać

$$\pi : \Sigma \times V \to \Sigma \times V \times \{L, R, U, D\}\,.$$

Nie będziemy tym razem wchodzić w szczegóły dowodu równoważności takiego automatu z klasyczną maszyną Turinga. Poprzestaniemy na stwierdzeniu, że symulacja pamięci wielowymiarowej na pojedynczej, dwuścieżkowej taśmie możliwa jest, gdy wprowadzimy w niej całkowitoliczbowe kartezjańskie współrzędne i wykorzystamy jedną ze ścieżek taśmy symulatora do przechowania zawartości komórek tabelarycznej pamięci, a drugą do zapisania ich współrzędnych, np.

...	*a*				*b*				*c*				*c*	...
...	#	*1*	,	*2*	#	*-1*	,	*0*	#	*2*	,	*-1*	#	...

gdzie znak a znajduje się w komórce adresowanej przez $(1, 2)$, b w komórce $(-1, 0)$ itd.

Widzimy więc, że wszystkie sensowne uogólnienia maszyn Turinga okazują się być równoważne definicji standardowej. Definicja 6.1 podana przez nas na początku rozdziału jest bodaj najprostszym wariantem automatu Turinga. Przekonaliśmy się, że uzupełnianie tej definicji o dodatkowe elementy, jak np. wzbogacenie repertuaru ruchów głowicy lub zapis wielościeżkowy, nie rozszerza możliwości maszyny. Te dodatkowe elementy pozwalają przeważnie na łatwiejsze, bardziej zwięzłe opisy procesów realizowanych przez automaty Turinga, niż w przypadku użycia do tego celu formalizmu standardowego. Zwróćmy np. uwagę, o ile prościej opisać można proces dodawania 2 liczb binarnych używając do tego celu maszyny 3-taśmowej (2 taśmy na argumenty, jedna na utworzenie wyniku), niż realizować to samo zadanie na maszynie jednotaśmowej, por. Zad. 3 na końcu rozdziału.

3 Uniwersalna maszyna Turinga

Maszyna Turinga jest abstrakcyjnym matematycznym pojęciem, które powinno reprezentować intuicyjnie rozumianą "efektywną procedurę obliczeniową" lub równoważnie powinno być uniwersalnym, formalnym modelem komputera. Jednak każda maszyna Turinga określona jest przez definicję swej funkcji przejścia, a tym samym jest abstrakcyjnym urządzeniem przeznaczonym do rozwiązania jednego, konkretnego zadania albo — wypowiadając to samo inaczej — do rozpoznawania słów jednego,

ustalonego języka. Nie widać zatem czy, a jeśli tak, to w jaki sposób, można byłoby "programować" taki automat do wykonania wielu różnych zadań.

Opiszemy tzw. uniwersalną maszynę Turinga M_U jako programowalny, formalny model komputera. Idea jest następująca: dane wejściowe stanowić będzie opis funkcji przejścia π dowolnej ustalonej maszyny M oraz dowolne słowo $\omega \in A^*$. Maszyna uniwersalna przeczyta najpierw opis M, a następnie zasymuluje jej działanie na "danych" ω. Alternatywnie możemy myśleć, że M_U czyta jako dane produkcje pewnej gramatyki G oraz słowo ω, a następnie na tej podstawie weryfikuje czy $\omega \in L(G()$.

Ustalmy na początek jednolity sposób kodowania opisu maszyny Turinga. Załóżmy, że zbiór stanów reprezentowany jest zawsze jako $\Sigma = \{s_1, s_2, \ldots, s_n\}$, gdzie standardowo s_1 oznacza stan początkowy, a s_2 jest pojedynczym stanem końcowym.[2] Ponadto przyjmujemy, że $V = \{x_1, x_2, \ldots, x_m\}$, gdzie zawsze $x_1 = O$. Z tak przyjętą konwencją notacyjną, do jednoznacznego opisu maszyny Turinga wystarczy podanie wszystkich zdefiniowanych wartości jej funkcji przejścia, $\pi(s_i, x_j) = (s_p, x_q, R/L)$, które można zapisać jednoznacznie w postaci ciągu, np.

$$s_1, x_3, s_4, x_2, L;\ s_4, x_3, s_2, x_2, R;\ \ldots\ s_3, x_3, s_2, x_3, L\,.$$

Z takiego ciągu można odczytać w szczególności ile jest stanów i ile liter w V. Przyjmijmy teraz następujące kodowanie:

$$\begin{array}{lll} s_1 \to 1 & a_1 \to 1 & L \to 1 \\ s_2 \to 11 & a_2 \to 11 & R \to 11 \\ \vdots & \vdots & \\ s_n \to 1^n & a_m \to 1^m & \end{array} \tag{6.1}$$

oraz znaki "," i ";" jako 0. Przy jego użyciu ciąg przejść definiujący maszynę zapiszemy jako

$$1011101111011010111101110110110110\ldots \text{ itd.} \tag{6.2}$$

Wynika stąd, że każdą maszynę Turinga można opisać skończonym ciągiem binarnym, oraz że każdy ciąg binarny — jeśli tylko ma strukturę j.w. — można zdekodować jako opis maszyny Turinga.

Dla przejrzystości opiszemy maszynę uniwersalną M_U jako automat trójtaśmowy, w którym pierwsza taśma przechowuje binarny opis (6.2) symulowanej maszyny M, druga pełni rolę jej taśmy roboczej, także używając na niej binarnego kodowania (6.1) z zerami w roli separatorów liter, a trzecia przechowuje identyfikator aktualnego stanu M.

Ruchy maszyny uniwersalnej polegają kolejno na:

— porównaniu zawartości taśmy III (nr stanu maszyny M) z początkiem aktualnie skanowanego kodu przejścia π dla M na pierwszej taśmie; w przypadku niezgodności — przewinięciu pierwszej taśmy do początku następnego opisu przejścia w M (za kolejnym piątym znakiem 0) i powtórzeniu tego kroku;

[2]Można zawsze przyjąć, że maszyna Turinga ma tylko jeden stan końcowy. Jeśli jest ich wiele, wystarczy uzupełnić zbiór Σ o nowy, unikalny tym razem stan finalny s_f, oraz rozszerzyć definicję funkcji przejścia o wartości $\pi(s_k, x) = (s_f, x, R)$ dla wszystkich dotychczasowych stanów końcowych s_k i takich $x \in V$, dla których $\pi(s_k, x)$ było dotąd nieokreślone (a więc powodowało zatrzymanie maszyny w stanie s_k).

— porównaniu kodu znaku x_i z opisu przejścia na taśmie I z kodem aktualnie skanowanego znaku na taśmie II; w przypadku niezgodności należy wrócić do poprzedniego kroku;

— po znalezieniu w poprzednich krokach opisu przejścia dla konfiguracji stan–litera wg. zawartości taśm II i III, symulacyjnym wykonaniu czynności maszyny M na taśmach II i III wskazanych w opisie przejścia na taśmie I (zmiana kodu litery i ruch głowicy na taśmie II i zmiana identyfikatora stanu ta taśmie III);

— powtórzeniu całego cyklu lub — w przypadku nieodnalezienia stosownej reguły przejścia na taśmie I — przejście do swojego stanu finalnego, z sygnalizacją czy ostatnim osiągniętym stanem symulowanej maszyny był akceptujący stan s_2.

Konstrukcja uniwersalnej maszyny Turinga ma przede wszystkim znaczenie teoretyczne. Jest ona wykorzystywana w dowodach niektórych twierdzeń o problemach nierozstrzygalnych.

Jako rezultat uboczny przeprowadzonej powyżej dyskusji otrzymujemy następujący, interesujący lemat.

LEMAT 6.1 *Zbiór wszystkich maszyn Turinga jest przeliczalny.*

Jest to oczywiste na podstawie faktu, że zbiór *skończonych* ciągów binarnych jest przeliczalny, a wśród nich zawarte są zakodowane opisy wszystkich maszyn Turinga. Przy okazji kodowanie to indukuje pewną numerację maszyn Turinga

$$M_1,\ M_2,\ M_3,\ \ldots \tag{6.3}$$

wg. wzrastających wartości liczbowych kodów tych maszyn. Do tej numeracji będziemy się jeszcze odwoływać w dalszej części rozdziału.

VI.2 Hipoteza Churcha-Turinga i języki rekursywnie przeliczalne

Powiążemy obecnie automaty Turinga z językami klasy $\mathcal{L}_0$. Nadamy także bardziej precyzyjne znaczenie pojęciom takim, jak obliczalność, rozstrzygalność czy nierozstrzygalność.

1 Równoważność maszyn Turinga i gramatyk typu 0

Naszym celem jest udowodnienie następującego twierdzenia:

TWIERDZENIE 6.2 *Klasa języków akceptowanych przez maszyny Turinga jest równa klasie $\mathcal{L}_0$ języków generowanych przez gramatyki typu 0.*

DOWÓD. Załóżmy najpierw, że $L \in \mathcal{L}_0$ oraz że G jest gramatyką generującą L. Istnieje deterministyczna maszyna w standardowej postaci, która akceptuje wszystkie

słowa z L, a zarazem jedynie te słowa. Posłużymy się tu jednak niedeterministycznym modelem maszyny dwutaśmowej, by uniknąć technicznych komplikacji, zaciemniających jedynie samą ideę dowodu. Wiemy już, że niedeterminizm i użycie wielu taśm nie definiują nowych klas automatów, bowiem każda z tych własności może być symulowana przez jednotaśmową, deterministyczną maszynę Turinga.

Maszyna Turinga M na jednej ze swoich taśm przechowuje zbiór produkcji gramatyki G, np. w postaci ciągu $\alpha_1 \to \beta_1 \# \ldots \# \alpha_m \to \beta_m$. Taśma II przechowuje testowane słowo ω. Maszyna najpierw przesuwa głowicę pierwszej taśmy, wybierając niedeterministycznie jedną z produkcji $\alpha_i \to \beta_i$. Podobnie niedeterministycznie wybierana jest pozycja głowicy nad słowem ω na taśmie II. Jeśli poczynając od tej pozycji kolejne litery w ω zgadzają się z prawą stroną wybranej produkcji β_i, wówczas maszyna przekształca słowo $\omega = \gamma\beta_i\eta$ w napis $\omega_1 = \gamma\alpha_i\eta$. Jeśli po pewnej liczbie takich cykli zawartość taśmy II zamieni się w $\omega_n = S$, automat przechodzi do swojego stanu końcowego, z którego nie ma dalszych ruchów. Zgodnie z definicją maszyny niedeterministycznej, napis wejściowy ω jest akceptowany, jeśli istnieje sekwencja ruchów maszyny prowadząca do jej zatrzymania w stanie finalnym. Jest więc oczywiste, że M akceptuje ω wtedy i tylko wtedy, gdy $S \Rightarrow_G^* \omega$, a więc gdy $\omega \in L(G)$.

Z kolei niech $L = L(M)$ będzie językiem akceptowanym przez maszynę Turuinga

$$M = (A, V, O, \Sigma, q_0, S_F, \{L, R\}, \pi)\,.$$

Utwórzmy gramatykę $G = (A, U, S_0, \Pi)$ typu 0, w której zbiór nieterminali utożsamiamy z

$$U \;=\; (A \cup \{\lambda\}) \times V \;\cup\; \Sigma \;\cup\; \{S_0, S_1, S_2\}\,,$$

natomiast Π składa się z następujących produkcji:

1. $S_0 \to q_0 S_1$
2. $S_1 \to (a, a)S_1 \,|\, S_2$, dla wszystkich $a \in A$
3. $S_2 \to (\lambda, O)S_2 \,|\, \lambda$
4. $q(a, X) \to (a, Y)r$, dla wszystkich $a \in A \cup \{\lambda\}$ oraz $X, Y \in V$, $q, r \in \Sigma$, dla których $\pi(q, X) = (r, Y, R)$
5. $(b, Z)q(a, X) \to r(b, Z)(a, Y)$, dla wszystkich $a, b \in A \cup \{\lambda\}$ oraz $X, Y, Z \in V$, $q, r \in \Sigma$, takich że $\pi(q, X) = (r, Y, L)$
6. $(a, X)q \to qaq$, $q(a, X) \to qaq$ i $q \to \lambda$, dla $a \in A \cup \{\lambda\}$, $q \in \Sigma$ i $X \in V$.

Zauważmy teraz, że produkcje 1–3 generują wyprowadzenia

$$S_0 \Rightarrow^* q_0(a_1, a_1) \ldots (a_n, a_n)(\lambda, B)^k\,,$$

gdzie $a_i \in A$ oraz $k \geqslant 0$. Wyprowadzenie to można dalej kontynuować używając produkcji 4 lub 5. Produkcje te symulują bezpośrednio ruchy maszyny M, przy czym zawartość jej taśmy kodowana jest przez drugi element w "złożonych" nieterminalach postaci (a, x). Proces wyprowadzania kończy się, gdy w formie zdaniowej pojawi się jakikolwiek stan końcowy $q \in S_F$. Wówczas wykorzystywane są reguły nr 6, w

wyniku czego otrzymujemy słowo $\omega = a_1 a_2 \dots a_n$. Łatwo teraz sprawdzić, że $\omega \in L(G)$ wtedy i tylko wtedy, gdy $\omega \in L(M)$. □

Przedstawiony w dowodzie opis działania maszyny Turinga, realizującej w trybie bottom-up redukcje wg. reguł gramatyki, można odwrócić w taki sposób, by symulowany był pełny przegląd drzewa wyprowadzeń, a więc proces analizy top-down. Bardziej naturalna jest deterministyczna wersja tego procesu. Załóżmy, że używamy przy tym 3 taśm. Na pierwszej, jak wyżej, przechowywane są produkcje G. Na taśmie II zapisujemy na początku symbol S. Maszyna przegląda produkcje w poszukiwaniu tych z lewą stroną S, generując wszystkie formy zdaniowe osiągalne z S w jednym kroku, a więc pierwsze piętro drzewa wyprowadzeń. Używa przy tym dodatkowego znaku # dla separacji poszczególnych form. W następnym cyklu w podobny sposób generowane są wszystkie formy drugiego piętra drzewa itd. W przypadku wygenerowania napisu terminalnego, jest on kopiowany na taśmę nr III. W ten sposób taśma III zapełnia się stopniowo listą słów $\alpha_1 \# \alpha_2 \# \dots$, $\alpha_i \in L$.

Przyjmijmy następującą definicję.

DEFINICJA 6.3 *Mówimy, że język L jest rekursywnie przeliczalny jeśli istnieje algorytm wypisujący systematycznie wszystkie jego słowa. Klasę tych języków oznaczamy symbolem $\mathcal{L}_{\mathrm{Rp}}$.*

Widzimy więc, że języki akceptowane przez maszyny Turinga, a więc wszystkie języki z $\mathcal{L}_0$ są rekursywnie przeliczalne,

$$\mathcal{L}_0 \subseteq \mathcal{L}_{\mathrm{Rp}}\,.$$

W następnym podrozdziale uzasadnimy także inkluzję odwrotną.

2 Hipoteza Churcha-Turinga. Hierarchia języków formalnych.

Wykazaliśmy, że maszyny Turinga stanowią klasę automatów o dokładnie takim samym "potencjale obliczeniowym" jak gramatyki Chomsky'ego typu 0. Nie jest to jedyna równoważność tego typu znana w matematyce. Istnieją inne formalne modele efektywnie wykonalnych procesów obliczeniowych, które mimo faktu, iż wyrosły z zupełnie odrębnych badań o często odległej tematyce, wszystkie z czasem okazały się być równoważne standardowym automatom Turinga. Wymieńmy tylko niektóre z nich, odsyłając zainteresowanego czytelnika do literatury: funkcje rekurencyjne, systemy Posta, układy Markowa, gramatyki macierzowe lub systemy Lindenmayera, [7, 10, 11]. Fakty te stanowią racjonalną bazę dla tzw. Hipotezy Churcha–Turinga.

Hipoteza Churcha–Turinga. *Klasa wszystkich efektywnie wykonalnych procedur jest tożsama z klasą maszyn Turinga.*

To, że maszyny Turinga są przykładami efektywnie wykonalnych procedur nie budzi naszej wątpliwości, natomiast nietrywialna część tezy Churcha orzeka zachodzenie także relacji odwrotnej: *każdy* bez wyjątku efektywnie wykonalny proces da się zrealizować w postaci maszyny Turinga. Hipotezy tej nie można udowodnić, bowiem odwołuje się ona do intuicyjnego i nie do końca precyzyjnego pojęcia "efektywnego procesu". Pierwszym krokiem do dowodu musiałoby być więc formalne zdefiniowanie takiego procesu. Tu natychmiast pojawia się nieusuwalny problem: czy

oto nowa formalna definicja rzeczywiście wiernie odpowiada pojęciu efektywnej procedury? Zatem próba udowodnienia hipotezy Churcha kreuje natychmiast nową tezę dokładnie tego samego rodzaju. W świetle naszych rozważań hipoteza Churcha jest na tyle dobrze umotywowana, że przyjęcie jej nie budzi dziś zastrzeżeń.

W podobnym duchu można sensownie postawić pytanie o relację między maszyną Turinga a współczesnym komputerem. Skoro opis automatu jest z definicji *skończony*, zawiera bowiem skończenie wiele stanów, symboli i reguł przejścia, nietrudno wyobrazić sobie program, napisany w języku C lub dowolnym innym, symulujący działanie deterministycznej maszyny Turinga. Jedyna wątpliwość może pochodzić stąd, że maszyna Turinga dysponuje nieograniczoną pamięcią sekwencyjną, podczas gdy pamięć realnego komputera jest skończona. W tym sensie jest on w stanie emulować co najwyżej skończone automaty, a więc jego zdolności rozpoznawania języków ściśle rzecz biorąc ograniczone do języków regularnych. Można jednak wyobrazić sobie, przynajmniej hipotetyczne, rozszerzanie pamięci komputera o kolejne terabajty w miarę potrzeb. W tym sensie komputer jest w stanie symulować działanie maszyny Turinga, jeśli tylko jesteśmy w stanie udostępnić mu "dostatecznie dużą pamięć".

Odwrotnie, czy działanie komputera może być wiernie symulowane przez maszynę Turinga? Zauważmy w pierwszej kolejności, że architektura wszystkich komputerów, a ściślej ich procesorów, udostępnia skończenie wiele elementarnych rozkazów operujących na skończonej liczbie słów 16-, 32- lub 64-bitowych. Działanie rozkazów, w tym także tych operujących zmiennoprzecinkowymi rejestrami, może być opisane przez rozbudowane sieci logiczne z bramkami operującymi na poziomie pojedynczych bitów. Cały repertuar rozkazów dowolnego procesora można więc zrealizować na maszynie Turinga. Z kolei symulacja adresowalnej pamięci może być wygodnie przeprowadzona na dwuścieżkowej taśmie, gdzie jedna ze ścieżek przechowuje numerycznie zapisane adresy słów, a druga równolegle przechowuje zapisane w niej dane. Upraszczając sobie zadanie, możemy przyjąć wielotaśmową architekturę maszyny Turinga symulującej komputer. Jedna z taśm (jak wyżej) reprezentuje pamięć i przechowywać może binarny kod programu do wykonania, a więc listę instrukcji procesora składającą się na ten program. Dalsze taśmy mogą reprezentować osobno rejestry komputera: licznik rozkazów, rejestr adresowy, rejestry arytmetyczne itp. Jest ich zawsze skończenie wiele, niezależnie od rodzaju procesora. Kolejne dwie taśmy symulować będą pliki wejściowy i wyjściowy. W końcu ostatnia z taśm realizuje funkcję pamięci roboczej. W takiej konfiguracji można już stosunkowo łatwo wyobrazić sobie cykl pracy maszyny Turinga symulującej komputer.

Wracając do relacji między językami klasy $\mathcal{L}_0$ a rekursywnie przeliczalnymi, widzimy, że jedną z konsekwencji hipotezy Churcha jest równość

$$\mathcal{L}_{\mathrm{Rp}} = \mathcal{L}_0 \,. \tag{6.4}$$

Ponadto łatwo przekonać się, że każdy język rekursywny L (p. Definicja 5.3) jest rekursywnie przeliczalny. Wystarczy bowiem generować kolejne napisy α z A^* w porządku leksykograficznym, a następnie sprawdzać czy $\alpha \in L$ oraz, w przypadku twierdzącej odpowiedzi, wypisywać je. Relacja odwrotna nie zachodzi: o ile dla języków rekursywnych zawsze potrafimy zdecydować czy $\alpha \in L$ czy też $\alpha \notin L$, to gdy L jest jedynie rekursywnie przeliczalny, stwierdzenie, że dane słowo $\alpha \notin L$ na ogół

nie jest możliwe: przeglądanie nieskończonej listy słów języka L nie daje podstaw do odrzucenia α w jakimkolwiek skończonym czasie. Jest to jednak tylko argument intuicyjny. Przy pewnym dodatkowym nakładzie pracy przekonamy się, że zgodnie z tą intuicją inkluzja $\mathcal{L}_{\mathrm{Rec}} \subseteq \mathcal{L}_{\mathrm{Rp}}$ jest rzeczywiście właściwa, najpierw jednak przeanalizujemy jak relacja miedzy $\mathcal{L}_{\mathrm{Rec}}$ a $\mathcal{L}_{\mathrm{Rp}}$ przejawia się w sposobie działania maszyny Turinga, możliwe są bowiem następujące trzy scenariusze:

a) maszyna zatrzymuje się w pewnym stanie końcowym $s \in S_F$ z powodu nieokreśloności funkcji przejścia dla aktualnej konfiguracji (s, a);

b) jak wyżej, z tą różnicą, że $s \notin S_F$;

c) maszyna Turinga nie zatrzymuje się, działając w nieskończonej pętli.

O pierwszych dwóch wariantach mówimy, że maszyna osiąga *stan stopu*. Jak już wiemy, w przypadku a) mówimy o akceptacji słowa wejściowego, w przypadku b) — o jego odrzuceniu. Nie ma jednak gwarancji, nawet w przypadku maszyny niedeterministycznej, że dla każdego napisu wejściowego istnieje ciąg ruchów osiągający stanu stopu. Może się zdarzyć, że dla pewnych napisów realizowany będzie jedynie scenariusz c). Słowa te oczywiście nie są kwalifikowane jako elementy akceptowanego przez automat języka, zgodnie z Definicją 6.2.

Gdy relacja przejścia maszyny ma taką strukturę, że dla dowolnego napisu wejściowego stan stopu jest osiągany, akceptowany język jest językiem rekursywnym, bo pytanie czy $\alpha \in L$ jest rozstrzygalne. Możemy na tej podstawie uściślić definicję klasy $\mathcal{L}_{\mathrm{Rec}}$:

DEFINICJA 6.4 *Język L określamy jako rekursywny jeśli istnieje akceptująca go maszyna Turinga, która osiąga stan stopu dla każdego napisu wejściowego.*

Okazuje się, że zachowanie typu c) jest w ogólnym przypadku nieusuwalną cechą maszyn Turinga, przez co klasa akceptowanych przez nie języków jest bardziej pojemna niż $\mathcal{L}_{\mathrm{Rec}}$. Przekonamy się jednak, że skonstruowanie choć jednego przykładu języka z $\mathcal{L}_{\mathrm{Rp}} - \mathcal{L}_{\mathrm{Rec}}$ nie jest zadaniem prostym.

Rozpoczniemy od zbudowania przykładu języka, który nie jest nawet rekursywnie przeliczalny. Języków takich jest w istocie bardzo wiele: sokro zbiór napisów A^* nad dowolnym alfabetem jest przeliczalny, zgodnie z Twierdzeniem 1.1 z Rozdziału I jego zbiór potęgowy, a więc zbiór wszystkich języków nad A jest nieprzeliczalny. Z drugiej strony a podstawie Lematu 6.1 i równości (6.4) wiemy, że języków rekursywnie przeliczalnych jest przeliczalnie wiele. A więc olbrzymia większość podzbiorów A^* to języki, które nie są rekursywnie przeliczalne, mamy zatem bez liku potencjalnych przykładów. Problem z charakteryzacją choćby jednego takiego języka bierze się stąd, że skoro nie jest on rekursywnie przeliczalny, tym samym nie można opisać go efektywna procedurą. A więc w samym postawieniu problemu tkwi już logiczny chochlik. Rozwiązanie jest jednak możliwe, jeśli wybierzemy nieco okrężną drogą.

TWIERDZENIE 6.3 *Isinieje język $L \in \mathcal{L}_{\mathrm{Rp}}$, którego dopełnienie $L^c \notin \mathcal{L}_{\mathrm{Rp}}$.*

DOWÓD. Ponownie musimy odwołać się do metody przekątniowej. Rozważmy wszystkie maszyny Turinga akceptujące języki nad alfabetem jednoliterowym $A = \{a\}$. Uporządkowanie automatów Turinga (6.3) indukuje unikalną numerację rozważanej

tu rodziny automatów, M_1, M_2, itd. Każdy z języków $L(M_i)$ jest rekursywnie przeliczalny, a także każdy rekursywnie przeliczalny język nad jednoliterowym alfabetem jest ujęty w tym ciągu.

Utwórzmy teraz nowy język wg. przepisu

$$L \;=\; \{a^k :\;\; a^k \in L(M_k)\}\,.$$

Przekonajmy się, że jest on rekursywnie przeliczalny. Mając dane k maszyna Turinga generuje kolejne ciągi binarne, sprawdzając czy kodują one poprawnie automat nad jednoliterowym alfabetem zgodnie z (6.2). Po znalezieniu k-tego z nich, kodującego automat M_k, maszyna przechowuje go na swojej taśmie i uzupełnia napisem a^k, po czym przechodzi do stanu, który rozpoczyna jej działanie jako uniwersalnej maszyny Turinga M_U. Akceptuje ona słowo a^k jeśli tylko $a^k \in L(M_k)$, a więc w efekcie akceptuje język L. Jest on zatem językiem klasy $\mathcal{L}_{\mathrm{Rp}}$.

Pokażemy następnie, że język L^c nie jest rekursywnie przeliczalny. Gdyby przeciwnie był on elementem $\mathcal{L}_{\mathrm{Rp}}$, jako język nad jednoliterowym alfabetem musiałby się znaleźć na naszej liście $L(M_1)$, $L(M_2)$, ..., a więc istniałoby n takie, że $L^c = L(M_n)$. Zapytajmy o słowo a^n: czy jest ono elementem L^c? Jeśli tak, to $a^n \in L(M_n)$, co jednak oznacza, że $a^n \in L$, a więc, że $a^n \notin L^c$. Doszliśmy do sprzeczności. Załóżmy na odwrót, że $a^n \notin L^c$, a więc $a^n \notin L(M_n)$, czyli $a^n \notin L$. To jednak oznacza, że $a^n \in L^c$ — ponownie sprzeczność. Przypuszczenie, że L^c jest rekursywnie przeliczalny musiało być zatem fałszywe. □

Przejdźmy do dowodu drugiego, tym razem prostego faktu.

LEMAT 6.2 *Jeśli zarówno L jak i L^c są językami rekursywnie przeliczalnymi, są one automatycznie rekursywne. Ponadto jeśli L jest rekursywny, wówczas na pewno L^c jest również rekursywny, a więc obydwa są rekursywnie przeliczalne.*

Dowód. Niech $L, L^c \in \mathcal{L}_{\mathrm{Rp}}$. Istnieją zatem maszyny Turinga M dla L i $\bar{M}$ dla L^c tworzące kompletne listy ich słów. Niech $\alpha \in A^*$. Pozwólmy maszynie M wygenerować jedno słowo z L i podobnie maszynie $\bar{M}$ — jedno słowo z L^c. Jeśli jest wśród nich α, dowiedzieliśmy się czy $\alpha \in L$, czy też $\alpha \in L^c$. W przeciwnym razie powtarzamy cykl generując kolejne słowa z L na przemian z tymi z L^c. Ponieważ jednak $\alpha \in L$ albo $\alpha \in L^c$, α musi pojawić się w skończonym czasie na jednej z list. Zatem problem czy $\alpha \in L$ jest rozstrzygalny. Ponieważ rezultat ten można odnieść zarówno do L jak i do L^c, obydwa języki są rekursywne.

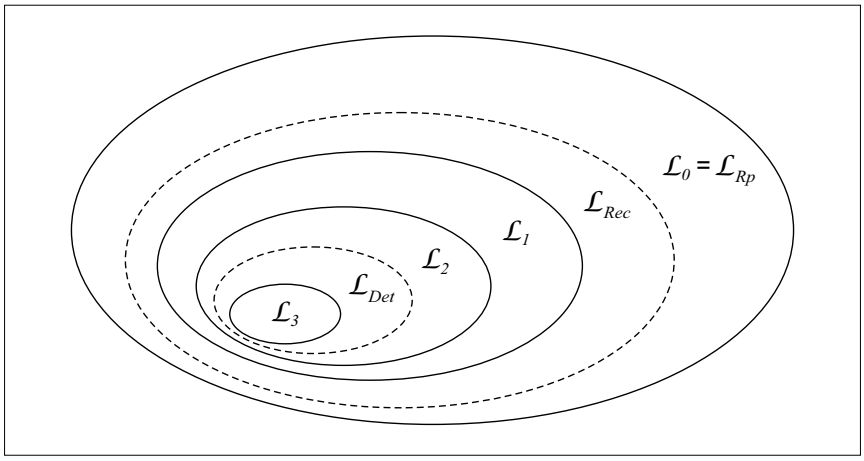
Jeśli zaś wiemy, że $L \in \mathcal{L}_{\mathrm{Rec}}$, wówczas algorytm weryfikujący czy dowolne $\alpha \in L$ jest także dobry dla L^c, a więc L^c też jest rekursywny. □

Możemy teraz sformułować oczekiwany rezultat jako wniosek z powyższych rozważań.

Wniosek.

$$\mathcal{L}_{\mathrm{Rp}} \;-\; \mathcal{L}_{\mathrm{Rec}} \;\neq\; \emptyset\,.$$

Istotnie, w Twierdzeniu 6.3 zbudowaliśmy język $L \in \mathcal{L}_{\mathrm{Rp}}$, którego dopełnienie nie leży w tej klasie. Zatem na podstawie ostatniego lematu L nie może być rekursywny.



Rys. 6.10: Hierarchia klas języków. Ciągłą linią oznaczyliśmy standardowe klasy języków Chomsky'ego.

Podsumujmy nasze wyniki w formie łatwego do zapamiętania diagramu obrazującego relacje między klasami języków, Rys. 6.10.

Zakończymy ten podrozdział interesującą charakteryzacją algebraiczną klasy $\mathcal{L}_{\mathrm{Rp}}$, p. np. [10].

TWIERDZENIE 6.4 *(Ginsburg et al. 1967) Każdy język rekursywnie przeliczalny L jest homomorficznym obrazem przekroju dwóch języków bezkontekstowych L_1 i L_2.*

$$L \;=\; h(L_1 \cap L_2)\,.$$

3 Zamkniętość klasy $\mathcal{L}_0$

Z Twierdzenia 6.3 i Lematu 6.2 wysnuć można bezpośrednie wnioski, że klasa $\mathcal{L}_{\mathrm{Rec}}$ jest zamknięta ze względu na operację mnogościowego dopełnienia języka, natomiast $\mathcal{L}_{\mathrm{Rp}}$ nie. Dwa twierdzenia, które przytoczymy bez dowodów, podsumowują własności zamkniętości klas $\mathcal{L}_{\mathrm{Rec}}$ i $\mathcal{L}_{\mathrm{Rp}}$ ze względu na rozmaite operacje.

TWIERDZENIE 6.5 *Klasa $\mathcal{L}_{\mathrm{Rp}}$ jest zamknięta ze względu na następujące operacje: suma, przekrój, konkatenacja, domknięcie konkatenacyjne, odbicie lustrzane, przekrój z językiem regularnym, dowolne podstawienie (w tym dowolny homomorfizm). Nie jest natomiast zamknięta ze względu na dopełnienie, a więc także ze względu na różnicę zwykłą i symetryczną.*

TWIERDZENIE 6.6 *Klasa $\mathcal{L}_{\mathrm{Rec}}$ jest zamknięta ze względu na następujące operacje: suma, przekrój, dopełnienie, konkatenacja, domknięcie konkatenacyjne, odbicie lustrzane, przekrój z językiem regularnym, λ-wolne podstawienie (w tym λ-wolny homomorfizm). Nie jest natomiast zamknięta ze względu na dowolne podstawienia i homomorfizmy.*

4 Granice obliczalności — problemy nierozstrzygalne

Jesteśmy obecnie wyposażeni w teoretyczne narzędzia pozwalające na bardziej precyzyjną analizę pojęć obliczalności i rozstrzygalności. We wcześniejszych rozdziałach, gdy mówiliśmy, że pewien problem jest rozstrzygalny lub nierozstrzygalny dla języków określonej klasy, odwoływaliśmy się do istnienia bądź nieistnienia ogólnej procedury, która np. na podstawie opisu gramatyki stwierdza jednoznacznie czy generowany przez nią język posiada pewne cechy. O ile dowodem istnienia procedury jest po prostu podanie jej opisu, to uzasadnienie *nieistnienia* ogólnej metody jest z reguły o wiele bardziej wyrafinowane logicznie. Okazuje się, że pojęcie maszyny Turinga pozwala na precyzyjne zdefiniowanie niezbędnych pojęć i tworzy formalna bazę do prowadzenia ścisłych dowodów nierozstrzygalności.

DEFINICJA 6.5 *Niech $f : X \to N$ będzie funkcją określoną na pewnym podzbiorze $X \subseteq N$. Mówimy, że f jest obliczalna, jeśli istnieje maszyna Turinga wyznaczająca wartości f dla wszystkich $n \in X$.*

Słowo wyjaśnienia: jeśli mówimy, że maszyna M wyznacza wartości f, mamy na myśli, że osiąga ona stan stopu dla wszystkich danych n z dziedziny f. Końcowa zawartość taśmy jest wówczas interpretowana jako wynik obliczenia $f(n)$. Zwróćmy uwagę, że dziedzina funkcji X jest integralną częścią pojęcia obliczalności: funkcja jest zawsze obliczalna *na pewnej dziedzinie*, na innej może nie być obliczalna.

Przez opis gramatyki rozumiemy słowo nad pewnym ustalonym alfabetem kodujące wszystkie jej produkcje, np. w postaci

$$\alpha_1 \to \beta_1 \# \alpha_2 \to \beta_2 \# \ \ldots \ \alpha_k \to \beta_k \,,$$

przy czym przyjęta jest standardowa konwencja dotycząca oznaczania nieterminali, por. np. dowód Twierdzenia 5.7. Słowo tej postaci kodujące gramatykę G oznaczymy krótko symbolem $<G>$.

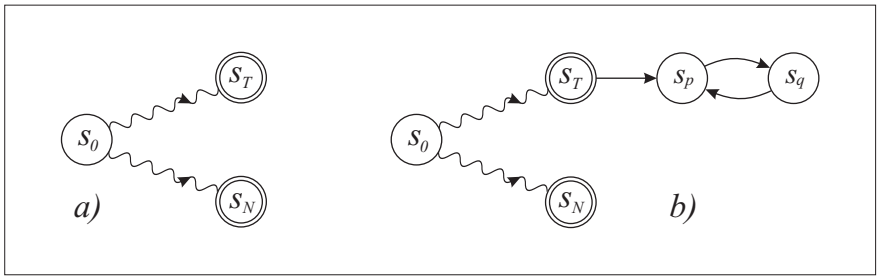
Niech P oznacza pewną własność, którą mogą posiadać lub nie języki określonej klasy $\mathcal{L}$ (generowane przez gramatyki kategorii $\mathcal{G}$, np. kontekstowe, liniowe itp.). Mówimy wówczas, że własność P jest *trywialna* w klasie $\mathcal{L}$, jeśli P jest prawdziwa dla wszystkich bez wyjątku języków w $\mathcal{L}$ albo fałszywa dla nich wszystkich. W przeciwnym wypadku mówimy, że P jest *nietrywialna* w $\mathcal{L}$. Z własnością P możemy związać język

$$L_P \ = \ \{<G>: \ G \in \mathcal{G} \quad \text{i} \quad L(G) \ \text{ posiada własność } \ P\} \,. \tag{6.5}$$

DEFINICJA 6.6 *Mówimy, że własność P jest rozstrzygalna w klasie języków $\mathcal{L}$, jeśli język L_P jest rekursywny.*

Innymi słowy własność P jest rozstrzygalna w $\mathcal{L}$, gdy istnieje maszyna Turinga, która działając na dowolnym słowie $<G>$ dla $G \in \mathcal{G}$ osiąga stan stopu i rozpoznaje czy $<G> \in L_P$, a więc czy $L(G)$ posiada czy też nie własność P. Nierozstrzygalność jest równoważna z nierekursywnością języka L_P. Może się więc np. zdarzyć, że dla pewnego słowa $< G >$ maszyna Turinga badająca własność P nie osiągnie stanu stopu.

Okazuje się, że w zasadzie wszystkie sensowne własności są w klasie $\mathcal{L}_0$ nierozstrzygalne. Orzeka o tym następujące twierdzenie, [4, 10]:



Rys. 6.11: a) Struktura stanów maszyny H_S. b) Rozszerzenie H° maszyny H_S.

TWIERDZENIE 6.7 (Rice, 1959) *Każda nietrywialna własność P języków klasy $\mathcal{L}_0$ jest nierozstrzygalna.*

Podamy teraz klasyczny przykład nierozstrzygalnego zagadnienia. Posłużymy się kodowaniem binarnym opisanym w podrozdziale VI.1.3.

DEFINICJA 6.7 *Niech ω_M oznacza słowo binarne opisujące maszynę Turinga $M = (A, V, O, \Sigma, s_0, S_F, \{L, R\}, \pi)$ oraz niech ω będzie binarnym kodem pewnego słowa nad alfabetem A maszyny M. Określmy język*

$$L_{\text{Stop}} = \{\omega_M \# \omega : \text{ maszyna Turinga } M \text{ osiąga stan stopu na danych } \omega\}.$$

Problem stopu polega na stwierdzeniu dla dowolnego napisu postaci $\omega_M\#\omega$ czy należy ono do języka L_{Stop}.

Mamy więc własność P postaci "M osiąga stan stopu na danych ω" w klasie $\mathcal{L}_0$ tym razem opisanej nie przez gramatyki, lecz przez maszyny Turinga M (słowa ω_M pełnią więc rolę słów $<G>$). Domyślamy się, że jest to własność nietrywialna w $\mathcal{L}_0$, a więc zgodnie z twierdzeniem Rice'a problem stopu byłby nierozstrzygalny. Podamy jednak bezpośredni dowód tego faktu.[3]

TWIERDZENIE 6.8 *Problem stopu jest nierozstrzygalny.*

Dowód. Załóżmy przeciwnie, że $L_{\text{Stop}} \in \mathcal{L}_{\text{Rec}}$, istnieje więc maszyna Turinga H_S osiągająca stan stopu dla dowolnych danych wejściowych $\omega_M\#\omega$. Bez zmniejszenia ogólności załóżmy, że maszyna H_S zatrzymuje się w jednym z dwóch stanów: s_T gdy $\omega_M\#\omega \in L_{\text{Stop}}$ lub s_N w przeciwnym wypadku. Sytuację ilustruje diagram a) na Rys. 6.11. Zmodyfikujemy strukturę maszyny H_S dodając do niej najpierw 2 nowe stany s_p i s_q oraz następujące przejścia:

$$\pi(s_T, x) = (s_p, x, R), \qquad \pi(s_p, x) = (s_q, x, L), \qquad \pi(s_q, x) = (s_p, x, R)$$

dla wszystkich $x \in V$. W ten sposób nowa maszyna H° po osiągnięciu stanu s_T wchodzi w nieskończoną pętlę $s_p \circlearrowleft s_q$, Rys. 6.11 b). Utwórzmy następnie maszynę K, której działanie polega na tym, że powiela ona swoje dane wejściowe ω, generując na taśmie napis postaci $\omega\#\omega$, po czym przesuwa głowicę na jego początek i

[3] W rzeczywistości dowód twierdzenia Rice'a korzysta z faktu, że problem stopu jest nierozstrzygalny. Dlatego niezależny dowód dla zagadnienia stopu jest niezbędny.

zatrzymuje się w stanie finalnym. Ostatni krok naszej konstrukcji polega na tym, że łączymy maszyny K i H° kaskadowo, to jest utożsamiamy stan finalny K ze stanem startowym s_0 maszyny H°. Otrzymaną w ten sposób maszynę oznaczamy symbolem H. Jak każda maszyna Turinga, H może być przedstawiona w postaci binarnego kodu ω_H.

Przeanalizujmy działanie maszyny H na danych wejściowych ω_H. W pierwszej fazie ω_H jest przekształcane w słowo $\omega_H\#\omega_H$, które w fazie drugiej staje się napisem wejściowym dla bloku H°. Zgodnie z określeniem H°, H wchodzi nieskończoną w pętlę $s_p \circlearrowleft s_q$ jeśli maszyna opisana przez ω_H, a więc H (!), osiąga stan stopu na danych ω_H. Jest to oczywiście sprzeczność. Alternatywnie H zatrzymuje się w stanie s_N jeśli maszyna kodowana przez ω_H wchodzi w pętlę na danych ω_H. Ponownie sprzeczność. Stąd wnosimy, że język L_{Stop} nie może być rekursywny, a zatem problem stopu jest nierozstrzygalny. □

Pytanie o rozstrzygalność problemu stopu jest w gruncie rzeczy równoważne pytaniu o to czy $\mathcal{L}_{\mathrm{Rec}} = \mathcal{L}_{\mathrm{Rp}}$. Z jednej strony bowiem, gdyby każdy język akceptowany przez maszyny Turinga był rekursywny, problem stopu byłby trywialny: *każda* maszyna osiąga stan stopu na *każdych* danych. Z drugiej strony gdyby istniała maszyna H rozstrzygająca dla dowolnych danych problem stopu, pytanie czy $\alpha \in L(M)$ dla dowolnej maszyny Turinga M byłoby rozstrzygalne. Wystarczyłoby uruchomić H na danych $\omega_M\#\alpha$ — jeśli odpowiedzią H byłoby "M nie zatrzymuje się dla danych α", wówczas z pewnością $\alpha \notin L(M)$, jeśli zaś odpowiedź byłaby przeciwna, wystarczyłoby uruchomić M na α, a po jej zatrzymaniu się sprawdzić czy osiągnęła stan finalny czy też nie. Tak więc $L(M) \in \mathcal{L}_{\mathrm{Rec}}$, a wobec dowolności M oznaczałoby to równość $\mathcal{L}_{\mathrm{Rec}} = \mathcal{L}_{\mathrm{Rp}}$.

Powyższy argument wiążący ze sobą problem stopu i problem należenia słowa do języka z $\mathcal{L}_0$ jest przykładem bardzo ważnej techniki redukcji używanej w dowodach nierozstrzygalności. Problem rozstrzygalności zagadnienia P redukuje się do pytania o rozstrzygalność innego zagadnienia Q, jeśli hipotetyczny algorytm (maszyna Turinga) rozstrzygający zagadnienie Q może być użyty do rozstrzygnięcia P,

$$\text{rozstrzygalność } Q \;\Rightarrow\; \text{rozstrzygalność } P\,.$$

Jeśli jednak wiemy już, że P jest nierozstrzygalny, oznacza to, że Q też nie może być rozstrzygalny. Tak więc, jeśli np. potrafimy wskazać konstruktywny sposób redukcji zagadnienia stopu do innego problemu, ten drugi problem nie może być rozstrzygalny. W gruncie rzeczy ścisłe dowody wszystkich wymienionych w tym skrypcie zagadnień nierozstrzygalnych przeprowadza się przez wskazanie jak można zredukować do nich jeden ze znanych problemów nierozstrzygalnych. Pamiętajmy, że choć nierozstrzygalność w klasie $\mathcal{L}_0$ jest wyczerpująco opisana przez twierdzenie Rice'a, nie da się go zastosować w odniesieniu do innych klas języków i np. nierozstrzygalność pytania o to czy dany język bezkontekstowy jest identyczny z A^* wymaga własnego dowodu. W klasie $\mathcal{L}_0$ to samo pytanie jest oczywiście nierozstrzygalne, ponieważ własność bycia językiem tożsamym z A^* jest nietrywialna.

Oto przykład argumentu przez redukcję. By zilustrować główną ideę metody, omijając rozbudowane w takich sytuacjach szczegóły techniczne, wykażemy nierozstrzygalność zagadnienia czy $L = \emptyset$ w klasie $\mathcal{L}_0$ (mimo, iż wiemy to już na bazie twierdzenia Rice'a).

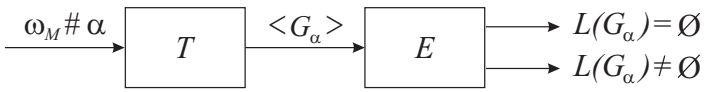
LEMAT 6.3 *Zagadnienie czy $L(G) = \emptyset$ jest nierozstrzygalne dla gramatyk typu 0.*

DOWÓD. Pokażemy, że problem należenia słowa do języka z $\mathcal{L}_0$ może być zredukowany do pytania o pustość języka tej klasy. Niech M będzie maszyną Turinga akceptującą język $L(G)$. Bez trudu możemy skonstruować maszynę K, która dla danych postaci α generuje napis $\alpha\#\alpha$ i cofa głowicę do jego pierwszej litery, oraz maszynę I_α akceptującą język jednoelementowy[4] $\{\alpha\}$. Opis I_α uzupełniamy o przejścia, które na wstępie przesuwają głowicę do pierwszej litery za znakiem $\#$, tak by I_α rozpoczęła pracę na drugiej części słowa $\alpha\#\alpha$. Istotne jest to, że mając dane słowo α jesteśmy w stanie wygenerować automatycznie napis kodujący taki automat I_α. Z tych elementów można łatwo zbudować maszynę, której działanie na danych α polega na uruchomieniu najpierw maszyny kopiującej K, przy czym jej stan finalny utożsamiamy ze stanem początkowym maszyny M. M działa dalej na pierwszej części słowa $\alpha\#\alpha$, a jej stan finalny (zakładamy jak poprzednio, że jest tylko jeden) utożsamiamy ze stanem startowym I_α. Nietrudno sprawdzić, że tak opisana maszyna M_α akceptuje język

$$L(M_\alpha) \;=\; L(M) \cap \{\alpha\}\,.$$

W dowodzie Twierdzenia 6.2 podaliśmy ogólny algorytm zamiany opisu maszyny Turinga na produkcje gramatyki typu 0. Możemy ta metodą wygenerować napis $<G_\alpha>$ kodujący gramatykę G_α odpowiadającą opisanej przed chwilą maszynie M_α.

Cały opisany wyżej proces może być zrealizowany przez procedurę, która startując z napisu $\omega_M\#\alpha$, kodującego opis maszyny M i słowo α, produkuje słowo $<G_\alpha>$ opisujące gramatykę G_α. Niech T oznacza maszynę Turinga realizującą tę procedurę. Załóżmy, że istnieje maszyna Turinga E, która dla danych $<G>$ będących opisem dowolnej gramatyki typu 0 rozstrzyga czy $L(G) = \emptyset$ (zakładamy więc, że problem pustości języka jest rozstrzygalny w $\mathcal{L}_0$). Możemy zatem połączyć maszyny T i E w kaskadę,



otrzymując automat, który z uwagi na fakt, że $\alpha \in L(M)$ wtedy i tylko wtedy, gdy $L(G_\alpha) \neq \emptyset$, odpowiadałby niezawodnie na pytanie czy $\alpha \in L(M)$ czy też nie. Wnosimy stąd, że założenie o rozstrzygalności problemu $L(G) = \emptyset$ było fałszywe.□

Omówimy krótko jeszcze jeden klasyczny problem nierozstrzygalny, tzw. problem Posta. Jest on często wykorzystywany w dowodach nierozstrzygalności w klasie $\mathcal{L}_2$.

DEFINICJA 6.8 (Problem Posta) *Dla pary skończonych zbiorów słów nad alfabetem A,*

$$P \;=\; \{\alpha_1, \alpha_2, \ldots, \alpha_N\} \qquad \textit{oraz} \qquad Q \;=\; \{\beta_1, \beta_2, \ldots, \beta_N\}$$

[4]Jest to oczywiście najprostszy skończony automat deterministyczny $\pi(s_i, a_i) = s_{i+1}$, $i = 0, 1 \ldots n$, gdzie $\alpha = a_0 a_1 \ldots a_n$ oraz $S_F = \{s_{n+1}\}$.

odpowiedzieć na pytanie: czy istnieje ciąg indeksów $i_1, i_2, \ldots, i_M$, $1 \leqslant i_k \leqslant N$, *dla których zachodzi równość konkatenacji*

$$\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_M} \;=\; \beta_{i_1}\beta_{i_2}\ldots\beta_{i_M}\,.$$

Jeśli taki ciąg indeksów istnieje mówimy, że problem Posta dla pary (P, Q) *posiada rozwiązanie.*

Dane dla problemu Posta można zakodować jako

$$\alpha_1\#\alpha_2\#\ldots\alpha_N\%\beta_1\#\beta_2\#\ldots\beta_N \tag{6.6}$$

i związać z nim język złożony ze słów ω tej postaci

$$L_{\text{Post}} = \{\omega :\ \omega \text{ ma postać (6.6) i opisuje problem Posta posiadający rozwiazanie}\}\,.$$

Okazuje się, że L_{Post} jest językiem rekursywnie przeliczalnym. Zgodnie z Definicją 6.6 oznacza to, że problem Posta jest nierozstrzygalny. Zwróćmy jednak uwagę, że w tej ogólnej definicji nierozstrzygalność charakteryzowana jest przez nierekursywność powiązanego języka (6.5), a więc na ogół język ten nie musi być nawet rekursywnie przeliczalny. Fakt, że $L_{\text{Post}} \in \mathcal{L}_{\text{Rp}}$ oznacza, że jesteśmy w nieco lepszej sytuacji: jeśli dla danej pary (P, Q) istnieje rozwiązanie, to jesteśmy w stanie je znaleźć, jednak w ogólnym przypadku nie można stwierdzić jego braku.

Nierozstrzygalność problemu Posta uzyskuje się redukując do niego zagadnienie należenia słowa do języka w klasie $\mathcal{L}_0$. Ze słowem $\omega_M\#\omega$ opisującym pytanie czy $\omega \in L(M)$ można związać specjalnie skonstruowany problem Posta w taki sposób, że posiada on rozwiązanie wtedy i tylko wtedy, gdy $\omega \in L(M)$. Tym samym gdyby problem Posta był rozstrzygalny, rozstrzygalny byłby także problem należenia.

Dzięki swemu sformułowaniu (badanie równości specyficznych konkatenacji słów) problem Posta okazuje się wyjątkowo poręczny w dowodach, w których poszukujemy sposobu na opisanie redukcji tego zagadnienia do typowych pytań zadawanych w klasie gramatyk bezkontekstowych. Problem stopu, jakkolwiek równoważny, jest technicznie bardziej od nich odległy. Oto przykład takiego wykorzystania problemu Posta.

TWIERDZENIE 6.9 *Problem polegający na tym, by na podstawie opisu gramatyki bezkontekstowej* G *stwierdzić czy jest ona niejednoznaczna, jest nierozstrzygalny.*

Dowód. Niech $P = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ i $Q = \{\beta_1, \beta_2, \ldots, \beta_n\}$ będą danymi dla problemu Posta nad alfabetem A. Rozszerzymy ten alfabet nowymi, różnymi od wszystkich $a \in A$ symbolami c_i w liczbie n,

$$A' = A \cup \{c_1, c_2, \ldots, c_n\}\,.$$

Zbudujmy dwie gramatyki bezkontekstowe

$$G_P \;=\; (A',\, \{S_P\},\, S_P,\, \Pi_P) \qquad \text{oraz} \qquad G_Q \;=\; (A',\, \{S_Q\},\, S_Q,\, \Pi_Q)$$

z produkcjami odpowiednio

$$S_P \to \alpha_i S_P c_i \,|\, \alpha_i c_i \qquad \text{oraz} \qquad S_Q \to \beta_i S_Q c_i \,|\, \beta_i c_i \qquad \text{dla} \quad i = 1, \ldots, n.$$

Wówczas $L(G_P)$ składa się ze słów postaci $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_m}c_{i_m}\ldots c_{i_2}c_{i_1}$, natomiast $L(G_Q)$ ze słów $\beta_{i_1}\beta_{i_2}\ldots\beta_{i_m}c_{i_m}\ldots c_{i_2}c_{i_1}$.

Niech dalej G będzie gramatyką postaci

$$G \;=\; (A',\ \{S, S_P, S_Q\},\ S,\ \Pi_P \cup \Pi_Q \cup \{S \to S_P | S_Q\})\,,$$

która jak wiadomo generuje język $L = L(G_P) \cup L(G_Q)$. Każda z gramatyk G_P i G_Q jest jednoznaczna. Jeśli np. ostatnią literą $\gamma \in L(G_P)$ jest c_i, wyprowadzenie γ w G_P musi się zaczynać od $S_P \Rightarrow \alpha_i S_P c_i$. Podobnie kolejne (licząc od końca) litery c_j określają jednoznacznie produkcje użyte w dalszych krokach wyprowadzenia γ w G_P lub w G_Q. Jeśli pewne słowo γ można wyprowadzić zarówno w G_P jak i G_Q, oznacza to, że zachodzi równość

$$\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_m} \;=\; \beta_{i_1}\beta_{i_2}\ldots\beta_{i_m}\,,$$

a więc, że problem Posta dla pary (P, Q) posiada rozwiązanie. Jak widać, jest to równoważne z niejednoznacznością gramatyki G.

Gdybyśmy więc dysponowali algorytmem rozstrzygającym dla dowolnej bezkontekstowej gramatyki G czy jest ona niejednoznaczna, można byłoby wykorzystać go wraz z przedstawioną wyżej konstrukcją do rozstrzygnięcia problemu Posta dla dowolnej pary (P, Q). □

VI.3 Zadania do Rozdziału VI

Zadanie 1

Narysować grafy maszyn Turinga rozpoznających języki

a) $L = \{a^m b^n :\ m > 0 \quad \text{i} \quad m \neq n\}$

b) $L = \{a^m b^n c^m + n :\ m, n \geqslant 0\}$

c) $L = \{a^m b^n c^m n :\ m, n \geqslant 1\}$

Zadanie 2

Narysować grafy maszyn Turinga obliczających następujące funkcje:

a) $f(\omega) = \overleftarrow{\omega}$ dla $\omega \in \{a, b\}^*$

b) $f(n) = 3n$

c) $f(m, n) = \begin{cases} n - m & n > m \\ 0 & n \leqslant m \end{cases}$

d) $f(n) = n \bmod 5$

e) $f(n) = \lfloor n/2 \rfloor$

W zadaniach b)–e) przyjmijmy unarną reprezentację liczb m, n (a więc np. 3 reprezentowane jest przez ciąg 111).

Zadanie 3

Narysować graf maszyny Turinga obliczającej sumę dwóch liczb w postaci binarnej.

Zadanie 4

Narysować graf maszyny Turinga porównującej dwa słowa nad $\{a, b\}$. Dane wejściowe przygotowane są w postaci $\alpha\#\beta$.

Zadanie 5

Zaproponować dogodną reprezentację liczb wymiernych m/n i architekturę maszyny Turinga, która obliczałaby ich sumy i iloczyny.

Zadanie 6

Zdefiniować maszynę Turinga z jednostronnie ograniczoną taśmą oraz udowodnić równoważność takiego modelu maszyny ze standardowym.

Zadanie 7

Pokazać, że maszyna Turinga bez możliwości usuwania znaków, tj. bez przejść postaci $\pi(s, a) = (q, O, L \text{ lub } R)$ jest równoważna ze standardową wersją.

Zadanie 8

Pokazać, że dla każdej maszyny Turinga można skonstruować maszynę równoważną o nie więcej niż 6 stanach.

Zadanie 9

Napisać w dowolnym języku programowania symulator uniwersalnej maszyny Turinga.

Zadanie 10

a) Niech L będzie językiem skończonym. Pokakzać że $L^+ = \bigcup_{n \geqslant 1} L^n$ jest językiem rekursywnie przeliczalnym i opisać algorytm wypisujący wszystkie jego słowa. b) Wykonać to samo zadanie gdy L jest językiem bezkontekstowym.

Zadanie 11

Uzasadnić dlaczego klasa $\mathcal{L}_{\mathrm{Rp}}$ jest zamknięta ze względu na przekrój.

Zadanie 12

Pokazać, że dopełnienie języka bezkontekstowego jest językiem rekursywnym.

Zadanie 13

Pokazać, że problem czy dwie maszyny Turinga akceptują ten sam język jest nierozstrzygalny.

Zadanie 14

Pokazać, że problem czy dla dowolnych maszyn Turinga M_1 i M_2 zachodzi $L(M_1) \subseteq L(M_2)$ jest nierozstrzygalny.

Zadanie 15

Czy problem $L = \overleftarrow{L}$ jest rozstrzygalny w klasie $\mathcal{L}_0$?

Zadanie 16

Czy w klasie $\mathcal{L}_0$ rozstrzygalny jest następujący problem: zbadać czy L zawiera słowo o długości 100.

Zadanie 17

Niech $P = \{001, 0011, 11, 101\}$ oraz $Q = \{01, 111, 111, 010\}$. Czy problem Posta dla P i Q posiada rozwiązanie?

Zadanie 18

Czy problem Posta nad dwuliterowym alfabetem jest rozstrzygalny?

Rozdział VII

Złożoność obliczeniowa

We wcześniejszych rozdziałach interesowaliśmy się głównie strukturalnymi cechami języków określonych klas i różnicami między nimi, czyniąc jedynie poboczne uwagi co do kosztów obliczeniowych związanych z ich rozpoznawaniem. Obecnie zajmiemy się tym zagadnieniem systematycznie i bardziej szczegółowo. Podsumujmy na początek naszą dotychczasową wiedzę.

Języki klasy $\mathcal{L}_3$ rozpoznawane są przez skończone automaty deterministyczne, działające w czasie liniowym względem długości analizowanego słowa. Wykorzystywana pamięć[1] (liczba stanów) jest *stała*, a więc nie zależy od rozmiaru danych wejściowych.

W przypadku, gdy z gramatyk wyeliminować można λ-produkcje proses wyprowadzeń ulega "uporządkowaniu", dzięki czemu długość generowanych form zdaniowych monotonicznie wzrasta. Wówczas algorytmy oparte o pełny przegląd drzewa wyprowadzeń są skuteczną choć kosztowną obliczeniowo metodą analizy składniowej. Koszt rośnie bowiem wykładniczo wraz z długością słowa. Dotyczy to języków klasy $\mathcal{L}_1$. W klasie języków bezkontekstowych istnieją bardziej wydajne metody parsingu (algorytm CYK) o złożoności $O(n^3)$ względem długości słowa, zaś w szczególnym wypadku języków deterministycznych, gdy zastosować można techniki typu $LL(k)$ lub $LR(k)$, analiza składniowa może być wykonana w czasie liniowym $O(n)$. Ilość wykorzystywanej pamięci roboczej w przypadku parsingu w klasach języków $\mathcal{L}_2$ i $\mathcal{L}_1$ jest co najwyżej liniową funkcją rozmiaru danych.

Określanie złożoności obliczeniowej ma sens dla maszyn Turinga, które zawsze osiągają stan stopu. Gdyby dopuścić zapętlone algorytmy, należałoby po prostu przyjąć, że ich złożoność czasowa jest nieskończona, co nie wnosiłoby niczego wartego uwagi do teorii. Złożoność parsingu dla języków klasy $\mathcal{L}_{\mathrm{Rec}}$ może w zasadzie być dowolnie duża i dotyczy to tak czasu obliczeń, jak i rozmiaru niezbędnej pamięci roboczej. W niniejszym rozdziale opiszemy rozwarstwienie $\mathcal{L}_{\mathrm{Rec}}$ na hierarchię klas złożoności obliczeniowej. Omówimy także jej związki z kategoryzacją Chomsky'ego.

[1] Do rozmiaru pamięci wykorzystywanej w procesie rozpoznawania słów nie wliczamy komórek przechowujących dane wejściowe. Istotna jest przestrzeń taśmy wykorzystywana do realizacji procesu parsingu i niezbędna liczba stanów automatu, która w każdej implementacji przekłada się wszakże na rozmiar macierzy przejścia.

VII.1 Złożoność a modele automatu Turinga

W Rozdziale VI.1 podaliśmy standardową definicję maszyny Turinga oraz opisaliśmy szereg wariantów jej architektury, bardziej poręcznych w zastosowaniach do rozmaitych teoretycznych konstrukcji, jak np. opis maszyny uniwersalnej. Zwróciliśmy przy tym uwagę na funkcjonalną równoważność wszystkich modeli maszyny Turinga. Automaty te nie są jednak zupełnie równoważne pod względem nakładu pracy jaką wykonują rozwiązując te same zadania. Weźmy pod uwagę wspomniany już wcześniej przykład procesu dodawania dwóch liczb w postaci binarnej. Jeśli realizować będziemy go na trójtaśmowej maszynie, której dwie taśmy przechowują argumenty, a trzecia przeznaczona jest na zapisanie wyniku, możliwa jest prosta symulacja pozycyjnego dodawania z przeniesieniem, gdy 3 głowice poruszają się synchronicznie w lewo od najmniej znaczących cyfr argumentów. Liczba niezbędnych kroków jest w tym przypadku taka sama jak długość danych, czyli n, a jeśli wziąć także pod uwagę niezbędne przemieszczenie głowic na koniec danych wejściowych przed rozpoczęciem właściwego procesu dodawania, łączna liczba operacji wyniesie $2n$, a więc jest liniowa.[2]

To samo zadanie wykonywane na jednej taśmie (p. Zadanie VI.3) wymaga większej liczby operacji, bowiem głowica musi wielokrotnie "transportować" bity pierwszego składnika na koniec drugiego i tam je sumować. Na każdy z n bitów pierwszego składnika głowica musi wykonać mniej więcej n ruchów w prawo i po wykonaniu sumowania ponownie n ruchów w lewo. Stąd liczba kroków całego procesu to mniej więcej $n \times 2n$, a więc $O(n^2)$.

Na podstawie powyższego przykładu widać, że odpowiedzi na pytania o złożoność obliczeniową są na ogół zależne od tego, jakiego modelu automatu użyjemy do opisu metody rozwiązania. Wolelibyśmy jednak by charakterystyka złożoności algorytmicznej problemu odzwierciedlała bardziej jego immanentny poziom skomplikowania, a w mniejszym stopniu własności użytego formalizmu. Jak zobaczymy, ta niedogodność jest w rzeczywistości mało istotna w przypadku zamiennego użycia automatów jedno- i wielotaśmowych, natomiast poważne różnice pojawiają się w przypadku alternatywnego zastosowania formalizmu deterministycznego i niedeterministycznego.

Rozpocznijmy od ogólnego wyniku dla automatów wielotaśmowych.

LEMAT 7.1 *Każdy algorytm, którego wykonanie na k-taśmowej maszynie Turinga wymaga $O(m)$ ruchów, może być zrealizowany na standardowej jednotaśmowej, maszynie kosztem co najwyżej $O(m^2)$ przejść.*

DOWÓD. Symulacja k-taśmowego automatu na jednej taśmie polega, jak pamiętamy, na sekwencyjnym przedstawieniu zawartości k taśm $\omega_1, \ldots, \omega_k$ w postaci jednego słowa $\omega_1\#\omega_2\#\ldots\omega_k$ z wstawionymi dodatkowymi k symbolami $\uparrow$ znaczącymi

[2] W teorii złożoności obliczeniowej, jak zobaczymy, na ogół nie interesuje nas drobiazgowe oszacowanie liczby kroków algorytmu, lecz raczej jego zachowanie ze względu na długość danych wejściowych. I tak np. nie jest istotne czy dany algorytm wykonuje efektywnie $2n$ kroków, czy może $3n + 10$, natomiast ważne jest, że czas jego działania rośnie *linowo* z długością danych. A więc w szczególności dla dwukrotnie dłuższych danych należy oczekiwać podwojenia czasu działania algorytmu. Oddaje to bardzo dobrze zwięzła i wygodna notacja Bachmanna-Landaua: w tym wypadku zaniedbując wszelkie stałe napiszemy, że złożoność czasowa metody wynosi $O(n)$.

aktualne pozycje odpowiednich głowic. Niech $n = \max\{|\omega_1|, |\omega_2|, \ldots, |\omega_k|\}$. Mamy $m \geqslant n$, ponieważ wykonując zadanie automat na ogół przegląda swoje dane co najmniej raz. Symulując jeden ruch automatu wielotaśmowego, głowica maszyny jednotaśmowej musi na ogół przejrzeć całą zawartość swojej taśmy, wykonując nie więcej niż $kn + \text{const}$, a więc $O(m)$ ruchów. Ponadto maszyna wielotaśmowa może podczas działania rozszerzać swój obszar roboczy, jednak w jednym ruchu może ona dodać nie więcej niż po 1 komórce na każdej taśmie. W sumie w m ruchach przestrzeń robocza może się więc powiększyć o maksymalnie km komórek. Liczbę ruchów maszyny jednotaśmowej można zatem oszacować z góry przez $m \times 2km$ lub krótko przez $O(m^2)$. □

Rozważmy teraz inny klasyczny przykład, na bazie którego postaramy się uchwycić istotę różnicy między rozwiązaniami deterministycznymi a niedeterministycznymi.

Przykład 7.1 Opiszemy tzw. problem spełnialności formuł logicznych, w skrócie SAT. Wszystkie funkcje logiczne od n zmiennych, $\Phi(x_1, \ldots, x_n)$ można przedstawić w tzw. normalnej postaci koniunkcyjnej, np.

$$\Phi(x_1, x_2, x_3) = (x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3)\,,$$

a więc w postaci wyrażenia zbudowanego z pewnej liczby czynników koniunkcyjnych, z których każdy jest alternatywą zmiennych x_i lub ich negacji $\bar{x}_j$. Funkcję logiczną Φ nazywamy spełnialną, jeśli istnieje choć jedno przypisanie wartości logicznych 0,1 zmiennym x_i, dla którego obliczona wartość Φ wynosi 1. Innymi słowy Φ jest spełnialna, jeśli nie jest tożsamościowo równa 0. Problem spełnialności polega na znalezieniu takiego postawienia. Rozwiązanie deterministyczne jest nam dobrze znane ze szkolnych lekcji logiki: należy utworzyć tabelkę z kolumnami odpowiadającymi wartościom $x_1, \ldots, x_n$ oraz Φ, a w wierszach wypisać kolejno wszystkie możliwe podstawienia 0,1 za x_i oraz wyliczoną dla nich wartość Φ. Obliczenia można przerwać gdy w kolejnym kroku w kolumnie Φ pojawi się 1. Tak więc maszyna Turinga realizująca podobną metodę rozwiązania generuje wszystkie n-elementowe ciągi zero-jedynkowe i oblicza dla nich systematycznie wartość Φ. Ponieważ liczba możliwych podstawień wynosi 2^n, złożoność tej metody jest w ogólności nie mniejsza niż $O(2^n)$.

Rozwiązanie przy pomocy niedeterministycznej maszyny Turinga jest znacznie prostsze. Wystarczy by wygenerowała ona niedeterministycznie dowolny n-elementowy ciąg wartości 0,1, a następnie sprawdziła czy wygenerowane podstawienie spełnia formułę Φ. Przypomnijmy, że zgodnie z definicją maszyny niedeterministycznej, rozwiązuje ona określone zadanie jeśli *istnieje* sekwencja jej niedeterministycznych ruchów kończąca się w stanie finalnym. Tak więc złożoność niedeterministycznego algorytmu dla problemu spełnialności wynosi $O(n)$.

Powyższy przykład jest istotny, ponieważ ilustruje on olbrzymią jakościową różnicę między rozwiązaniem deterministycznym a niedeterministycznym. Dla problemu spełnialności nie jest znany algorytm deterministyczny o mniejszym niż wykładniczy koszcie obliczeniowym. Oznacza to, że szkolna metoda tabeli logicznej jest

w gruncie rzeczy najlepszą metodą rozwiązania problemu spełnialności, jaką dziś dysponujemy. Przejście między automatem niedeterministycznym a równoważnym deterministycznym charakteryzuje ogólnie następujący lemat.

LEMAT 7.2 *Każdy algorytm, który realizuje się na niedeterministycznej maszynie Turinga w $O(m)$ krokach może być wykonany przez równoważną deterministyczną maszynę w co najwyżej $O(K^m)$ krokach.*

Uzasadnienie tego lematu polega na przeanalizowaniu kosztów symulacji opisanej w szkicowym dowodzie Twierdzenia 6.1. Stała K ma związek z maksymalnym stopniem rozgałęzień w drzewie przejść niedeterministycznej maszyny.

By lepiej uświadomić sobie charakter różnicy między wykładniczym a wielomianowym kosztem obliczeniowym przeanalizujmy zestawione poniżej dane. Porównujemy szacunkowe czasy wykonania algorytmów o typowych, spotykanych w praktyce złożonościach $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ i $O(2^n)$. Czas w sekundach dla różnych wartości n wyliczyliśmy wg. dostępnych w sieci testowych danych dla przykładowego procesora Intel Core i7-4790K, 4GhZ. Jest on w stanie wykonać ok. 10^{11} operacji zmiennoprzecinkowych na sekundę (czyli ok. 100 gigaflops).

n	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
10	10^{-10}	$3,3 \times 10^{-10}$	10^{-9}	10^{-8}	10^{-8}
100	10^{-9}	$6,6 \times 10^{-9}$	10^{-7}	10^{-5}	3×10^{22} lat
1000	10^{-8}	10^{-7}	10^{-5}	10^{-2}	3×10^{293} lat
1000000	10^{-5}	2×10^{-4}	10	116 dni	∞

Najszybsze superkomputery wykonują obecnie od kilku do kilkudziesięciu petaflops, czyli maksymalnie ok. 10^{16} działań na sekundę. Wyniki w tabeli można by więc poprawić o 5 rzędów wielkości, jednak nie zmienia to jakościowo dramatycznej różnicy między wydajnością algorytmów o złożoności wielomianowej w stosunku do metod o kosztach obliczeniowych narastających wykładniczo: już dla danych umiarkowanego rozmiaru ($n = 100$) obserwujemy wykładniczą eksplozję czasu ich działania. Dla porównania, wiek Wszechświata szacowany jest na ok. 1.4×10^{10} lat.

Widzimy więc, że złożoność wykładnicza w praktyce dyskwalifikuje pewne metody, gdyż nawet dla danych rozsądnej wielkości stają się one bezużyteczne, a rozwiązania okazują się nieosiągalne.

Powyższe rozważania prowadzą nas do wniosku, że różnice w oszacowaniach złożoności procesów obliczeniowych wynikające z użycia różnych modeli maszyny Turinga do ich opisu mają drugorzędne znaczenie, jeśli pozostajemy w klasie automatów deterministycznych. Metoda, którą potrafimy zrealizować wielomianowym względem rozmiaru danych kosztem na wielotaśmowej maszynie Turinga, może być zaimplementowana także na standardowej maszynie bez nadmiernego wzrostu złożoności. Inaczej rzecz ma się w przypadku realizacji niedeterministycznego algorytmu na urządzeniu deterministycznym, bowiem w ogólności spodziewać się wówczas można wykładniczego wzrostu kosztów.

VII.2 Złożoność czasowa i pamięciowa

Dyskusja z poprzedniego rozdziału stanowi motywację dla zdefiniowania oddzielnych klas języków w obrębie $\mathcal{L}_{\mathrm{Rec}}$, różniących się złożonością obliczeniową rozpoznających je algorytmów. W definicjach posłużymy się standardowym, jednotaśmowym modelem maszyny Turinga. Określmy najpierw kilka przydatnych pojęć.

(i) Niech $T_M(\omega)$ oznacza liczbę ruchów, które wykonuje deterministyczna maszyna Turinga M na danych wejściowych $\omega \in L(M)$ do chwili osiągnięcia stanu stopu.

(ii) W przypadku niedeterministycznej maszyny $T_M(\omega)$ oznacza *minimalną* liczbę ruchów jakie wykona M na $\omega \in L(M)$ przed osiągnięciem stanu stopu.

(iii) Niech $S_M(\omega)$ oznacza wielkość najdłuższego odcinka taśmy roboczej wykorzystywanego przez M podczas działania na danych $\omega \in L(M)$.

(iv) W przypadku gdy M jest niedeterministyczna, niech $S_M(\omega, P)$ oznacza maksymalną wielkość wykorzystanej pamięci roboczej podczas wykonywania sekwencji ruchów P na danych $\omega \in L(M)$. Wówczas definiujemy $S_M(\omega) = \min_P S_M(\omega, P)$.

Zwróćmy uwagę, że zarówno T_M jak i S_M określone są wyłącznie dla danych $\omega \in L(M)$, a więc dla słów, dla których stan stopu jest osiągany nawet w przypadku języków klasy $\mathcal{L}_0$.

DEFINICJA 7.1 *Częściową funkcję* $T_M : \mathbb{N} \nrightarrow \mathbb{N}$ *postaci*

$$T_M(n) \;=\; \max_{|\omega|=n} T_M(\omega)$$

nazywamy czasową złożonością obliczeniową maszyny Turinga M (deterministycznej lub nie). Z kolei częściową funkcję $S_M : \mathbb{N} \nrightarrow \mathbb{N}$ *daną jako*

$$S_M(n) \;=\; \max_{|\omega|=n} S_M(\omega)$$

nazywamy złożonością pamięciową maszyny M.

T i S są funkcjami częściowymi, gdyż prawe strony w ich definicjach określone są jedynie dla napisów $\omega \in L(M)$. Mając zdefiniowane dwa rodzaje złożoności maszyny Turinga możemy z kolei określić podobne wielkości dla języków.

DEFINICJA 7.2 *Mówimy, że język $L \in \mathcal{L}_{\mathrm{Rec}}$ jest klasy* $\mathrm{DTIME}(f(n))$ *jeśli istnieje deterministyczna maszyna Turinga M taka, że $L = L(M)$ oraz $T_M(n) = O(f(n))$. Podobnie mówimy, że L jest klasy* $\mathrm{NTIME}(f(n))$ *jeśli istnieje niedeterministyczna maszyna Turinga M akceptująca L, dla której $T_M(n) = O(f(n))$.*

Klasy języków o określonej złożoności pamięciowej $\mathrm{DSPACE}(f(n))$ i $\mathrm{NSPACE}(f(n))$ zdefiniowane są analogicznie.

Oto dwie inkluzje, które nie wymagają dowodów:

$$\begin{aligned} \mathrm{DTIME}(f(n)) &\subseteq \mathrm{NTIME}(f(n)) \\ \mathrm{DTIME}(f(n)) &\subseteq \mathrm{DTIME}(g(n)) \quad \text{gdy} \quad f(n) = O(g(n))\,. \end{aligned}$$

i podobnie dla złożoności pamięciowej. Inną prostą do uzasadnienia inkluzją jest

$$\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n))\,,$$

ponieważ przy wykorzystaniu fragmentu taśmy roboczej długości k maszyna wykonuje co najmniej k ruchów. Ponadto istnieją przykłady języków wskazujące np., że $\mathrm{DTIME}(n^k) \subsetneq \mathrm{DTIME}(n^{k+1})$. W szczególności wiemy już, że $\mathcal{L}_3 \subseteq \mathrm{DTIME}(n)$ oraz $\mathcal{L}_2 \subseteq DTIME(n^3)$. Zachodzi jednak następujące ogólne twierdzenie.

TWIERDZENIE 7.1 *Nie istnieje totalna funkcja $F : \mathbb{N} \to \mathbb{N}$ obliczalna w sensie Turinga, dla której zachodziłoby*

$$\mathcal{L}_{\mathrm{Rec}} \subseteq DTIME(F(n))\,.$$

Ujmując to innym językiem, dla każdej dowolnie szybko wzrastającej funkcji obliczalnej F istnieją języki rekursywne o złożoności większej niż F. Dowód tego intrygującego faktu znaleźć można np. w książce [7].

VII.3 Klasy złożoności obliczeniowej. Kategorie $\mathcal{P}$ i $\mathcal{NP}$ oraz NP-zupełność

Wróćmy jeszcze raz do fundamentalnej różnicy między złożonością rozwiązań deterministycznych i niedeterministycznych dla tych samych problemów. Oto dwa bardzo ważne pojęcia, które formalizują obserwacje na temat tego rozróżnienia.

DEFINICJA 7.3 *Klasa języków $\mathcal{P}$ zdefiniowana jest jako*

$$\mathcal{P} \;=\; \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}(n^k)\,.$$

Podobnie określamy

$$\mathcal{NP} \;=\; \bigcup_{k\in\mathbb{N}} \mathrm{NTIME}(n^k)\,.$$

$\mathcal{P}$ często określa się mianem klasy problemów (języków) rozwiązalnych (ang. *tractable*). Znane są bowiem dla nich deterministyczne algorytmy o wielomianowej złożoności, które w praktyce są w stanie produkować rozwiązania nawet dla danych znacznych rozmiarów. Nie do końca poprawne jest z kolei automatyczne utożsamianie problemów w klasie $\mathcal{NP}$ z tymi, dla których istnieją algorytmy o niewielomianowej złożoności. Ściśle rzecz biorąc $\mathcal{NP}$ to klasa problemów, dla których znane są *wielomianowe* algorytmy niedeterministyczne. Problem, dla którego znamy wykładnicze rozwiązanie deterministyczne wcale nie musi być rozwiązalny w czasie wielomianowym na maszynie niedeterministycznej. Mówiąc potocznie, dla zagadnień typu $\mathcal{NP}$ można szybko (w czasie wielomianowym) sprawdzić poprawność lub niepoprawność przypadkowo wybranego, domniemanego rozwiązania. W odniesieniu do problemu spełnialności, Przykład 7.1, fakt, że $\mathrm{SAT} \in \mathcal{NP}$ oznacza dokładnie tyle, że wybierając przypadkowe podstawienie 0,1 za zmienne x_i jesteśmy w stanie obliczyć $\Phi(x_1, \ldots, x_n)$ w czasie wielomianowym i zweryfikować tym samym czy podstawienie rozwiązuje problem spełnialności. Zwróćmy przy tym uwagę, że nie stwierdziliśmy

dotąd, iż nie istnieje wielomianowy algorytm deterministyczny rozstrzygający problem SAT, powiedzieliśmy jedynie, że takiego dziś nie znamy.

Problemy w klasie $\mathcal{NP}$ w języku angielskim określane są mianem *intractable*. Bezrefleksyjna próba przetłumaczenia tego słowa na język polski jako "nierozwiązalne" nie jest zbyt szczęśliwa, należałoby użyć raczej słowa "oporne". Wszak znamy dla nich deterministyczne metody rozwiązania, na ogół o wykładniczej złożoności, polegające na symulacji przeglądu z powrotami drzewa niedeterministycznych wyprowadzeń. Są to jednak metody niepraktyczne, ponieważ, jak stwierdziliśmy, wykraczają poza ludzką skalę czasową, nawet dla danych zupełnie skromnego rozmiaru.

W podobny sposób definiuje się także kilka innych klas złożoności obliczeniowej. I tak np.

$$\begin{aligned} \mathrm{DLOG} &= \mathrm{DSPACE}(\log_2 n) \\ \mathrm{PSPACE} &= \bigcup_{k\in\mathbb{N}} \mathrm{DSPACE}(n^k) \\ \mathrm{EXPTIME} &= \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}(2^{n^k}) \\ \mathrm{EXPSPACE} &= \bigcup_{k\in\mathbb{N}} \mathrm{DSPACE}(2^{n^k}) \end{aligned}$$

oraz ich niedeterministyczne odpowiedniki NLOG, NPSPACE, NEXPTIME i NEXPSPACE. Dla tych klas znane są następujące inkluzje:

$$\begin{aligned} \mathrm{DLOG} \subseteq \mathrm{NLOG} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathrm{PSPACE} = \mathrm{NPSPACE} \\ \subseteq \mathrm{EXPTIME} \subseteq \mathrm{NEXPTIME} \subseteq \mathrm{EXPSPACE}\,. \end{aligned}$$

Warto zauważyć, że klasyfikacja powyższa nie ma zbyt wiele wspólnego z hierarchią Chomsky'ego. Pewne relacje są oczywiste, np. $\mathcal{L}_3 \subseteq \mathrm{DSPACE}(0)$ (bez użycia pamięci roboczej), a więc także $\mathcal{L}_3 \subseteq \mathrm{DLOG}$. Języki bezkontekstowe leżą w klasie $\mathrm{DTIME}(n^3)$, a deterministyczne nawet w $\mathrm{DTIME}(n)$, a więc $\mathcal{L}_2 \subseteq \mathcal{P}$. Języki bezkontekstowe są rozpoznawane przez niedeterministyczne automaty liniowo ograniczone, a więc $\mathcal{L}_1 = \mathrm{NSPACE}(n)$. Natomiast jak pokazuje Twierdzenie 7.1 istnieją języki rekursywne poza wszystkimi powyższymi klasami.

Równość klas PSPACE = NPSPACE jest konsekwencją twierdzenia W. Savitcha z 1970 roku, [10], które mówi, że koszty pamięciowe symulacji maszyny niedeterministycznej przez deterministyczną rosną co najwyżej kwadratowo,

$$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{DSPACE}((f(n))^2)\,.$$

Inną jego konsekwencją jest równość EXPSPACE = NEXPSPACE Relacja ta przypomina oszacowanie wzrostu złożoności czasowej przy symulacji maszyny wielotaśmowej przez jednotaśmową, p. Lemat 7.1. Zauważmy przy tym, że analogiczne oszacowanie wzrostu złożoności czasowej przy przejściu od niedeterminizmu do determinizmu jest zgodnie z Lematem 6.1 o wiele bardziej pesymistyczne,

$$\mathrm{NTIME}(f(n)) \subseteq \mathrm{DTIME}(K^{f(n)})\,,$$

a więc w rezultacie mamy jedynie inkluzję $\mathcal{NP} \subseteq \mathrm{EXPTIME}$.

Jak stwierdziliśmy na początku tego podrozdziału, najistotniejsze z punktu widzenia praktyki programistycznej jest rozróżnienie między klasami $\mathcal{P}$ i $\mathcal{NP}$. Udowodnienie, że konkretny problem (język) należy do klasy $\mathcal{P}$ polega na wskazaniu rozwiązującego go algorytmu o co najwyżej wielomianowej złożoności czasowej. Nie zawsze jest to zadanie proste. Np. klasyczne zagadnienie programowania liniowego, bardzo ważnej techniki optymalizacyjnej wykorzystywanej w różnych zastosowaniach matematyki, czekało na swoje rozwiązanie o wielomianowej złożoności przez niemal 40 lat. Jednak o wiele bardziej skomplikowanym zadaniem jest wskazanie realistycznego dolnego ograniczenia na liczbę operacji niezbędną do rozwiązania pewnej klasy zadań. Przykład problemu spełnialności formuł logicznych jest w tym kontekście bardzo typowy. Najprostszy — i jak dotąd jedyny jaki znamy — deterministyczny algorytm rozstrzygający SAT wykonuje $O(2^n)$ sprawdzeń. Nie wiemy czy rzeczywiście 2^n jest dolnym oszacowaniem liczby kroków niezbędnych do rozwiązania problemu SAT w każdym przypadku. Gdyby tak było, poszukiwanie lepszej, wielomianowej metody byłoby bezcelowe. Ponieważ jednak dolne oszacowania złożoności czasowej jakimi dysponujemy dla większości problemów są w najlepszym wypadku kwadratowe względem rozmiaru danych, nie można na ich podstawie wykluczyć istnienia bardziej wydajnych metod dla problemów takich jak SAT. Dotykamy tu najistotniejszego, nierozwiązanego zagadnienia w teorii złożoności obliczeniowej:

PROBLEM: *Rozstrzygnąć czy* $\mathcal{P} = \mathcal{NP}$, *czy też* $\mathcal{P} \subsetneq \mathcal{NP}$.

Mówiąc bardziej opisowo, problem sprowadza się do odpowiedzi na pytanie: czy to, że dla pewnego zadania znamy prostą metodę weryfikacji przypadkowo wybranego, domniemanego rozwiązania, gwarantuje istnienie także prostej, ale *systematycznej* metody poszukiwania rozwiązań?

Problem ten pojawił się historycznie po raz pierwszy w korespondencji Kurta Gödla z Johnem von Neumannem w 1956 roku. Precyzyjne sformułowanie i pierwsza próba jego systematycznej analizy pochodzą z pionierskiej i do dziś uważanej za przełomową pracy Stephena Cooka [2]. Omówimy pokrótce idee tam zarysowane.

DEFINICJA 7.4 *Mówimy, że język* $L_1 \subseteq A^*$ *jest wielomianowo redukowalny do języka* $L_2 \subseteq B^*$, *co symbolicznie zapisujemy jako* $L_1 \propto L_2$, *jeśli istnieje maszyna Turinga* T *o wielomianowej złożoności czasowej, która dla dowolnych danych* $\alpha \in A^*$ *tworzy na taśmie słowo* $\beta \in B^*$, *w taki sposób, że* $\alpha \in L_1 \Leftrightarrow \beta \in L_2$.

Języki L_1 *i* L_2 *są wielomianowo równoważne, gdy zachodzi zarówno* $L_1 \propto L_2$ *jak i* $L_2 \propto L_1$.

Sens pojęcia wielomianowej redukowalności jest następujący: jeśli dysponujemy maszyną Turinga M rozpoznającą język L_2 i potrafimy zredukować wielomianowo L_1 do L_2, możemy wykorzystać M do rozstrzygania czy $\alpha \in L_1$, używając najpierw T do przetłumaczenia słowa α na β, a następnie uruchomienia M na β. Wiemy wszak, że $\alpha \in L_1 \Leftrightarrow \beta \in L_2$. Skoro tłumaczenie α na β jest procesem o wielomianowej złożoności, nie podnosi ono zasadniczo kosztów obliczeniowych dla problemu $\alpha \in L_1$ w stosunku do $\beta \in L_2$. W szczególności jeśli $L_2 \in \mathcal{P}$, wówczas także $L_1 \in \mathcal{P}$.

W bardziej praktycznym wymiarze powiemy, że jeśli dysponujemy algorytmem rozwiązującym zadanie Q i potrafimy bez nadmiernego nakładu pracy przetworzyć

dane dla problemu P na serię danych dla problemu Q, możemy wykorzystać algorytm dla Q, być może w wielokrotnej iteracji, do rozwiązania P. Złożoność takiego rozwiązania jest wówczas niemal taka sama jak złożoność algorytmu Q. Redukcja $P \propto Q$ oznacza, że nakład pracy przy konwersji danych (w tym dopuszczalne ich wydłużenie) jest co najwyżej wielomianowy, a więc że procedura dla Q jest wykonywana co najwyżej wielomianową liczbę razy.

Pomysł Cooka był następujący: jeśli zdołamy pogrupować znane problemy obliczeniowe w klasy zagadnień wielomianowo równoważnych, łatwiej będzie zorientować się w podobieństwach i różnicach między nimi co do stopnia ich trudności obliczeniowej. Z jednej strony znajomość wielomianowego w czasie rozwiązania dla choćby jednego członka takie grupy oznacza, że wszystkie problemy w niej zawarte należą do klasy $\mathcal{P}$. Z drugiej strony brak znanych wielomianowych procedur dla wszystkich problemów w pewnej licznej ich klasie uprawdopodobniałby tezę, że są to zagadnienia o znacznej inherentnej trudności, leżące poza klasą $\mathcal{P}$.

Oto kolejna definicja Cooka, idąca o krok dalej.

DEFINICJA 7.5 *Język $L \in \mathcal{NP}$, do którego wielomianowo redukują się wszystkie inne języki w klasie $\mathcal{NP}$ nazywamy NP-zupełnym.*

Oczywiste jest w tym miejscu pytanie: czy ta definicja w ogóle ma sens? Czy istnieje choć jeden język o tak silnej własności? Dowód istnienia NP-zupełnego języka jest głównym wynikiem w pracy Cooka.

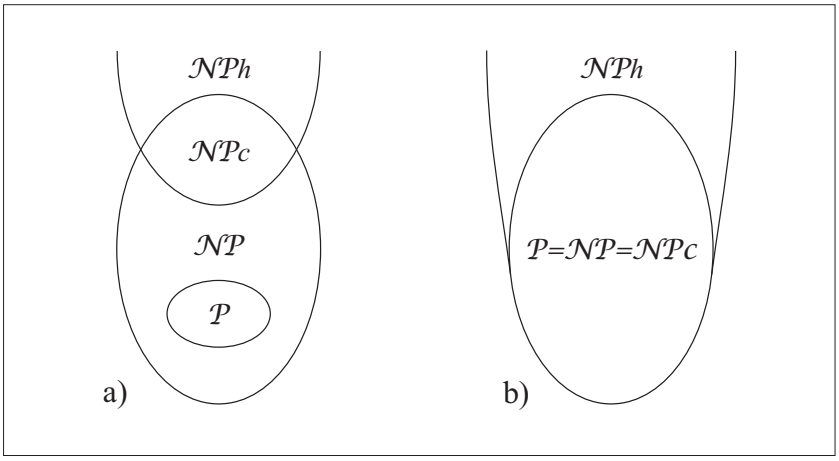
TWIERDZENIE 7.2 (Cook, 1971) *Problem SAT jest NP-zupełny.*

Dowód tego twierdzenia jest raczej złożony technicznie, jednak jego idea jest stosunkowo prosta: istnieje konstruktywny sposób zakodowania opisu dowolnej maszyny Turinga M rozpoznającej język klasy $\mathcal{NP}$ w postaci pewnej funkcji logicznej Φ_M i danych x_i dla niej tak, że $\alpha \in L(M)$ dokładnie wtedy, gdy $\Phi_M(x_1, \ldots x_n) = 1$. Cook wskazał taką metodę kodowania, pokazując jednocześnie, że niezbędny nakład pracy jest wielomianowy.

Zagadnienia NP-zupełne uważać można za *najtrudniejsze* w klasie $\mathcal{NP}$. Jeśli bowiem udałoby się nam zaprojektować algorytm o wielomianowej złożoności czasowej dla pewnego NP-zupełnego zadania Q, oznaczałoby to, że $\mathcal{P} = \mathcal{NP}$, ponieważ każdy inny problem, za pośrednictwem wielomianowej redukcji, dałoby się rozwiązać algorytmem dla Q.

Mając już jeden problem (język) NP-zupełny, dużo łatwiej jest wskazać następne. Jeśli bowiem P jest NP-zupełny oraz $P \propto Q$ dla pewnego problemu $Q \in \mathcal{NP}$, wówczas Q także jest NP-zupełny, ponieważ, jak łatwo zauważyć, relacja $\propto$ jest przechodnia w $\mathcal{NP}$. W ten sposób w ciągu kilkudziesięciu lat odkryto ponad tysiąc problemów NP-zupełnych, w tym wiele zagadnień arytmetycznych, kombinatorycznych, dotyczących grafów, optymalizacyjnych itp.

Zwróćmy uwagę, że definicja NP-zupełności wymaga, aby problem sam był klasy $\mathcal{NP}$. Istnieje inne, ogólniejsze pojęcie zagadnienia *NP-trudnego*. Jest to problem niekoniecznie klasy $\mathcal{NP}$, a więc na ogół wymagający istotnie większych nakładów obliczeniowych niż NPTIME, do którego można wielomianowo zredukować wszystkie zagadnienia w $\mathcal{NP}$. Relację między klasami $\mathcal{P}$, $\mathcal{NP}$ oraz NP-zupełnością i NP-trudnością ilustruje Rys. 7.1.



Rys. 7.1: Relacje między klasami $\mathcal{P}$ i $\mathcal{NP}$ oraz kategoriami problemów NP-zupełnych ($\mathcal{NP}c$) i NP-trudnych ($\mathcal{NP}h$), a) przy założeniu, że $\mathcal{P} \subsetneq \mathcal{NP}$ oraz b) jeśli $\mathcal{P} = \mathcal{NP}$.

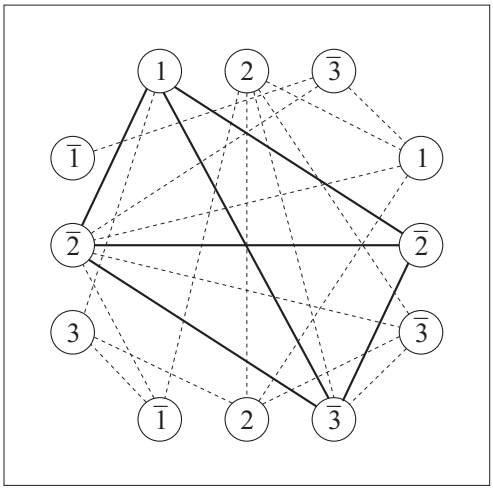
Jako przykład dowodu NP-zupełności pokażemy jak problem SAT może być zredukowany do tzw. problemu kliki w grafie.

Przykład 7.2 W pierwszym kroku pokażemy jak zredukować problem SAT do prostszego problemu 3SAT, w którym funkcja Φ ma specjalną postać, mianowicie każdy jej czynnik koniunkcyjny zawiera dokładnie 3 zmienne, a więc np.

$$\Phi(x_1, x_2, x_3, x_4) = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_2 \vee x_3 \vee \bar{x}_4)$$

Okazuje się, że każdą funkcję logiczną można przekształcić do takiej postaci wielomianowym względem liczby zmiennych nakładem pracy. Oto algorytm dla funkcji w normalnej postaci koniunkcyjnej.

1. Czynniki z 1 lub dwoma zmiennymi rozszerzamy przez powtórzenie zmiennych, np.
$$x_1 \vee \bar{x}_2 \quad \rightarrow \quad x_1 \vee x_1 \vee \bar{x}_2 \,.$$
2. Czynniki o 3 zmiennych pozostawiamy bez zmian.
3. Czynniki o 4 lub więcej zmiennych rozpisujemy na koniunkcje większej liczby czynników o 3 zmiennych z wykorzystaniem tzw. dodatkowych zmiennych łącznikowych. Np.
$$x_1 \vee x_2 \vee x_3 \vee x_4 \quad \rightarrow \quad (x_1 \vee x_2 \vee z) \wedge (\bar{z} \vee x_3 \vee x_4) \,.$$
Równoważność logiczna obydwu reprezentacji jest łatwa do wykazania. Jeśli oryginalna formuła jest niespełniona, oznacza to, że wszystkie $x_i = 0$. Wówczas — przez obecność zmiennej z i jej negacji $\bar{z}$ — jeden z czynników po prawej stronie musi być niespełniony. Odwrotnie, jeśli wyrażenie po prawej stronie jest spełnione, obydwa czynniki koniunkcji muszą być równe 1, a to oznacza, że jeśli $z = 0$, x_1 lub x_2 musi być równe 1. Jeśli zaś $z = 1$, wówczas jedna ze zmiennych x_i w drugim nawiasie musi być równa 1. Zatem wyrażenie po lewej też będzie spełnione.



Rys. 7.2: Graf dla formuły Φ danej w równaniu (7.1). Wierzchołek oznaczony symbolem 1 odpowiada zmiennej x_1, symbolem $\bar{2}$ — zmiennej $\bar{x}_2$ itd. Dla czytelności rysunku pominęliśmy niektóre krawędzie.

Dla większej liczby zmiennych mamy systematycznie

$$x_1 \vee x_2 \vee \ldots \vee x_m \;\; \rightarrow \;\; (x_1 \vee x_2 \vee z_1) \wedge (\bar{z}_1 \vee x_3 \vee z_2) \wedge (\bar{z}_2 \vee x_4 \vee z_3) \wedge \ldots \wedge (\bar{z}_{m-3} \vee x_{m-1} \vee x_m)$$

Podobnie jak wyżej, zachodzi równoważność logiczna obu postaci.

Jeśli więc funkcja Φ od n zmiennych w normalnej postaci koniunkcyjnej składa się z k czynników, a niektóre z nich grupują maksymalnie m zmiennych, każdy z nich rozpisany będzie na $m-2$ nowe czynniki o 3 zmiennych. Nowa formuła Φ_3, zależna od co najwyżej $n+k(m-3)$ zmiennych, liczyć będzie maksymalnie $k(m-2)$ czynniki, a zatem nasza transformacja jest wielomianowa. Problem 3SAT, tak samo jak standardowy SAT, należy oczywiście do klasy NTIME(n), jest więc problemem w $\mathcal{NP}$, a w świetle przeprowadzonej redukcji jest także problemem NP-zupełnym.

Dalej pokażemy jak zastąpić badanie 3SAT rozwiązaniem pewnego problemu z teorii grafów. Problem istnienia kliki rzędu k w danym grafie to pytanie, czy zawiera on jako swój podgraf pełny graf stopnia k. Graf pełny to graf, w którym wszystkie pary wierzchołków są połączone. Nazwa pochodzi stąd, że jeśli interpretujemy graf jako reprezentację relacji towarzyskich w obrębie pewnej liczby osób, kilka będzie grupą, w której każdy zna każdego.

Jeśli Φ jest formułą w koniunkcyjnej postaci z czynnikami o 3 zmiennych, stowarzyszymy z nią graf w następujący sposób. Z każdym czynnikiem związana jest grupa 3 wierzchołków grafu, etykietowanych nazwami występujących tam zmiennych z ewentualna negacją, p. Rys. 7.2 odpowiadający przykładowej formule

$$\Phi \;=\; (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3)\,. \qquad (7.1)$$

Kolejne czynniki reprezentowane są na rysunku trójkami wierzchołków, od góry zgodnie z ruchem wskazówek zegara. Następnie rysujemy połączenia wierzchołków

między grupami. Połączenie dwóch wierzchołków oznacza, że obydwie zmienne nimi reprezentowane mogą jednocześnie przyjmować wartość 1. I tak np. wierzchołek 2 pierwszej grupy łączy się z wierzchołkami 1 i $\bar{3}$ drugiej grupy, lecz nie z $\bar{2}$. Dla lepszej czytelności rysunek nie pokazuje wszystkich krawędzi. Każda klika stopnia 4 odpowiada przypisaniu odpowiednim zmiennym wartości 1 w taki sposób, że Φ jest spełniona. Na naszym rysunku wybraliśmy klikę odpowiadającą przypisaniu $x_1 = \bar{x}_2 = \bar{x}_3 = 1$. Ponieważ w grafie odpowiadającym k czynnikom formuły Φ mamy $3k$ wierzchołków, a liczba krawędzi wychodzących z jednego wierzchołka nie przekracza $3(k-1)$, dlatego maksymalna liczba wszystkich krawędzi to $6k(k-1)/2$, a więc konstrukcja grafu wymaga wielomianowego nakładu pracy względem rozmiaru Φ. Pozostaje jeszcze upewnić się, że problem kliki jest klasy $\mathcal{NP}$. Tak jest w istocie, gdyż automat, bazując na niedeterministycznym wyborze k wierzchołków w grafie, może w $O(k^2)$ krokach sprawdzić czy każde dwa spośród nich łączy krawędź, a więc czy tworzą one klikę. Udowodniliśmy tym samym, że problem istnienia w grafie kliki stopnia k jest NP-zupełny.

Jak wspomnieliśmy, opisano już bardzo obszerny katalog problemów NP-zupełnych. Dla żadnego z nich nie jest znana wielomianowa w czasie metoda rozwiązania. Jest to mocna przesłanka wspierająca hipotezę, że $\mathcal{P} \neq \mathcal{NP}$. Tym samym znaczna część teoretycznie rozwiązalnych zadań prawdopodobnie pozostanie na zawsze poza zasięgiem mocy obliczeniowej komputerów. Wśród nich znajduje się wiele zagadnień optymalizacyjnych posiadających istotne praktyczne zastosowania, np. tzw. problem komiwojażera, lub problem minimalizacji liczby przecięć krawędzi grafu w jego reprezentacji na płaszczyźnie. To ostatnie zagadnienie jest wykorzystywane m.in. w projektowaniu odpornych na indukcyjne zakłócenia siatek połączeń we współczesnych mikroprocesorach.

Warto w tym miejscu wspomnieć o innym, w wielu wypadkach skutecznym podejściu do problemów opornych obliczeniowo. Jeśli np. liczba krawędzi grafu przekracza pewną krytyczną wartość, szanse na to, że istnieje w nim klika określonego rzędu znacznie rosną. Poza tym takich klik może być wiele. Tak więc jeśli mamy powody przypuszczać, że liczba możliwych rozwiązań jest zauważalnie duża, możemy zdać się na przypadek i wygenerować losowo potencjalne rozwiązanie. Kluczowy w tym momencie jest fakt, że poruszamy się w klasie $\mathcal{NP}$, a to oznacza, że weryfikacja poprawności odgadniętego rozwiązania nie jest obliczeniowo kosztowna i nie trzeba się nią martwić nawet przy wielokrotnym powtarzaniu losowania. Jeśli bowiem powtórzymy taki proces dostatecznie wiele razy, wówczas prawdopodobieństwo choć jednego trafnego wyboru wydatnie wzrośnie. Jest przy tym istotne, aby wymagana liczba losowych prób gwarantująca znaczną szansę powodzenia była niewielka w porównaniu z wykładniczą liczbą kroków w systematycznym, deterministycznym przeglądzie przestrzeni możliwych rozwiązań. Np. jeśli pewna formuła logiczna byłaby spełniona przez około 1% ze wszystkich 2^n zero-jedynkowych podstawień, wówczas prawdopodobieństwo, że żadne ze 100 niezależnie wylosowanych podstawień nie spełnia tej formuły wynosiłoby około 0.37. Gdyby zaś liczbę losowań powiększyć do 500, to samo prawdopodobieństwo spadłoby poniżej 0.01.

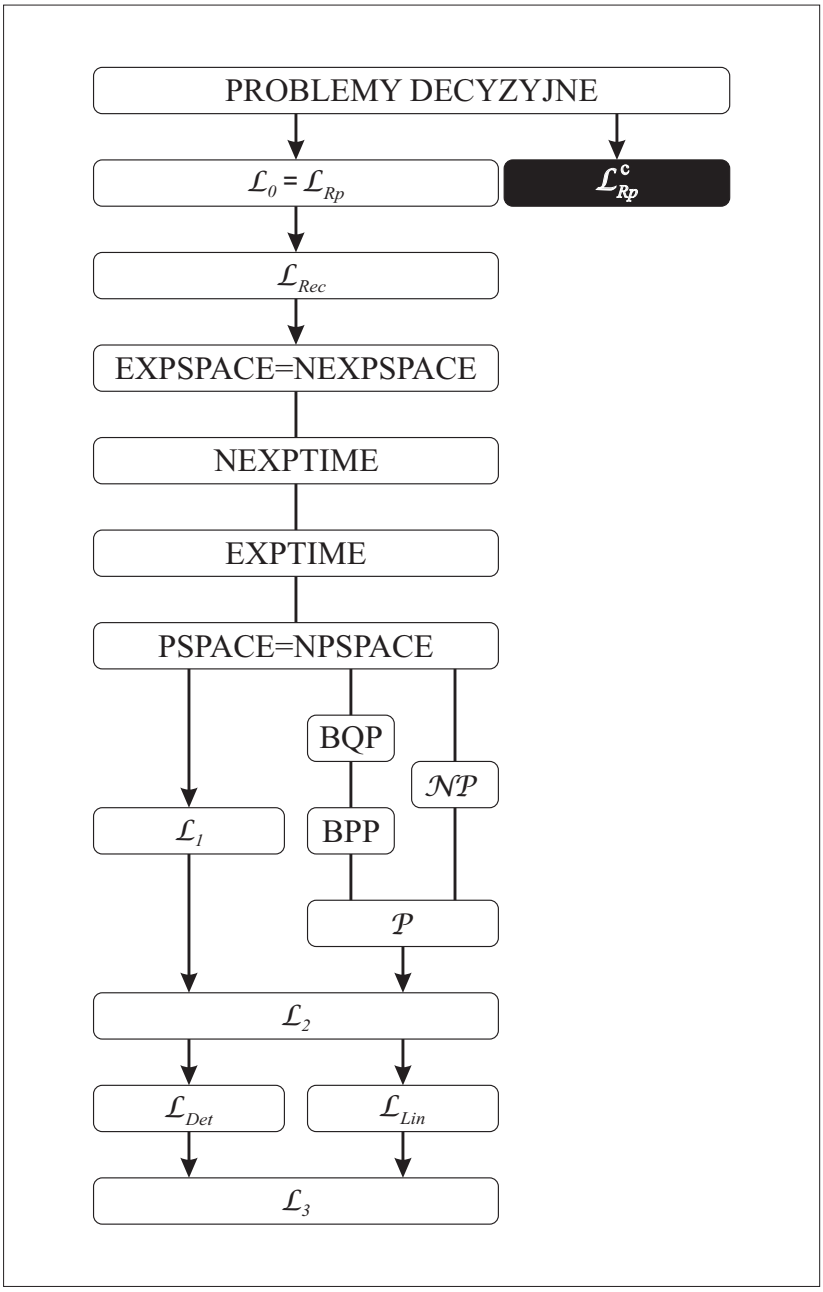
Idąc tym tropem, zdefiniowano tzw. probabilistyczne maszyny Turinga, które pełnią rolę modeli reprezentujących klasę algorytmów losowych, w tym powszech-

nie znanych metod obliczeniowych Monte Carlo. Są to niedeterministyczne maszyny Turinga, w definicji których z relacją przejścia związany jest pewien rozkład prawdopodobieństwa. Maszyna wybiera więc kolejne ruchy losowo, zgodnie z tym rozkładem. Ponadto element losowości zawarty jest także w procesie akceptacji danych wejściowych: maszyna może bowiem z pewnym prawdopodobieństwem podać błędną odpowiedź na pytanie czy $\alpha \in L$. Istnieje wiele modeli probabilistycznych maszyn Turinga, różniących się miedzy innymi szczegółami sposobu akceptacji słów. Dla każdego z tych modeli zdefiniować można szereg klas złożoności obliczeniowej. Jedną z takich klas, dobrze oddającą podbudowane praktyką intuicje co do obliczeniowej skuteczności wielu metod losowych, jest tzw. klasa BPP (bounded error probabilistic polynomial time). Jest to odpowiednik klasy $\mathcal{P}$ dla maszyn probabilistycznych, w działaniu których prawdopodobieństwo podania błędnej odpowiedzi jest ograniczone, np. przez konwencję, że błędne odrzucenie słowa $\alpha \in L(M)$ (bądź zaakceptowanie słowa spoza $L(M)$) zdarza się rzadziej niż raz na 3 przypadki. Przydatność takiego modelu obliczeń polega na tym, że jeśli powtórzymy niedoskonałe losowe obliczenia na tych samych danych niewielką liczbę razy, niemal na pewno pojawi się wśród nich poprawna odpowiedź, a tę poprawność możemy przecież łatwo zweryfikować.

Zdarzają się też sytuacje, w których znaczna złożoność czasowa problemu obliczeniowego traktowana jest jako jego zaleta. Ważnym przykładem jest kryptograficzne wykorzystanie zagadnienia rozkładu liczby N na czynniki pierwsze. Nie jest znana wielomianowa względem liczby bitów $n = \lceil \log_2 N \rceil$ metoda faktoryzacji liczb, z drugiej strony wiele wskazuje na to, że nie jest to problem NP-zupełny. Znane są sub-wykładnicze algorytmy faktoryzacji o złożoności $O\left(2^{C\sqrt[3]{n \log^2 n}}\right)$. Obliczeniowa oporność tego zagadnienia jest właśnie pożądaną cechą przy tworzeniu tzw. szyfrów z publicznym kluczem (algorytm RSA) oraz w niektórych schematach podpisu elektronicznego. Liczba N stanowi podstawę publicznego klucza szyfrującego i jest wybierana wśród liczb postaci $N = pq$, gdzie p i q są dużymi, co najmniej kilkuset bitowymi liczbami pierwszymi. Ujawniana jest sama liczba N, natomiast jej czynniki p i q, bez znajomości których niemożliwe jest deszyfrowanie wiadomości, pozostają znane jedynie właścicielowi klucza. Co prawda znajomość liczby N umożliwia każdemu poznanie metody odczytu szyfrogramów przez rozkład N na czynniki p i q. Jednak jest to zadanie praktycznie niewykonalne ze względu na niewielomianową złożoność czasową algorytmu faktoryzacji.

Z kolei kwantowy algorytm P. Shora [12] z 1997 roku dokonuje rozkładu liczby N w wielomianowym czasie. Jego publikacja zapoczątkowała trwającą do dziś wielką falę zainteresowania kwantową informatyką i badaniami nad możliwością zbudowania kwantowego komputera. Dało to także asumpt do teoretycznych badań nad złożonością obliczeniową na bazie kwantowego modelu maszyny Turinga. Klasa BQP (bounded error quantum polynomial time), do której należy algorytm Shora, jest odpowiednikiem wspomnianej wyżej klasy BPP dla klasycznego modelu probabilistycznego. Ponieważ kwantowy komputer teoretycznie "może" więcej, zachodzi inkluzja $\mathrm{BPP} \subseteq \mathrm{BQP}$, jednak nie wiadomo czy jest ona właściwa. Niewiele wiadomo też o wzajemnej relacji tych klas z $\mathcal{NP}$.

Uporządkujmy na koniec nasze rozważania, zbierając wszystkie relacje między omawianymi klasami złożoności w formie diagramu, p. Rys. 7.3. Przez problemy



Rys. 7.3: Diagram klasyfikacji problemów decyzyjnych, na podstawie [15]. Czarnym kolorem oznaczyliśmy dopełnienie klasy $\mathcal{L}_{\mathrm{Rp}}$, a więc klasę problemów silnie nierozstrzygalnych. Strzałki oznaczają relację właściwej inkluzji, natomiast linie bez grota oznaczają inkluzje, co do których nie wiadomo czy są właściwe, czy też są równościami. Między kategoriami sąsiadującymi w poziomie nie zachodzą relacje inkluzji, choć ich przekroje są na ogół niepuste.

decyzyjne rozumiemy wszelkie problemy obliczeniowe, które standardowo sprowadzamy do uniwersalnej postaci pytania czy $\alpha \in L$ dla języka L odpowiedniej klasy. Na diagramie klasy języków uporządkowane są w pionie malejąco, przy czym strzałka oznacza inkluzję właściwą, np. $\mathcal{L}_{\mathrm{Rec}} \subsetneq \mathcal{L}_{\mathrm{Rp}}$, natomiast linia bez grota oznacza inkluzję, o której jak dotąd nie wiadomo czy jest właściwa, czy być może jest równością. Klas sąsiadujących w poziomie nie łączą relacje inkluzji, lecz ich przekroje na ogół nie są puste, tak jak np. dla $\mathcal{L}_{\mathrm{Det}}$ i $\mathcal{L}_{\mathrm{Lin}}$.

Najwyższa w diagramie klasa odpowiada wszystkim możliwym językom. Jak wiemy, dla zdecydowanej większości z nich nie istnieją *żadne* algorytmiczne metody rozstrzygania czy $\alpha \in L$. Zgodnie z tezą Churcha, $\mathcal{L}_{\mathrm{Rp}}$ jest największą klasą problemów posiadających algorytmiczne rozwiązania. Problemy *w pełni rozstrzygalne*, a więc te, dla których istnieją algorytmy skutecznie odpowiadające na wszelkie pytania czy $\alpha \in L$, czy też $\alpha \notin L$, tworzą klasę $\mathcal{L}_{\mathrm{Rec}}$. Języki rekursywnie przeliczalne określane są niekiedy jako *słabo rozstrzygalne* w odróżnieniu do języków w pełni *nierozstrzygalnych*, leżących w dopełnieniu $\mathcal{L}_{\mathrm{Rp}}$ (oznaczonym na naszym diagramie kolorem czarnym). Gdy bowiem $L \in \mathcal{L}_{\mathrm{Rp}}$, wówczas pytania o to czy $\alpha \in L$, dla tych α, które rzeczywiście należą do L, otrzymują zawsze pozytywną odpowiedź, ponieważ rozpoznająca L maszyna Turinga osiąga dla nich stan stopu. Jedynie w wypadku pytania o należenie, postawionego dla $\alpha \notin L$, rozstrzygającej odpowiedzi nie otrzymamy. W szczególności język L_{Post}, por. Definicja 6.8, jest właśnie taki: jeśli kodowany przez słowo α problem Posta posiada rozwiązanie, można je algorytmicznie znaleźć. W przeciwnym wypadku maszyna Turinga nie potrafi potwierdzić braku rozwiązania.

Z kolei klasa języków rozstrzygalnych $\mathcal{L}_{\mathrm{Rec}}$ rozwarstwia się na szereg podklas o malejącej złożoności czasowej i pamięciowej. Wzajemne relacje między tymi klasami nie są do końca jasne, a nasz diagram ukazuje tylko niektóre z nich.[3] Najbardziej intrygujące są niekompletnie jak dotąd opisane inkluzje między klasami $\mathcal{P}$ i $\mathcal{NP}$, $\mathcal{P}$ i BPP oraz BPP i BQP. Każda z ewentualności — równość albo nierówność — pociąga za sobą daleko idące logiczne konsekwencje dla podstaw informatyki i wielu dziedzin matematyki.

VII.4 Zadania do Rozdziału VII

Zadanie 1

Opisać złożoność metody wyszukiwania przez maszynę Turinga zapisanego na jednej taśmie wzorca $\alpha \in \{a, b\}^*$ wśród słów zapisanych na drugiej taśmie, $\beta_1 \# \beta_2 \# \beta_n$, zakładając, że $|\alpha|, |\beta_i| \leqslant m$ i $m \ll n$.

Zadanie 2

Czy, a jeśli tak, to jak zmieni się złożoność wyszukiwania z poprzedniego zadania, jeśli założymy, że słowa β_i są uporządkowane leksykograficznie?

[3] Strona internetowa `https://complexityzoo.uwaterloo.ca/Complexity_Zoo` gromadzi informacje na temat różnych, często bardzo egzotycznych, klas złożoności obliczeniowej, definiowanych dla odmiennych modeli maszyny Turinga.

Zadanie 3

Zadanie polega na sprawdzeniu czy w nieuporządkowanym zbiorze liczb

$$B = \{n_1, n_2, \ldots, n_m\}$$

występują a) duplikaty $n_i = n_j$, b) trójki $n_i = n_j = n_k$. Jaka jest złożoność metody w obydwu przypadkach? Jak zmienią się te złożoności, jeśli algorytm może wykorzystać uporządkowanie zbioru?

Zadanie 4

Przedstawić problem spełnialności formuł logicznych jako problem decyzyjny $\alpha \in L_{\mathrm{SAT}}$ dla stosownie zdefiniowanego języka.

Zadanie 5

Czy problem SAT posiada rozwiązanie dla formuły $\Phi(x_1, x_2, x_3) = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$?

Zadanie 6

Zaproponować dwutaśmową maszynę Turinga rozpoznającą język $L = \{\omega\omega : \ \omega \in \{a, b\}\}$ w czasie liniowym względem długości ω.

Zadanie 7

Pokazać, że język L z poprzedniego zadania należy do klasy NTIME(n). Udowodnić, że jest on także językiem klasy DTIME(n).

Zadanie 8

Czy język $L = \{\omega\omega\omega : \ \omega \in \{a, b\}\}$ jest także językiem klasy DTIME(n)?

Zadanie 9

Czy istnieją języki $L \in \mathcal{L}_{\mathrm{Rec}} - \mathrm{NTIME}(2^n)$?

Literatura

[1] Alfred V. Aho, Jeffrey D. Ullman, *The Theory of Parsing, Translation and Compiling*, vol. I & II, Prentice Hall, 1972.

[2] Stephen Cook, *The complexity of theorem proving procedures*, Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, str. 151–158.

[3] Alonzo Church, *An unsolvable problem for elementary number theory*, The American Journal of Mathematics **58** (1936), str. 345–363.

[4] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, *Wprowadzenie do teorii automatów, języków i obliczeń*, PWN, Warszawa 2005.

[5] Dexter C. Kozen, *Automata and Computability*, Springer, 1997.

[6] Dexter C. Kozen, *Theory of Computatin*, Springer, 2006.

[7] Peter Linz, *An Introduction to Formal Languages and Automata*, Jones & Barlett Publ., 2006.

[8] Rohit Parikh, *On Context-Free Languages*, Journal of the Association for Computing Machinery **13** (4), str. 570–581.

[9] John. C. Martin, *Introduction to Languages and the Theory of Computation*, McGraw-Hill, 2003.

[10] Grzegorz Rozenberg, A. Salomaa (red.), *A Handbook of Formal Languages*, vol. I–III Springer, 1997.

[11] Arto Salomaa, *Formal languages*, Academic Press, 1973.

[12] Peter W. Shor, *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*, SIAM J. Sci. Statist. Comput. **26** (1997), str. 1484–1509.

[13] Alan M. Turing, *On computable numbers with an application to the Entscheidungsproblem*, Proc. London Math. Soc. **2**, 42 (1936), str. 230–265.

[14] Adam Brooks Webber, *Formal Language – A Practical Introduction*, Franklin, Beedle & Ass., 2008.

[15] Na podstawie artykułu w Wikipedii (wersja angielska) “Complexity class”, https://en.wikipedia.org/wiki/Complexity_class [data dostępu 24.09.2015], na licencji CC BY–SA.

Wykaz symboli i oznaczeń

W nawiasach kwadratowych podano nr strony, na której znajduje się określenie symbolu.

$\subseteq$, $\subsetneq$	relacja inkluzji [8]
$\mathcal{P}(A)$	zbiór potęgowy zbioru A [8]
$\mathcal{F}(A)$	rodzina skończonych podzbiorów A [8]
$A \oplus B$	różnica symetryczna zbiorów [9]
$A \times B$	iloczyn kartezjański zbiorów [10]
$[a]_R$	klasa abstrakcji relacji równoważności R [11]
$A \nrightarrow B$	częściowe odwzorowanie z A do B [12]
$F^{-1}(V)$	przeciwobraz V przez odwzorowanie F [12]
$F \circ G$	złożenie odwzorowań [12]
$m \mid n$	m jest podzielnikiem n
$O(f)$	symbol asymptotyczny Bachmanna-Landaua [20]
A^*	domknięcie konkatenacyjne A, zbiór napisów nad A [27]
$\alpha\beta$	konkatenacja napisów α i β [27]
λ	napis pusty [27]
$\lvert\alpha\rvert$	długość napisu α [27]
$\lvert\alpha\rvert_a$	liczba liter a w α [27]
$L_1 L_2$	konkatenacja języków L_1 i L_2 [29]
L_1/L_2, $L_1 \backslash L_2$	iloraz prawo- i lewostronny języków L_1 i L_2 [30]
$\frac{dL}{d\beta}$	pochodna języka z względem β [30]
$\overleftarrow{\alpha}$, $\overleftarrow{L}$	lustrzane odbicie [30]
$h^*(L)$	obraz języka L przez homomorfizm h [31]
$\chi^*(L)$	obraz L przez podstawienie χ [31]
$\gamma \to \eta$	produkcja gramatyki [35]
$\alpha \Rightarrow_G \beta$	relacja wyprowadzenia w gramatyce G [35]
$\alpha \Rightarrow_G^* \beta$	tranzytywne domknięcie relacji $\Rightarrow_G$ [35]
$L(G)$	język generowany przez gramatykę G [35]
$\Rightarrow_M$	relacja przejścia automatu M [42]
$\mathcal{L}_3$	klasa języków regularnych [43]
$\mathcal{L}_2$	klasa języków bezkontekstowych [81]

$\mathcal{L}_{\mathrm{Ndet}}$	klasa języków niedeterministycznych [99]
$\mathcal{L}_{\mathrm{Det}}$	klasa języków deterministycznych [102]
$LL(k)$	typ gramatyki w klasie języków deterministycznych [104]
$\mathcal{LL}(k)$	klasa języków bezkontekstowych o gramatyce $LL(k)$ [107]
$\mathcal{L}_{\mathrm{Lin}}$	klasa języków generowanych przez gramatyki liniowe [107]
$\mathcal{L}_1$	klasa języków kontekstowych [126]
$\mathcal{L}_{\mathrm{Rec}}$	klasa języków rekursywnych [131]
$\mathcal{W}_G(\omega)$	przestrzeń robocza słowa ω w gramatyce G [133]
M_U	uniwersalna maszyna Turinga [149]
$\mathcal{L}_0$	klasa języków generowana przez gramatyki typu 0 [150]
$\mathcal{L}_{\mathrm{Rp}}$	klasa języków rekursywnie przeliczalnych [152]
$<G>$	standardowa reprezentacja gramatyki w postaci słowa [157]
SAT	problem spełnialności formuł logicznych [167]
$\mathcal{P}$, $\mathcal{NP}$	klasy złożoności obliczeniowej [170]