Uniwersytet im. Adama Mickiewicza w Poznaniu
Wydział Matematyki i Informatyki

Paweł Skórzewski

Wydajne algorytmy parsowania dla języków o szyku swobodnym

Rozprawa doktorska

Promotor:
prof. UAM dr hab. Krzysztof Jassem

Promotor pomocniczy:
dr Filip Graliński

Poznań 2014

Badania, których wyniki przedstawia niniejsza praca,
zostały sfinansowane ze środków Narodowego Centrum Nauki
przyznanych na podstawie decyzji numer DEC-2011/01/N/ST6/02032.

Spis treści

Podziękowania

Składam serdeczne podziękowania Panu Profesorowi Krzysztofowi Jassemowi za okazaną życzliwość, liczne cenne rady i uwagi oraz za wszelką pomoc naukową.

Dziękuję również Panu Doktorowi Filipowi Gralińskiemu za wszelką okazaną pomoc, zwłaszcza dotyczącą parsera Gobio i systemu PSI-Toolkit. Dziękuję całemu zespołowi Pracowni Systemów Informacyjnych UAM za cenne dyskusje i pomysły.

Chciałbym również wyrazić swoją wdzięczność Rodzicom oraz przede wszystkim mojej Żonie Magdalenie za nieustające wsparcie i wielką cierpliwość.

Rozdział 1

Wstęp

Języki naturalne cechuje wielka różnorodność pod względem swobody szyku wyrazów w zdaniu. Istnieją języki, w których szyk wyrazów pełni kluczową rolę w identyfikacji poszczególnych składników zdania. Takie języki nazywamy analitycznymi bądź izolującymi. W językach analitycznych swoboda szyku zdania jest mocno ograniczona. Na drugim biegunie mamy języki fleksyjne, w których kategorie gramatyczne są realizowane za pomocą odmiany wyrazów. Pozwala to na względną swobodę, jeśli chodzi o szyk wyrazów. W tych językach zamiana szyku wyrazów służy uwydatnieniu jakiegoś członu przez umieszczenie go na początku lub na końcu wypowiedzenia. Do języków o swobodnym szyku wyrazów zalicza się między innymi język polski.

Analiza składniowa pełni kluczową rolę w przetwarzaniu języka naturalnego. Opracowano wiele metod parsowania języków naturalnych, jednak istnieje wciąż potrzeba znajdowania nowych, lepszych i wydajniejszych rozwiązań. Dominacja języków analitycznych, takich jak angielski czy chiński, sprawia, że badania koncentrują się głównie na językach o szyku ustalonym, podczas gdy języki o szyku swobodnym są słabiej zbadane.

Wśród metod formalnego opisu języków można wyróżnić różnego rodzaju gramatyki probabilistyczne. Wykorzystują one narzędzia rachunku prawdopodobieństwa do opisu języka. Dzięki temu można tworzyć modele języków, porównywać poprawność różnych zdań czy wiarygodność różnych interpretacji danego zdania.

Niniejsza rozprawa stawia sobie za cel zbadanie pewnego problemu teoretycznego z zakresu gramatyk probabilistycznych oraz optymalizację związanego z nim problemu implementacyjnego.

Część teoretyczna poświęcona jest zagadnieniom formalnego opisu języków o szyku swobodnym i algorytmom ich analizy składniowej. Rozważam w niej sposoby wykorzystywania gramatyk probabilistycznych do opisu języków swobodnego szyku. Definiuję autorski formalizm probabilistycznych gramatyk binarnych generujących drzewa (PTgBG), który stanowi probabilistyczne rozszerzenie formalizmu TgBG (gramatyk binarnych generujących drzewa). Prezentuję również parser wykorzystujący ten formalizm.

Z drugiej strony celem niniejszej pracy jest zbadanie, w jaki sposób implementacja algorytmu parsowania wpływa na jego wydajność. W szczególności, przedstawiam proces adaptacji parsera do systemu przetwarzania języka naturalnego. Analizuję napotkane trudności i ewaluuję wydajność na poszczególnych etapach optymalizacji. Przedstawiam też wnioski płynące z tego procesu.

Rozdział 1 stanowi krótkie wprowadzenie w tematykę pracy oraz opisuje jej podstawowe cele i założenia, a także przedstawia strukturę pracy.

Rozdział 2 zawiera podstawowe wiadomości z zakresu teorii języków i gramatyk formalnych, w tym informacje na temat probabilistycznych gramatyk bezkontekstowych (PCFG).

W rozdziale 3 przedstawiam istniejące formalizmy, które umożliwiają opis języków o szyku swobodnym. Staram się przedstawić różnorodne podejścia do tego problemu w sposób spójny za pomocą aparatu matematycznego wprowadzonego w rozdziale 2. Prezentacja formalizmów w tej pracy może więc znacznie różnić się od oryginałów.

Rozdział 4 zawiera autorską koncepcję probabilistycznych gramatyk binarnych generujących drzewa (PTgBG). Dowodzę w nich twierdzeń mówiących o związkach między PTgBG a PCFG i między klasami generowanych przez nie języków.

Rozdział 5 zawiera przegląd istniejących algorytmów parsowania, które wykorzystują prawdopodobieństwa i inne rodzaje wag.

W rozdziale 6 opisuję implementację parsera korzystającego z gramatyk PTgBG. Przedstawiam opis adaptacji tegoż parsera do zestawu narzędzi przetwarzania języka naturalnego. Opisuję proces optymalizacji zaimplementowanego parsera. Rozdział zawiera też ewaluację uzyskanych wyników i płynące z niej wnioski.

Krótkie podsumowanie całej pracy stanowi rozdział 7.

Rozdział 2

Podstawowe pojęcia

Niniejszy rozdział wprowadza podstawowe pojęcia niezbędne w dalszej części pracy. Zostały one pogrupowane tematycznie. Podrozdział 2.1 definiuje podstawowe cegiełki teorii języków formalnych, takie jak symbole, alfabety i łańcuchy. Gramatyki formalne opisane są w podrozdziale 2.2. Podrozdział 2.3 przybliża pojęcia związane z drzewami składniowymi, natomiast w podrozdziale 2.4 wyjaśnia się na przykładzie probabilistycznych gramatyk bezkontekstowych, w jaki sposób wprowadza się prawdopodobieństwa do opisu języków formalnych.

2.1. Symbole, alfabety, łańcuchy, języki[1]

Pojęcie 2.1 (symbol)**.** Pojęcie *symbolu* będziemy traktować jako pojęcie pierwotne, nieposiadające ścisłej definicji. Symbolem może być dowolny pojedynczy znak, litera, cyfra, wyraz itp.

Definicja 2.2 (alfabet)**.** Dowolny skończony zbiór symboli nazywamy *alfabetem*.

Uwaga 2.1 (słownik)**.** Czasami, zwłaszcza gdy rozważanymi symbolami będą wyrazy języka naturalnego, będziemy mówić *słownik* zamiast *alfabet*. Takie nazewnictwo jest wówczas bardziej intuicyjne i zapobiega nieporozumieniom.

Przykład 2.1 (alfabety)**.** Za alfabety możemy uważać następujące zbiory:

(a) $\{A, B, C, D, E, F, G, H, I, K, L, M, N, O, P, Q, R, S, T, U, V, X, Y, Z\}$,
(b) $\{a, ą, b, c, ć, d, e, ę, f, g, h, i, j, k, l, ł, m, n, ń, o, ó, p, r, s, ś, t, u, w, y, z, ź, ż\}$,
(c) $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$,
(d) $\{(0), (1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), \ldots, (59)\}$,
(e) $\{\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta, \iota, \kappa, \lambda, \mu, \nu, \xi, o, \pi, \rho, \sigma, \tau, \upsilon, \phi, \chi, \psi, \omega\}$,
(f) $\{ę, \flat, \forall, \diamondsuit, \partial\}$,

[1] Na podstawie [19]

(g) $\{\, kot\,,\ pies\,,\ lis\,,\ biega\,,\ szczeka\,\}$.

Elementy tych zbiorów to symbole. Zbiór z ostatniego przykładu (g) wygodniej byłoby nazwać raczej słownikiem niż alfabetem.

Zbiór $\{0, 1, 2, 3, \ldots\}$ (zbiór wszystkich liczb naturalnych) nie jest alfabetem, ponieważ nie jest skończony.

Definicja 2.3 (słowo, łańcuch)**.** *Słowem* lub *łańcuchem nad alfabetem* V nazywamy skończony ciąg symboli tego alfabetu. W przypadku, gdy wiadomo, o jaki alfabet chodzi, będziemy mówić krótko: *słowo* lub *łańcuch*.

Łańcuch (słowo) przedstawiany jest jako napis, w którym symbole pisane są jeden za drugim.

Podobnie jak w przypadku *alfabetu* i *słownika*, również w tym przypadku występują dwa różne określenia na to samo pojęcie. Ponieważ w tej pracy będą występować przykłady odwołujące się do języka naturalnego, będę unikał stosowania określenia *słowo*, aby nie doprowadzać do nieporozumień. Określenie *łańcuch* jest bardziej jednoznaczne.

Przykład 2.2 (łańcuchy)**.** Weźmy alfabet $V = \{a, b, c, d, e\}$. Wówczas łańcuchami nad alfabetem V są na przykład ciągi *abc*, *aaaab* czy *ebeced*.

Przykład 2.3 (łańcuchy)**.** Niech dany będzie alfabet $V = \{\, kot\,,\ pies\,,\ lis\,,\ biega\,,\ szczeka\,\}$. Wówczas łańcuchami nad alfabetem V są na przykład ciągi *kot szczeka*, *biega lis lis szczeka* czy *kot pies kot kot kot* (gdy mowa o łańcuchach złożonych z wyrazów, poszczególne wyrazy będziemy dla przejrzystości oddzielać spacjami, których nie należy traktować jako osobne symbole).

Definicja 2.4 (długość łańcucha)**.** Liczbę symboli w łańcuchu w nazywamy *długością* łańcucha w i oznaczamy przez $|w|$.

Przykład 2.4 (długość łańcucha)**.** Łańcuch *aabcc* (nad alfabetem $\{a, b, c\}$) ma długość 5, a łańcuch *pies szczeka* (nad słownikiem (g) z przykładu 2.1) ma długość 2. Możemy zatem napisać:

$$|aabcc| = 5\ ,$$
$$|\, pies\ szczeka\,| = 2\ .$$

Definicja 2.5 (łańcuch pusty)**.** Ciąg niezawierający żadnych symboli nazywamy *łańcuchem pustym* i oznaczamy symbolem ϵ.

Łańcuch pusty ma długość 0 ($|\epsilon| = 0$) i jest łańcuchem nad każdym alfabetem.

Definicja 2.6 (konkatenacja, złożenie)**.** Jeżeli wypiszemy kolejno wszystkie symbole jednego łańcucha, a zaraz za nimi kolejno wszystkie symbole

drugiego łańcucha, to otrzymany w ten sposób nowy łańcuch nazywamy *konkatenacją* albo *złożeniem* tych dwóch łańcuchów. Podobnie definiujemy konkatenację większej liczby łańcuchów.

Przykład 2.5 (konkatenacja)**.** Konkatenacją łańcuchów $ABCDE$ i FGH (nad alfabetem (a) z przykładu 2.1) jest łańcuch $ABCDEFGH$.

Konkatenacją łańcuchów prs, $ś$, tu i $wyzźż$ (nad alfabetem (b) z przykładu 2.1) jest łańcuch $prsśtuwyzźż$.

Konkatenacją łańcuchów *szczeka pies* i *biega kot* (nad alfabetem (g) z przykładu 2.1) jest łańcuch *szczeka pies biega kot*.

Konkatenację n łańcuchów oznaczonych przez w_1, w_2, ..., w_n oznaczamy przez zestawienie obok siebie oznaczeń tych łańcuchów, w tym przypadku przez $w_1w_2 \ldots w_n$.

Bezpośrednio z definicji operacji konkatenacji wynika jej łączność:

$$x(yz) = xyz = (xy)z \ .$$

Definicja 2.7 (podłańcuch)**.** Łańcuch w' jest podłańcuchem łańcucha w, jeżeli istnieją łańcuchy u, v takie, że $w = uw'v$.

Łańcuch pusty jest podłańcuchem każdego łańcucha.

Przykład 2.6 (podłańcuchy)**.** Podłańcuchami łańcucha *abcde* są na przykład łańcuchy ϵ, *a*, *abc*, *bcd*, *de*, *abcde*.

Podłańcuchami łańcucha *pies kot lis* są na przykład łańcuchy ϵ, *lis*, *pies kot*, *kot*, *pies kot lis*.

Definicja 2.8 (język)**.** Dowolny podzbiór zbioru wszystkich słów nad danym alfabetem nazywamy *językiem*.

Zbiór wszystkich słów nad danym alfabetem V jest również językiem i oznacza się go symbolem V^*.

Język wszystkich słów nad danym alfabetem V z wyjątkiem słowa pustego ϵ oznacza się symbolem V^+.

Przez V^n będziemy rozumieć zbiór $\{w \in V^* \colon |w| = n\}$.

Przykład 2.7 (języki)**.** Niech dany będzie alfabet $V = \{a, b, c\}$. Przykładami języków złożonych z łańcuchów nad tym alfabetem są między innymi następujące zbiory łańcuchów:

(a) $V^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \ldots, ccc, aaaa, \ldots\}$,
(b) $V^+ = \{a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \ldots, ccc, aaaa, \ldots\}$,
(c) $\{a, aa, aaa, aaaa, \ldots\}$,
(d) $\{\epsilon, abc, cab, cba\}$,

(e) $\{\epsilon\}$,
(f) $\emptyset$.

Zauważmy, że języki utworzone z łańcuchów nad (skończonym) alfabetem mogą być zarówno zbiorami skończonymi, jak i nieskończonymi, a nawet zbiorem pustym.

2.2. Gramatyki

Definicja 2.9 (gramatyka)**.** *Gramatyką (nieograniczoną)* nazywamy uporządkowaną czwórkę $G = (V, T, R, S)$, w której:
— V jest alfabetem, nazywanym *alfabetem symboli pomocniczych (nieterminalnych)* albo krótko *alfabetem zmiennych*,
— T jest rozłącznym z V alfabetem, nazywanym *alfabetem symboli końcowych (terminalnych)*,
— R jest zbiorem *reguł produkcji*, czyli napisów postaci $\zeta \to \xi$, gdzie $\zeta \in (V \cup T)^+$, $\xi \in (V \cup T)^*$,
— wyróżniony symbol $S \in V$ nazywany jest *symbolem początkowym*.

Jeżeli produkcja r jest postaci $\zeta \to \xi$, to łańcuch ζ będziemy nazywać *poprzednikiem produkcji* r, a łańcuch ξ — *następnikiem produkcji* r.

Definicja 2.10 (gramatyka kontekstowa)**.** *Gramatyką kontekstową* nazywamy gramatykę $G = (V, T, R, S)$, w której każda produkcja jest postaci $\zeta A \xi \to \zeta \omega \xi$, gdzie $A \in V$, $\zeta, \xi \in (V \cup T)^*$, $\omega \in (V \cup T)^+$.

Definicja 2.11 (gramatyka bezkontekstowa z ϵ-produkcjami)**.** *Gramatyką bezkontekstową z ϵ-produkcjami* nazywamy gramatykę $G = (V, T, R, S)$, której każda produkcja jest postaci $A \to \omega$, gdzie $A \in V$, $\omega \in (V \cup T)^*$.

Definicja 2.12 (gramatyka bezkontekstowa ϵ-wolna)**.** *Gramatyką bezkontekstową bez ϵ-produkcji* albo *gramatyką bezkontekstową ϵ-wolną* nazywamy gramatykę $G = (V, T, R, S)$, której każda produkcja jest postaci $A \to \omega$, gdzie $A \in V$, $\omega \in (V \cup T)^+$.

Twierdzenie 2.1. *Każda gramatyka bezkontekstowa ϵ-wolna jest gramatyką kontekstową.*

Dowód. Jeżeli w definicji gramatyki kontekstowej (definicja 2.10) przyjmiemy $\zeta = \xi = \epsilon$, otrzymamy definicję gramatyki bezkontekstowej ϵ-wolnej. □

Gramatyki bezkontekstowe ϵ-wolne będziemy krótko nazywać *gramatykami bezkontekstowymi* (ang. *context-free grammars*, CFG). W dalszej części pracy będziemy zajmować się tylko takimi gramatykami.

Przykład 2.8 (gramatyka bezkontekstowa)**.** Przykład gramatyki bezkontekstowej $G = (V, T, R, S)$:
- $V = \{S\}$,
- $T = \{a, b, c\}$,
- $R = \{S \to aSb,\ S \to ab,\ S \to c\}$,
- $S \in V$ jest symbolem początkowym.

Przykład 2.9 (gramatyka bezkontekstowa)**.** Przykład gramatyki bezkontekstowej $G = (V, T, R, S)$:
- $V = \{\, S\,,\ N\,,\ V\,\}$,
- $T = \{\, \textit{psy}\,,\ \textit{koty}\,,\ \textit{myszy}\,,\ \textit{widzą}\,,\ \textit{jedzą}\,\}$,
- $R = \{\, S \to N\ V\ N\,,\ S \to N\ V\,,\ S \to V\ N\,,\ N \to \textit{psy}\,,\ N \to \textit{koty}\,,\ N \to \textit{myszy}\,,\ V \to \textit{jedzą}\,,\ V \to \textit{widzą}\,\}$,
- $S \in V$ jest symbolem początkowym.

Definicja 2.13 (postać normalna Chomsky'ego)**.** Gramatyka bezkontekstowa $G = (V, T, R, S)$ jest w *postaci normalnej Chomsky'ego*, jeżeli każda jej produkcja ma postać $A \to BC$ lub $A \to a$, gdzie $A, B, C \in V$, $a \in T$.

Definicja 2.14 (postać zbinaryzowana)**.** Gramatyka bezkontekstowa $G = (V, T, R, S)$ jest w *postaci zbinaryzowanej*, jeżeli każda jej produkcja ma co najwyżej dwa symbole po prawej stronie.

Uwaga 2.2. Każda gramatyka w postaci normalnej Chomsky'ego jest jednocześnie w postaci zbinaryzowanej. Gramatyka w postaci zbinaryzowanej nie musi w ogólności być w postaci normalnej Chomsky'ego.

Gramatyka z przykładu 2.8 nie jest gramatyką w postaci normalnej Chomsky'ego, ponieważ zawiera regułę $S \to aSb$, która ma więcej niż dwa symbole po prawej stronie, oraz regułę $S \to ab$, która ma po prawej stronie więcej niż jeden symbol końcowy.

Gramatyka z przykładu 2.9 również nie jest gramatyką w postaci normalnej Chomsky'ego, ponieważ zawiera regułę $S \to N\ V\ N$, która ma więcej niż dwa symbole po prawej stronie. Gramatyki z przykładów 2.8 i 2.9 nie są również gramatykami w postaci zbinaryzowanej.

Gramatyka z przykładu 2.10 jest gramatyką w postaci zbinaryzowanej, ale nie jest gramatyką w postaci normalnej Chomsky'ego.

Przykład 2.10 (postać zbinaryzowana)**.** Przykład gramatyki bezkontekstowej $G = (V, T, R, S)$ w postaci zbinaryzowanej:
- $V = \{S, X\}$,
- $T = \{a, b, c\}$,
- $R = \{S \to aX,\ X \to Sb,\ S \to ab,\ S \to c\}$,
- $S \in V$ jest symbolem początkowym.

Gramatyki z przykładów 2.11 i 2.12 są w postaci normalnej Chomsky'ego. Są one jednocześnie gramatykami w postaci zbinaryzowanej.

Przykład 2.11 (postać normalna Chomsky'ego)**.** Przykład gramatyki bezkontekstowej $G = (V, T, R, S)$ w postaci normalnej Chomsky'ego:
- $V = \{S, X, A, B\}$,
- $T = \{a, b, c\}$,
- $R = \{S \to AX,\ S \to AB,\ S \to c,\ X \to SB,\ A \to a,\ B \to b\}$,
- $S \in V$ jest symbolem początkowym.

Przykład 2.12 (postać normalna Chomsky'ego)**.** Przykład gramatyki bezkontekstowej $G = (V, T, R, S)$ w postaci normalnej Chomsky'ego:
- $V = \{\, S\,,\ N\,,\ VP\,,\ V\,\}$,
- $T = \{\, \mathit{psy}\,,\ \mathit{koty}\,,\ \mathit{myszy}\,,\ \mathit{widzą}\,,\ \mathit{jedzą}\,\}$,
- $R = \{\, S \to N\ VP\,,\ S \to VP\ N\,,\ VP \to V\ N\,,\ VP \to N\ V\,,\ VP \to \mathit{widzą}\,,\ VP \to \mathit{jedzą}\,,\ N \to \mathit{psy}\,,\ N \to \mathit{koty}\,,\ N \to \mathit{myszy}\,,\ V \to \mathit{jedzą}\,,\ V \to \mathit{widzą}\,\}$,
- $S \in V$ jest symbolem początkowym.

Definicja 2.15 (bezpośrednia wyprowadzalność)**.** Niech $G = (V, T, R, S)$ będzie dowolną gramatyką oraz $\zeta, \xi, \omega, \omega' \in (V \cup T)^*$. Jeżeli istnieje produkcja $r = (\omega \to \omega') \in R$, to łańcuch $\zeta\omega'\xi$ nazywamy *bezpośrednio wyprowadzalnym* z łańcucha $\zeta\omega\xi$ w gramatyce G (przy użyciu produkcji r). Zapisujemy ten fakt jako

$$\zeta\omega\xi \Rightarrow_G \zeta\omega'\xi\ .$$

Jeżeli jasne jest, o jaką gramatykę chodzi, symbol gramatyki możemy pominąć:

$$\zeta\omega\xi \Rightarrow \zeta\omega'\xi\ .$$

Definicja 2.16 (wyprowadzenie)**.** Niech $G = (V, T, R, S)$ będzie gramatyką oraz niech $\omega, \omega' \in (V \cup T)^*$. *Wyprowadzeniem* łańcucha ω' z łańcucha ω w gramatyce G nazywamy wówczas ciąg łańcuchów $\zeta_0, \zeta_1, \ldots, \zeta_m \in (V \cup T)^*$, $m \geqslant 1$, taki, że:

$$\zeta_0 = \omega,\ \zeta_m = \omega'\ ,$$
$$\zeta_0 \Rightarrow_G \zeta_1,\ \zeta_1 \Rightarrow_G \zeta_2,\ \ldots,\ \zeta_{m-1} \Rightarrow_G \zeta_m\ .$$

Liczbę m nazywamy *długością wyprowadzenia*.

Definicja 2.17 (wyprowadzalność)**.** Łańcuch $\omega' \in (V \cup T)^*$ nazywamy *wyprowadzalnym* z łańcucha $\omega \in (V \cup T)^*$ w gramatyce $G = (V, T, R, S)$, jeżeli istnieje wyprowadzenie łańcucha ω' z łańcucha ω w gramatyce G. Wyprowadzalność łańcucha ω' z ω zapisujemy następująco:

$$\omega \Rightarrow^*_G \omega'\ .$$

Jeżeli nie prowadzi to do niejasności, można pominąć symbol gramatyki i napisać po prostu

$$\omega \Rightarrow^* \omega' .$$

Relacja wyprowadzalności w gramatyce $G = (V, T, R, S)$ jest zwrotna, tj. dla każdego łańcucha $\omega \in (V \cup T)^*$ zachodzi związek

$$\omega \Rightarrow^*_G \omega .$$

Przykład 2.13 (wyprowadzenie)**.** Niech dana będzie gramatyka G z przykładu 2.12. W gramatyce tej łańcuch *jedzą myszy* jest wyprowadzalny z symbolu VP, co możemy zapisać jako

$$VP \Rightarrow^* \textit{jedzą myszy} .$$

Można się przekonać, że istotnie, wszystkie warunki ku temu są spełnione:

$$VP \overset{①}{\Rightarrow} V\ N \overset{②}{\Rightarrow} \textit{jedzą}\ N \overset{③}{\Rightarrow} \textit{jedzą myszy} ,$$

$$\begin{aligned} ① &\quad VP \to V\ N \in R , \\ ② &\quad V \to \textit{jedzą} \in R , \\ ③ &\quad N \to \textit{myszy} \in R . \end{aligned}$$

W dalszej części pracy, jeżeli będzie mowa o wyprowadzeniu pewnego łańcucha bez podania, z jakiego łańcucha został on wyprowadzony, będziemy przyjmować, że chodzi o wyprowadzenie z symbolu początkowego gramatyki.

Przykład 2.14 (wyprowadzenie z symbolu początkowego)**.** Niech dana będzie gramatyka G z przykładu 2.12. Łańcuch *koty jedzą myszy* jest wyprowadzalny z symbolu początkowego S gramatyki G:

$$S \Rightarrow^* \textit{koty jedzą myszy} .$$

Istotnie, wyprowadzeniem łańcucha *koty jedzą myszy* jest ciąg

$$\begin{aligned} S \Rightarrow N\ VP \Rightarrow N\ V\ N \Rightarrow \textit{koty}\ V\ N \Rightarrow \\ \Rightarrow \textit{koty jedzą}\ N \Rightarrow \textit{koty jedzą myszy} . \end{aligned}$$

Jeden łańcuch może posiadać kilka wyprowadzeń z danego symbolu (w tym również symbolu początkowego gramatyki):

Przykład 2.15 (różne wyprowadzenia tego samego łańcucha)**.** Niech dane będą: gramatyka bezkontekstowa z przykładu 2.12 i łańcuch *koty jedzą myszy* . Łańcuch ten jest wyprowadzalny z symbolu początkowego S, o czym

przekonaliśmy się w przykładzie 2.14, konstruując odpowiednie wyprowadzenie. Z drugiej strony, ciąg

$$S \Rightarrow VP\ N \Rightarrow N\ V\ N \Rightarrow koty\ V\ N \Rightarrow$$
$$\Rightarrow koty\ jedzą\ N \Rightarrow koty\ jedzą\ myszy\ .$$

także jest wyprowadzeniem łańcucha *koty jedzą myszy* z symbolu S. Widać zatem, że istotnie, jeden łańcuch może posiadać więcej niż jedno wyprowadzenie z danego symbolu.

Definicja 2.18 (forma zdaniowa)**.** Niech $G = (V, T, R, S)$ będzie gramatyką. Łańcuch $\omega \in (V \cup T)^*$ nazywamy *formą zdaniową* tej gramatyki, jeżeli

$$S \Rightarrow_G^* \omega\ .$$

Definicja 2.19 (język generowany przez gramatykę)**.** *Język generowany przez gramatykę* $G = (V, T, R, S)$ definiujemy jako zbiór form zdaniowych tej gramatyki złożonych z samych symboli końcowych i oznaczamy przez $L(G)$:

$$L(G) := \{w \in T^* \colon S \Rightarrow_G^* w\}\ .$$

Definicja 2.20 (język kontekstowy)**.** *Językiem kontekstowym* nazywamy język generowany przez pewną gramatykę kontekstową.

Definicja 2.21 (język bezkontekstowy)**.** *Językiem bezkontekstowym* nazywamy język generowany przez pewną gramatykę bezkontekstową.

Uwaga 2.3. Ponieważ żadna gramatyka bezkontekstowa (ϵ-wolna) nie generuje łańcucha pustego ϵ, zatem w świetle powyższej definicji żaden język bezkontekstowy nie zawiera łańcucha pustego ϵ.

Twierdzenie 2.2. *Każdy język bezkontekstowy jest językiem kontekstowym.*

Dowód. Teza wynika bezpośrednio z twierdzenia 2.1. □

Definicja 2.22 (gramatyki równoważne)**.** Gramatyki G_1 i G_2 nazywamy *równoważnymi*, jeżeli

$$L(G_1) = L(G_2)\ .$$

Twierdzenie 2.3. *Jeżeli G jest gramatyką bezkontekstową z ϵ-produkcjami, to $L(G) \setminus \{\epsilon\}$ jest językiem bezkontekstowym.*

Dowód. W [19] dowiedzione jest twierdzenie, że jeśli G jest gramatyką bezkontekstową z ϵ-produkcjami, to istnieje gramatyka bezkontekstowa ϵ-wolna, która generuje język $L(G) \setminus \{\epsilon\}$. □

Twierdzenie 2.4. *Każdy język bezkontekstowy jest generowany przez pewną gramatykę bezkontekstową w postaci normalnej Chomsky'ego.*

Dowód. Dowód powyższego faktu polega na odpowiednim przekształceniu zestawu reguł gramatyki bezkontekstowej, tak aby reguły niebędące w postaci normalnej Chomsky'ego zostały zastąpione przez odpowiednie reguły postaci $A \to BC$ lub $A \to a$. Algorytm, za pomocą którego można to uzyskać, został przedstawiony m.in. w [19]. □

Przykład 2.16 (język generowany przez gramatykę bezkontekstową)**.** Gramatyka bezkontekstowa G z przykładu 2.8 generuje język

$$L(G) = \{c, ab, acb, aabb, aacbb, aaabbb, \ldots\} .$$

Język $L(G)$ jest językiem bezkontekstowym.

Gramatyka bezkontekstowa w postaci normalnej Chomsky'ego z przykładu 2.11 generuje ten sam język.

Przykład 2.17 (język generowany przez gramatykę bezkontekstową)**.** Gramatyka bezkontekstowa G z przykładu 2.9 generuje język

$$L(G) = \{\, \textit{psy widzą}\,,\, \textit{psy widzą psy}\,,\, \textit{psy widzą koty}\,, \ldots \\ \ldots,\, \textit{jedzą myszy}\,,\, \textit{koty jedzą myszy}\,, \ldots\} .$$

Język $L(G)$ jest językiem bezkontekstowym.

Gramatyka bezkontekstowa w postaci normalnej Chomsky'ego z przykładu 2.12 generuje ten sam język.

2.3. Drzewa składniowe

Definicja 2.23 (drzewo)**.** *Drzewo* nad niepustym zbiorem $\mathcal{W}$ etykiet węzłów i niepustym zbiorem $\mathcal{E}$ etykiet krawędzi definiujemy jako obiekt t postaci $c[e_1 : t_1, \ldots, e_n : t_n]$, gdzie $t_1, \ldots, t_n$ są drzewami nad $\mathcal{W}$ i $\mathcal{E}$, natomiast c jest elementem zbioru $\mathcal{W}$.

Niech $t = (c, ((e_1, t_1), \ldots, (e_n, t_n)))$ będzie drzewem. Węzeł etykietowany jako c nazywamy *korzeniem* drzewa t i piszemy $c = \mathrm{root}(t)$. Drzewo postaci $(c, ())$ nazywamy *liściem*. Dla uproszczenia drzewo o korzeniu etykietowanym c i bezpośrednich poddrzewach $t_1, \ldots, t_n$ będziemy oznaczać przez $c[e_1 : t_1, \ldots, e_n : t_n]$ (zamiast $(c, ((e_1, t_1), \ldots, (e_n, t_n)))$).

Definicja 2.24 (drzewo składniowe)**.** Niech Σ, $\mathcal{C}$ and $\mathcal{R}$ będą skończonymi zbiorami takimi, że $\mathcal{C} \cap (\Sigma \times \mathbb{N}_+) = \emptyset$. *Drzewem składniowym* nad alfabetem Σ, zbiorem kategorii $\mathcal{C}$ i zbiorem ról składniowych $\mathcal{R}$ nazywamy dowolne drzewo $t \in \hat{\mathcal{T}}(\mathcal{C} \cup (\Sigma \times \mathbb{N}_+), \mathcal{R})$,[2] które spełnia następujące warunki:

[2]Symbol $\mathbb{N}_+$ oznacza zbiór liczb naturalnych dodatnich (czyli bez zera); zbiór liczb naturalnych wraz z zerem będę oznaczał przez $\mathbb{N}$.

— wszystkie etykiety liści są ze zbioru Σ, zaś wszystkie pozostałe etykiety są ze zbioru $\mathcal{C}$,
— dla dowolnego indeksu k istnieje co najwyżej jedna etykieta postaci (a, k) w drzewie t.

Zbiór wszystkich takich drzew oznaczamy przez $\mathcal{T}(\Sigma, \mathcal{C}, \mathcal{R})$.

Drzewo składniowe można traktować jako drzewo, którego wewnętrzne węzły są etykietowane kategoriami ($c \in \mathcal{C}$), a liście parami: symbol terminalny — indeks ($(a, k) \in \Sigma \times \mathbb{N}_+$). Krawędzie drzewa składniowego są etykietowane ich rolami składniowymi ($r \in \mathcal{R}$).

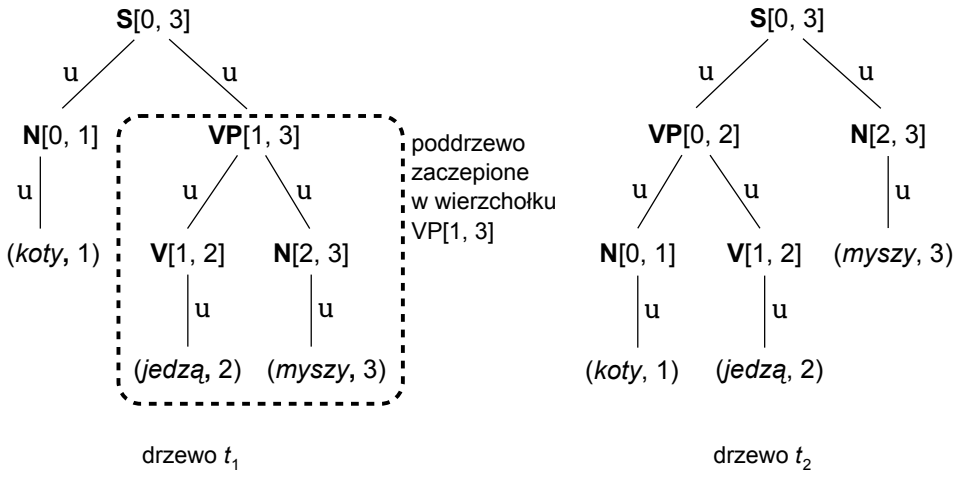
Definicja 2.25 (plon, baza)**.** Niech t będzie drzewem składniowym. Niech $\{(a_1, k_1), \ldots, (a_n, k_n)\}$, $k_1 < \ldots < k_n$, będzie zbiorem wszystkich etykiet liści. Wówczas:

1. Łańcuch $a_1 a_2 \ldots a_n$ będziemy nazywać *plonem* drzewa t i oznaczać przez $\mathrm{yield}(t)$.
2. Zbiór $\{k_1, \ldots, k_n\}$ będziemy nazywać *bazą* drzewa t i oznaczać przez $\sigma(t)$.

Definicja 2.26 (drzewo składniowe dla gramatyki bezkontekstowej)**.** Niech $G = (V, T, R, S)$ będzie gramatyką bezkontekstową. Drzewo t jest *drzewem składniowym* (lub *drzewem wyprowadzenia*) dla gramatyki G, jeżeli:
— $t \in \mathcal{T}(T, V, \{\mathsf{u}\})$, gdzie u jest specjalnym symbolem, oznaczającym nieistotną rolę składniową,
— korzeń drzewa t etykietowany jest symbolem początkowym S gramatyki G,
— jeśli synowie wierzchołka drzewa t etykietowanego symbolem A mają kolejno etykiety $X_1, X_2, \ldots, X_k \in V \cup T$, to $A \to X_1 X_2 \ldots X_k$ jest produkcją ze zbioru R.

Definicja 2.27 (poddrzewo)**.** *Poddrzewem* drzewa składniowego Γ zaczepionym w wierzchołku x o etykiecie A nazywamy drzewo złożone ze wszystkich potomków wierzchołka x wraz z łączącymi je łukami.

Różne wierzchołki danego drzewa składniowego mogą być etykietowane tymi samymi symbolami gramatyki. Aby odróżnić różne wierzchołki o tych samych etykietach, wprowadzamy następującą umowę. Niech $G = (V, T, R, S)$ będzie gramatyką bezkontekstową. Niech Γ będzie drzewem wyprowadzenia łańcucha $w = w_1 w_2 \ldots w_n \in T^+$ w gramatyce G. Jeżeli plonem poddrzewa zaczepionego w wierzchołku x o etykiecie $X \in V \cup T$ jest łańcuch $w_{i+1} \ldots w_j \in T^+$, to wierzchołek x będziemy oznaczać przez $X[i, j]$. Parę liczb (i, j) będziemy nazywać *zakresem* wierzchołka x.



Rysunek 2.1. Przykładowe drzewa składniowe łańcucha *koty jedzą myszy* . Wyróżniono poddrzewo drzewa t_1 zaczepione w wierzchołku $VP\,[1,3]$

Przykład 2.18 (drzewo składniowe)**.** Niech dana będzie gramatyka G z przykładu 2.12. Dwa przykładowe drzewa składniowe łańcucha *koty jedzą myszy* pokazane są na rysunku 2.1. Wierzchołki oznaczono za pomocą etykiet i zakresów. Wyróżniono poddrzewo drzewa t_1 zaczepione w wierzchołku $VP\,[1,3]$.

Twierdzenie 2.5. *Niech $G = (V, T, R, S)$ będzie gramatyką bezkontekstową. Wówczas łańcuch $\omega \in (V \cup T)^+$ jest wyprowadzalny z symbolu początkowego gramatyki G wtedy i tylko wtedy, gdy ω jest plonem pewnego drzewa składniowego w gramatyce G.*

Dowód. Dowód powyższego twierdzenia jest przedstawiony w [19]. □

Definicja 2.28 (wyprowadzenie lewostronne)**.** Niech $G = (V, T, R, S)$ będzie gramatyką bezkontekstową. *Wyprowadzeniem lewostronnym* nazywamy takie wyprowadzenie $\zeta_0, \zeta_1, \ldots, \zeta_m \in (V \cup T)^+$, w którym w każdym kroku dokonujemy zastąpienia symbolu nieterminalnego leżącego najbardziej na lewo. Innymi słowy, dla każdego $i \in \{1, \ldots, m\}$ zachodzi:

$$\zeta_{i-1} = uA\xi \Rightarrow u\omega\xi = \zeta_i$$

dla pewnych $u \in T^*$, $A \in V$, $\xi \in (V \cup T)^*$, $\omega \in (V \cup T)^+$ oraz dla $A \to \omega \in R$.

Wyprowadzenie lewostronne łańcucha $\omega \in (V \cup T)^+$ z symbolu początkowego S gramatyki bezkontekstowej $G = (V, T, R, S)$ będziemy nazywać krótko *wyprowadzeniem lewostronnym łańcucha* ω. Ilekroć będzie mowa o wyprowadzeniu lewostronnym łańcucha ω z symbolu (bądź łańcucha) innego niż symbol początkowy, będzie to wyraźnie zaznaczone.

Jeden łańcuch może posiadać kilka różnych wyprowadzeń lewostronnych.

Przykład 2.19. Niech dana będzie gramatyka G z przykładu 2.12. Jednym z wyprowadzeń lewostronnych łańcucha *koty jedzą myszy* w gramatyce G jest ciąg

$$\begin{aligned} S \Rightarrow\ & N\ VP \Rightarrow \textit{koty}\ VP \Rightarrow \textit{koty}\ V\ N \Rightarrow \\ & \Rightarrow \textit{koty jedzą}\ N \Rightarrow \textit{koty jedzą myszy}\ . \end{aligned}$$

Innym wyprowadzeniem lewostronnym tego samego łańcucha jest ciąg

$$\begin{aligned} S \Rightarrow\ & VP\ N \Rightarrow N\ V\ N \Rightarrow \textit{koty}\ V\ N \Rightarrow \\ & \Rightarrow \textit{koty jedzą}\ N \Rightarrow \textit{koty jedzą myszy}\ . \end{aligned}$$

Widać więc, że istotnie jeden łańcuch może mieć więcej niż jedno wyprowadzenie lewostronne.

Twierdzenie 2.6. *Niech $G = (V, T, R, S)$ będzie gramatyką bezkontekstową oraz $\omega \in T^+$. Wówczas następujące warunki są równoważne:*

(1) $S \Rightarrow^* \omega$.
(2) Istnieje drzewo składniowe w gramatyce G o plonie ω.
(3) Istnieje wyprowadzenie lewostronne łańcucha ω w gramatyce G.

Dowód. Dowód można znaleźć w [19]. □

Jeżeli w gramatyce bezkontekstowej $G = (V, T, R, S)$ łańcuch $\omega \in (V \cup T)^+$ posiada wyprowadzenie lewostronne, to istnieje drzewo składniowe w gramatyce G o plonie ω.

2.4. Probabilistyczne gramatyki bezkontekstowe (PCFG)

Definicja 2.29 (probabilistyczna gramatyka bezkontekstowa)**.** *Probabilistyczną gramatyką bezkontekstową* (PCFG, ang. *probabilistic context-free grammar*) nazywamy uporządkowaną piątkę (V, T, R, S, P), w której:

— czwórka (V, T, R, S) tworzy gramatykę bezkontekstową,
— $P\colon R \to [0, 1]$ jest funkcją spełniającą warunek:

$$\sum_{(A \to w) \in R} P(A \to w) \ = \ 1 \quad \text{dla każdego } A \in V \tag{2.1}$$

Funkcję P nazywamy *prawdopodobieństwem reguły.*

Ponieważ w każdej probabilistycznej gramatyce bezkontekstowej (V, T, R, S, P) czwórka (V, T, R, S) jest (nieprobabilistyczną) gramatyką bezkontekstową, zatem definicje postaci normalnej Chomsky'ego, wyprowadzeń, wyprowadzalności, języka generowanego czy drzewa składniowego dla PCFG niczym

nie różną się od analogicznych definicji dla CFG. Podobnie rzecz ma się, jeśli chodzi o twierdzenia i własności.

Przykład 2.20 (PCFG)**.** Przykład probabilistycznej gramatyki bezkontekstowej $G = (V, T, R, S, P)$, która powstała z gramatyki bezkontekstowej z przykładu 2.12 przez dopisanie prawdopodobieństw reguł:

— $V = \{ S, N, VP, V \}$,

— $T = \{ \textit{psy}, \textit{koty}, \textit{myszy}, \textit{widzą}, \textit{jedzą} \}$,

— $R = \{ S \rightarrow N\ VP,\ S \rightarrow VP\ N,\ VP \rightarrow V\ N,\ VP \rightarrow N\ V,\ VP \rightarrow \textit{widzą},\ VP \rightarrow \textit{jedzą},\ N \rightarrow \textit{psy},\ N \rightarrow \textit{koty},\ N \rightarrow \textit{myszy},\ V \rightarrow \textit{jedzą},\ V \rightarrow \textit{widzą} \}$,

— $S \in V$ jest symbolem początkowym,

— prawdopodobieństwa reguł:

$$
\begin{aligned}
&P(\, S \rightarrow N\ VP \,) = 0.8\ , \\
&P(\, S \rightarrow VP\ N \,) = 0.2\ , \\
&P(\, VP \rightarrow V\ N \,) = 0.4\ , \\
&P(\, VP \rightarrow N\ V \,) = 0.1\ , \\
&P(\, VP \rightarrow \textit{widzą} \,) = 0.15\ , \\
&P(\, VP \rightarrow \textit{jedzą} \,) = 0.35\ , \\
&P(\, N \rightarrow \textit{psy} \,) = 0.25\ , \\
&P(\, N \rightarrow \textit{koty} \,) = 0.5\ , \\
&P(\, N \rightarrow \textit{myszy} \,) = 0.25\ , \\
&P(\, V \rightarrow \textit{jedzą} \,) = 0.3\ , \\
&P(\, V \rightarrow \textit{widzą} \,) = 0.7\ .
\end{aligned}
$$

Definicja 2.30 (prawdopodobieństwo wyprowadzenia lewostronnego)**.** *Prawdopodobieństwem wyprowadzenia lewostronnego* nazywamy iloczyn prawdopodobieństw wszystkich reguł użytych w tym wyprowadzeniu lewostronnym.

Każdemu drzewu składniowemu możemy jednoznacznie przyporządkować wyprowadzenie lewostronne, stąd następująca definicja:

Definicja 2.31 (prawdopodobieństwo drzewa składniowego)**.** *Prawdopodobieństwo drzewa składniowego* definiujemy jako prawdopodobieństwo wyprowadzenia lewostronnego odpowiadającego temu drzewu.

Prawdopodobieństwo drzewa t będziemy oznaczać przez $\mathbf{P}(t)$.

Definicja 2.32 (drzewo Viterbiego)**.** *Drzewem Viterbiego* dla danego łańcucha nazywamy to spośród wszystkich drzew składniowych dla tego łańcucha, którego prawdopodobieństwo jest największe.

Definicja 2.33 (prawdopodobieństwo łańcucha)**.** *Prawdopodobieństwem łańcucha* nazywamy sumę prawdopodobieństw wszystkich drzew składniowych tego łańcucha.

Przykład 2.21 (prawdopodobieństwo drzewa, drzewo Viterbiego, prawdopodobieństwo łańcucha)**.** Niech dana będzie probabilistyczna gramatyka bezkontekstowa $G = (V, T, R, S, P)$ z przykładu 2.20 oraz przykładowe drzewa składniowe t_1 i t_2 z przykładu 2.18. Ponieważ drzewa t_1 i t_2 zostały skonstruowane według reguł gramatyki (V, T, R, S) z przykładu 2.12, na której zbudowano probabilistyczną gramatykę bezkontekstową G, więc drzewa te są również zgodne z regułami gramatyki G.

Drzewu t_1 odpowiada wyprowadzenie lewostronne

$$\begin{aligned} S \Rightarrow\ & N\ VP \Rightarrow \textit{koty}\ VP \Rightarrow \textit{koty}\ V\ N \Rightarrow \\ & \Rightarrow \textit{koty jedzą}\ N \Rightarrow \textit{koty jedzą myszy}\ . \end{aligned}$$

Stąd

$$\mathbf{P}(t_1) = 0.8 \cdot 0.5 \cdot 0.4 \cdot 0.3 \cdot 0.25 = 0.012\ . \tag{2.2}$$

Drzewu t_2 odpowiada wyprowadzenie lewostronne

$$\begin{aligned} S \Rightarrow\ & VP\ N \Rightarrow N\ V\ N \Rightarrow \textit{koty}\ V\ N \Rightarrow \\ & \Rightarrow \textit{koty jedzą}\ N \Rightarrow \textit{koty jedzą myszy}\ . \end{aligned}$$

Stąd

$$\mathbf{P}(t_2) = 0.2 \cdot 0.1 \cdot 0.5 \cdot 0.3 \cdot 0.25 = 0.00075\ . \tag{2.3}$$

Widać, że $\mathbf{P}(t_1) > \mathbf{P}(t_2)$, a zatem t_1 jest drzewem Viterbiego dla łańcucha *koty jedzą myszy* .

Nietrudno również obliczyć prawdopodobieństwo całego łańcucha, które wynosi

$$\mathbf{P}(\textit{ koty jedzą myszy }) = \mathbf{P}(t_1) + \mathbf{P}(t_2) = 0.012 + 0.00075 = 0.01275\ .$$

Rozdział 3

Przegląd istniejących formalizmów opisu języków o szyku swobodnym

Swobodny szyk wyrazów i obecność nieciągłości syntaktycznych jest cechą wielu języków, w szczególności języków fleksyjnych. Szereg przykładów z języków ukraińskiego i nowogreckiego można znaleźć w [29]. W języku polskim mogą pojawić się następujące typy nieciągłości:

A. Przydawka przymiotna jest oddzielona od rzeczownika, którego określa: ***Ładną*** *masz* ***sukienkę***.
B. Pytajny zaimek przymiotny jest oddzielony od rzeczownika, którego dotyczy: ***Która*** *jest* ***godzina****?*
C. Rzeczownik jest oddzielony od przydawki dopełniaczowej, która go określa ***Samochodem*** *jeżdżę* ***brata***.
D. Rzeczownik jest oddzielony od frazy przyimkowej, która go określa: ***Z matematyki*** *nie mam* ***podręcznika***.
E. Przysłówek jest oddzielony od przymiotnika, którego określa: ***Bardziej*** *nie można być* ***niemiłym***.
F. Składniki orzeczenia w czasie przyszłym złożonym oddzielone są od siebie przez podmiot: *Czy* ***będzie*** *ktoś* ***odwiedzać*** *mnie?*
G. Składniki orzeczenia w czasie przyszłym złożonym oddzielone są od siebie przez dopełnienie: *Czy* ***będziesz*** *mnie często* ***odwiedzać****?*
H. Partykuła jest oddzielona od czasownika: *Ja* ***bym*** *tak nigdy nie* ***postąpił!***
I. Dopełnienie jest oddzielone od rządzącego nim czasownika: *Takich* ***filmów*** *nie będę* ***oglądać***.
J. Bezokolicznik jest oddzielony od rządzącego nim czasownika: ***Kazał*** *was król* ***przyprowadzić***.
K. Zaimek zwrotny jest oddzielony od czasownika: *Jan* ***się*** *nie lubi* ***myć***.

Problem opisu języków o swobodnym szyku wyrazów był szczegółowo badany i został szeroko opisany w literaturze. Niniejszy rozdział zawiera przegląd wybranych rozwiązań. Różnorodne podejścia do problemu często wymagały jeśli nie sformalizowania, to przynajmniej autorskiego ujednolicenia i przedstawienia w spójny sposób tak, aby łatwiej można było porównać je między sobą.

Zawartość niniejszego rozdziału została pogrupowana tematycznie w podrozdziały. Podrozdział 3.1 przedstawia klasyczne formalizmy, które umożliwiają opis swobodnego szyku. Podrozdział 3.2 opisuje, jak kwestia nieciągłości syntaktycznych została rozwiązana w wybranych bankach drzew. Podrozdział 3.3 poświęcony jest wielojęzycznemu formalizmowi Grammatical Framework, natomiast w podrozdziale 3.4 zostały przedstawione formalizmy stworzone głównie z myślą o opisie języka polskiego.

3.1. Formalizmy oparte na gramatykach bezkontekstowych umożliwiające opis swobodnego szyku

3.1.1. Nieuporządkowane gramatyki bezkontekstowe (UCFG)[1]

Nieuporządkowane gramatyki bezkontekstowe (ang. *unordered context free grammars*, UCFG) definiuje się tak samo, jak gramatyki bezkontekstowe (Definicja 2.12), lecz inaczej określa się w nich relację wyprowadzalności, a co za tym idzie — język generowany przez gramatykę.

Definicja 3.1 (bezpośrednia wyprowadzalność w nieuporządkowanej gramatyce bezkontekstowej)**.** Niech $G = (V, T, R, S)$ będzie dowolną nieuporządkowaną gramatyką bezkontekstową oraz niech $\zeta, \xi, \omega \in (V \cup T)^*$ i $A \in V$. Jeżeli istnieje produkcja $r = (A \to \omega) \in R$, to każdy łańcuch $\zeta\omega'\xi$ taki, że ω' powstaje z ω przez zmianę kolejności symboli, nazywamy *bezpośrednio wyprowadzalnym* z łańcucha $\zeta A\xi$ w nieuporządkowanej gramatyce bezkontekstowej G (przy użyciu produkcji r). Zapisujemy ten fakt jako

$$\zeta A\xi \Rightarrow_G \zeta\omega'\xi \ .$$

Definicje wyprowadzenia, wyprowadzalności i języka generowanego przez nieuporządkowaną gramatykę bezkontekstową są identyczne z odpowiednimi definicjami dla gramatyk uporządkowanych (Definicje 2.16, 2.17 i 2.19), z zastrzeżeniem, iż pojęcie bezpośredniej wyprowadzalności w nich wykorzystywane zdefiniowane jest jak wyżej.

Każdy język generowany przez daną nieuporządkowaną gramatykę bezkontekstową może zostać wygenerowany przez pewną (uporządkowaną) gramatykę bezkontekstową:

[1] Na podstawie [2]

Twierdzenie 3.1. *Jeżeli G jest nieuporządkowaną gramatyką bezkontekstową, to istnieje (uporządkowana) gramatyka bezkontekstowa G' taka, że*

$$L(G') = L(G) \ .$$

Dowód. Wystarczy skonstruować gramatykę G' zastępując każdą produkcję $A \to \omega$ gramatyki G przez zbiór reguł $A \to \omega'$, gdzie ω' przebiega wszystkie permutacje łańcucha ω. □

Przykład 3.1 (Nieuporządkowana gramatyka bezkontekstowa)**.** Przykład nieuporządkowanej gramatyki bezkontekstowej $G = (V, T, R, S)$:

— $V = \{ S, N, NP, V, VP, A \}$,

— $T = \{ \textit{psy}, \textit{koty}, \textit{myszy}, \textit{widzą}, \textit{jedzą}, \textit{czarne}, \textit{białe}, \textit{bure} \}$,

— $R = \{ S \to NP\ VP,\ NP \to A\ N,\ NP \to N,\ VP \to V\ NP,$ $VP \to V,\ N \to \textit{psy},\ N \to \textit{koty},\ N \to \textit{myszy},\ V \to \textit{jedzą},$ $V \to \textit{widzą},\ A \to \textit{czarne},\ A \to \textit{białe},\ A \to \textit{bure} \}$,

— $S \in V$ jest symbolem początkowym.

Do języka generowanego przez G należą zdania takie, jak np.: *bure koty jedzą białe myszy* czy *psy czarne czarne koty widzą*.

Problem parsingu nieuporządkowanych gramatyk bezkontekstowych należy do klasy problemów NP-zupełnych. [2]

3.1.2. Gramatyki ID/LP[2]

Gramatyki ID/LP (ang. *immediate dominance* 'bezpośrednia dominacja', *linear precedence* 'liniowy porządek') stanowią rozszerzenie nieuporządkowanych gramatyk bezkontekstowych.

Występują w nich reguły dwojakiego rodzaju:

— reguły dominacji (ID) — nie różnią się niczym od produkcji używanych w nieuporządkowanych gramatykach bezkontekstowych i służą wskazaniu, jakie symbole są bezpośrednio wyprowadzalne z danego symbolu, bez określania ich kolejności,

— reguły porządku (LP) — ograniczają dowolność kolejności symboli.

Definicja 3.2 (gramatyka ID/LP)**.** *Gramatyką ID/LP* nazywamy uporządkowaną piątkę $G = (V, T, R_{\mathrm{ID}}, R_{\mathrm{LP}}, S)$, w której:

— czwórka $(V, T, R_{\mathrm{ID}}, S)$ tworzy nieuporządkowaną gramatykę bezkontekstową,

— R_{LP} jest zbiorem napisów postaci $A \prec B$, $A, B \in V$.

Zbiór R_{ID} nazywamy zbiorem *reguł dominacji*, zaś R_{LP} nazywamy zbiorem *reguł porządku* gramatyki G.

[2] Na podstawie [2, 12]

Definicja 3.3 (bezpośrednia wyprowadzalność w gramatyce ID/LP)**.** Niech $G = (V, T, R_{\mathrm{ID}}, R_{\mathrm{LP}}, S)$ będzie dowolną gramatyką ID/LP oraz $\zeta, \xi, \omega \in (V \cup T)^*$ i $A \in V$. Mówimy, że łańcuch $\zeta\omega'\xi$ jest *bezpośrednio wyprowadzalny* z łańcucha $\zeta A\xi$ w gramatyce ID/LP G, jeżeli spełnione są następujące warunki:

— istnieje reguła dominacji $(A \to \omega) \in R_{\mathrm{ID}}$ taka, że ω' powstaje z ω przez zmianę kolejności symboli,

— dla każdej reguły porządku $(X \prec Y) \in R_{\mathrm{LP}}$, jeżeli symbole X i Y występują w łańcuchu ω', to symbol X występuje po lewej stronie symbolu Y w tym łańcuchu.

Zapisujemy ten fakt jako

$$\zeta A\xi \Rightarrow_G \zeta\omega'\xi \ .$$

Przykład 3.2 (Gramatyka ID/LP)**.** $G = (V, T, R_{\mathrm{ID}}, R_{\mathrm{LP}}, S)$ jest przykładem gramatyki ID/LP, która powstała z nieuporządkowanej gramatyki bezkontekstowej z przykładu 3.1 przez dopisanie reguł porządku:

— $V = \{ S, N, NP, V, VP, A \}$,

— $T = \{ \textit{psy}, \textit{koty}, \textit{myszy}, \textit{widzą}, \textit{jedzą}, \textit{czarne}, \textit{białe}, \textit{bure} \}$,

— $R_{\mathrm{ID}} = \{ S \to NP\ VP,\ NP \to A\ N,\ NP \to N,\ VP \to V\ NP,$ $VP \to V,\ N \to \textit{psy},\ N \to \textit{koty},\ N \to \textit{myszy},\ V \to \textit{jedzą},$ $V \to \textit{widzą},\ A \to \textit{czarne},\ A \to \textit{białe},\ A \to \textit{bure} \}$,

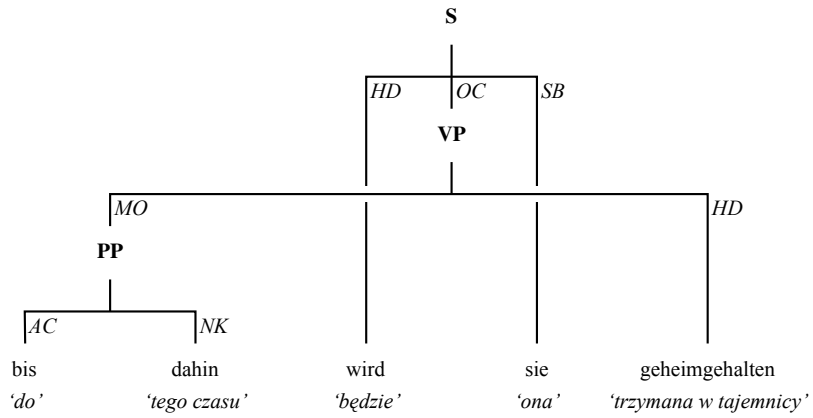
— $R_{\mathrm{LP}} = \{A \prec N\}$,

— $S \in V$ jest symbolem początkowym.

Język generowany przez gramatykę G jest bardzo podobny do języka generowanego przez gramatykę z przykładu 3.1, ale zawiera tylko zdania, w których rzeczowniki występują po określających je przymiotnikach. Dlatego wśród zdań języka $L(G)$ znajduje się zdanie *bure koty jedzą białe myszy*, ale już zdanie *psy czarne czarne koty widzą* nie należy do tego języka.

Gramatyki ID/LP stanowią pewien kompromis pomiędzy (uporządkowanymi) gramatykami bezkontekstowymi a nieuporządkowanymi gramatykami bezkontekstowymi. Z jednej strony zapewniają swobodę szyku fraz, a z drugiej strony umożliwiają nakładanie ograniczeń na tenże szyk.

Podobnie jak problem parsingu nieuporządkowanych gramatyk bezkontekstowych, problem parsingu gramatyk ID/LP również należy do klasy problemów NP-zupełnych. [2]



Rysunek 3.1. Przykład drzewa z krzyżującymi się gałęziami

3.2. Rozszerzenia gramatyk bezkontekstowych umożliwiające opis nieciągłości pojawiających się w bankach drzew

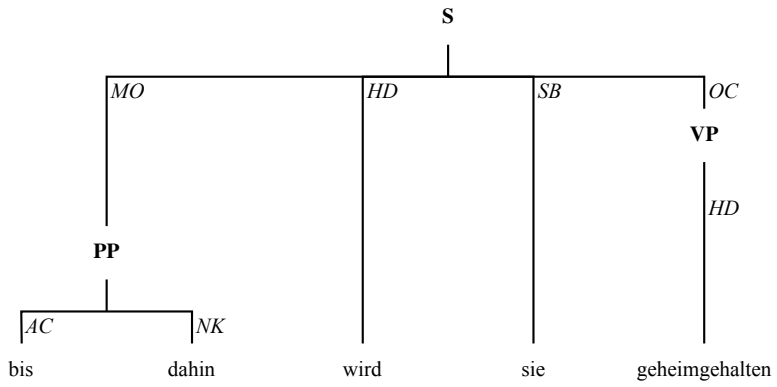
3.2.1. Reprezentacja struktury zdań w bankach drzew

Bank drzew to korpus składający się ze sparsowanych zdań. Struktura sparsowanych zdań zazwyczaj przedstawiana jest w postaci drzew składniowych — stąd nazwa. Na ogół dla konkretnego banku są to drzewa składniowe dla ustalonej gramatyki. Używa się w tym celu różnego rodzaju gramatyk — niektóre banki drzew oparte są o gramatyki zależnościowe, inne o gramatyki struktur frazowych (np. gramatyki bezkontekstowe). W przypadku banków drzew opartych o gramatyki bezkontekstowe stworzonych dla języków, w których występują nieciągłości syntaktyczne, konieczne jest zaadaptowanie użytej gramatyki tak, aby nieciągłości te mogły być reprezentowane.

Nieciągłości syntaktyczne zdań mogą być wielorako reprezentowane w bankach drzew. W kolejnych sekcjach nieco szczegółowiej przedstawione zostaną trzy metody: *node-raising* [31], *node-splitting* [3] i *node-adding* [21].

3.2.2. *Node-raising*

NEGRA [57] jest bankiem drzew dla niemieckiego. Zawiera ponad 20 000 zdań (350 000 słów) niemieckiego języka pisanego.języka Format drzew w banku NEGRA jest w zasadzie taki jak w gramatyce bezkontekstowej, lecz w celu reprezentacji składników nieciągłych dopuszcza się krzyżowanie gałęzi drzewa (rys. 3.1). Czasami również zdarza się, że pojedyncze zdanie jest reprezentowane przez kilka drzew. W notacji parsera NEGRA nie istnieje wyróżniony symbol, który mógłby być symbolem początkowym gramatyki.



Rysunek 3.2. Drzewo z rysunku 3.1 po zastosowaniu procedury *node-raising*

Kübler [31] opisuje, jak sprowadzić ten formalizm do gramatyki bezkontekstowej. Aby każde zdanie miało przyporządkowane dokładnie jedno drzewo, dodawany jest symbol początkowy w ten sposób, aby stał się korzeniem nowego drzewa. Korzenie dotychczasowych drzew stają się synami symbolu początkowego.

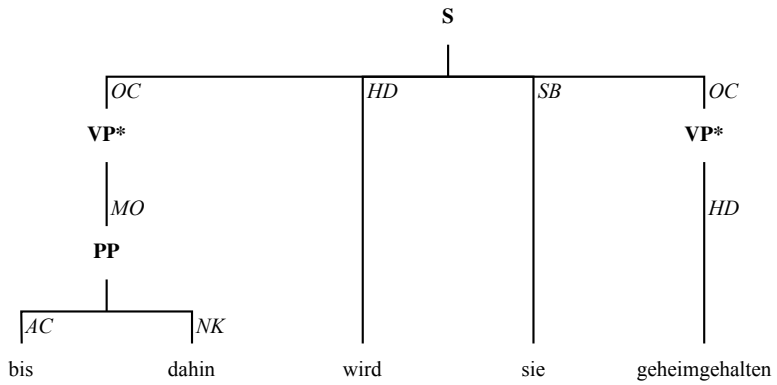
Krzyżujące się gałęzie zostają usunięte przy użyciu metody nazywanej *node-raising*: składniki połączone gałęzią, która krzyżuje się z inną, zostają odizolowane, a następnie zostają dołączone wyżej, do węzła nadrzędnego, w ten sposób, aby nie było potrzeba krzyżować gałęzi (rys. 3.2).

Zaletą tej metody jest to, że liczba węzłów drzewa zostaje zachowana. Niestety, wada tego podejścia jest taka, że procedura odwrotna — zamiana drzewa pozbawionego w ten sposób krzyżujących się gałęzi na wyjściowe drzewo z krzyżującymi się gałęziami — jest skomplikowana. Nie zostają też w pełni zachowane zależności między poszczególnymi składnikami, co sprawia, że uzyskana gramatyka bezkontekstowa nie jest w pełni równoważna formalizmowi wyjściowemu.

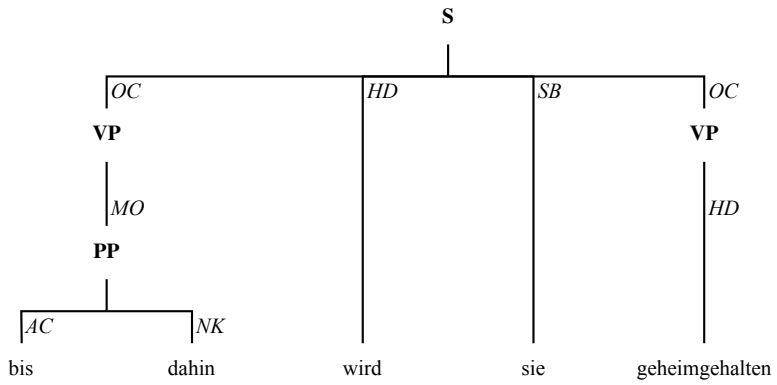
3.2.3. *Node-splitting*

W celu eliminacji wad procedury *node-raising* Boyd [3] proponuje inną metodę, którą nazywa *node-splitting*. W metodzie tej każdy węzeł o nieciągłym plonie zostaje podzielony na węzły cząstkowe odpowiadające ciągłym fragmentom plonu wyjściowego węzła (rys. 3.3). W ten sposób zależności między składnikami zostają zachowane.

Aby ułatwić procedurę odwrotną, węzły cząstkowe są odpowiednio oznaczane. Niekiedy w jednym zdaniu występują dwa nieciągłe węzły opatrzone tą samą etykietą — w takim przypadku odwrócenie przekształcenia może być utrudnione.



Rysunek 3.3. Drzewo z rysunku 3.1 po zastosowaniu procedury *node-splitting*



Rysunek 3.4. Drzewo z rysunku 3.1 po zastosowaniu procedury *node-adding*

3.2.4. *Node-adding*

Opracowana przez Hsu [21] metoda *node-adding* jest zmodyfikowaną wersją procedury *node-splitting*. Zamiast dzielić nieciągły węzeł na węzły cząstkowe, węzeł wyjściowy jest powielany dla każdego ciągłego fragmentu jego plonu (rys. 3.4). Dzięki temu nie potrzeba wprowadzać nowych symboli.

3.3. Wielojęzyczny formalizm *Grammatical Framework*[3]

Grammatical Framework (GF) jest formalizmem, który można opisywać wielorako:

— jako formalizm z gatunku gramatyk kategorialnych, jak CCG (*combinatory categorial grammars*) czy ACG (*abstract categorial grammars*),
— jako język do programowania gramatyk, jak YACC czy Bison,
— jako funkcyjny język programowania, jak Haskell czy Lisp,

[3]Na podstawie [43]

— jako framework do przetwarzania języków naturalnych,
— jako system automatycznego dowodzenia twierdzeń.

GF jest oparty na *Logical Framework* (LF) [17] — formalizmie opisu systemów logiki formalnej. LF umożliwia m.in. budowanie dowodów formalnych i ich weryfikację. GF w zamyśle twórców stanowi rozszerzenie systemu LF o możliwość definiowania składni konkretnej, gdy dana jest składnia abstrakcyjna.

Aarne Ranta jako źródła inspiracji do stworzenia *Grammatical Framework* wskazuje prace Haskella Curry'ego i Richarda Montague'a. W latach 60. XX w. Curry wprowadził rozróżnienie między strukturami gramatycznymi odpowiedzialnymi za podział wyrażeń na elementy znaczeniowe (uniwersalnymi) a strukturami odpowiedzialnymi za ich formę (charakterystycznymi dla danego języka). Curry zaproponował również projekt gramatyki wielojęzycznej, która opisywałaby oba rodzaje struktur dla wielu języków równocześnie, opartej na logice kombinatorycznej (rachunek CL [63]). Curry zwracał również uwagę na różnice między językami naturalnymi a formalnymi, takie jak np. fakt, że jedna struktura głęboka (semantyczna) może posiadać wiele reprezentacji powierzchniowych. Pomysły Curry'ego były rozwijane przez Montague'a z wykorzystaniem logiki modalnej. Te wszystkie aspekty przyświecały twórcom formalizmu GF.

Adam Slaski w swojej pracy magisterskiej [59] dokonał próby zastosowania formalizmu GF do opisu języka polskiego. Zaimplementowany został jedynie fragment języka polskiego. Jak sam autor pracy stwierdza, uzyskanego parsera nie można uznać za w pełni funkcjonalne narzędzie, zarówno ze względów poprawnościowych, jak i wydajnościowych.

Przykład 3.3 (prosta gramatyka w GF)**.** Poniżej podany jest przykład prostej gramatyki w GF, która generuje zdania typu: *Ala ma kota*, *Jacek ma psa*, *Dzieci mają rybki*.

Plik gramatyki abstrakcyjnej — wspólnej dla wszystkich języków:

```
abstract Sentence = {
    flags
        startcat = Phrase ;

    cat
        Phrase ; Subject ; Object ;

    fun
        Ma : Subject -> Object -> Phrase ;
        Ala, Jacek, Dzieci : Subject ;
        Kot, Pies, Rybki : Object ;
}
```

Plik gramatyki konkretnej dla języka polskiego:

```
concrete SentencePl of Sentence = {

    lincat
        Phrase = {s : Str} ;
        Subject = {s : Str ; n : Number} ;
        Object = {s : Str} ;

    lin
        Ma subject object = {
            s = subject.s ++ miec subject.n ++ object.s
        } ;
        Ala = {s = "Ala" ; n = Sg} ;
        Jacek = {s = "Jacek" ; n = Sg} ;
        Dzieci = {s = "Dzieci" ; n = Pl} ;
        Kot = {s = "kota"} ;
        Pies = {s = "psa"} ;
        Rybki = {s = "rybki"} ;

    param
        Number = Sg | Pl ;

    oper
        miec : Number -> Str =
            \n -> case n of {
                Sg => "ma" ;
                Pl => "mają"
            } ;

}
```

Plik gramatyki abstrakcyjnej opisuje zależności pomiędzy kategoriami gramatyki. W podanym przykładzie kategorią startową (`startcat`) jest zdanie (`Phrase`). Po słowie kluczowym `cat` wymienione są kategorie gramatyki. Po słowie kluczowym `fun` opisane są typy poszczególnych funktorów (takich jak np. `Ma`) i innych elementów.

Plik gramatyki konkretnej opisuje, w jaki sposób gramatyka abstrakcyjna realizowana jest dla danego języka. Zdefiniowane są parametry (`param`) i operatory (`oper`) oraz opisane zasady konstruowania wyrażeń i uzgadniania wartości parametrów.

3.4. Formalizmy stworzone głównie z myślą o języku polskim i innych językach słowiańskich

3.4.1. Formalizm FROG[4]

Formalizm FROG (*free order definite clause grammar*, 'DCG szyku swobodnego') został po raz pierwszy przedstawiony w [66], jako rozszerzenie formalizmu DCG.

Definicja 3.4 (Gramatyka FROG)**.** *Gramatyką FROG* nazywamy uporządkowaną piątkę $G = (V, T, R, S, \mathcal{R})$, w której:

(1) V jest alfabetem, nazywanym *alfabetem symboli pomocniczych (nieterminalnych)*,
(2) T jest rozłącznym z V alfabetem, nazywanym *alfabetem symboli końcowych (terminalnych)*,
(3) wyróżniony symbol $S \in V$ nazywany jest *symbolem początkowym*,
(4) $\mathcal{R}$ jest zbiorem, nazywanym *zbiorem ról składniowych*,
(5) R jest zbiorem *reguł*, czyli napisów jednej z trzech następujących postaci:
   (i) $A \to \epsilon$, gdzie $A \in V$ (*ϵ-produkcje*),
   (ii) $A \to r : w$, gdzie $A \in V$, $w \in T$, $r \in \mathcal{R}$ (*reguły leksykalne*),
   (iii) $A^{(*)} \xrightarrow{(*)(\sim)} r_1 : B_1^{(*)} \dots r_n : B_n^{(*)}$, gdzie $A, B_1, \dots, B_n \in V$, $r_1, \dots r_n \in \mathcal{R}$; przy czym symbole w nawiasach, czyli $(*)$ oraz $(\sim)$, występują w regułach opcjonalnie; przy symbolu $\to$ nie może wystąpić jednocześnie $*$ i $\sim$; ponadto $\sim$ może wystąpić jedynie wtedy, gdy $n = 2$.

Gramatyki FROG przypominają nieuporządkowane gramatyki bezkontekstowe (z ϵ-produkcjami), do których dołączono pewne ograniczenia na szyk składników. Reguły $A \to \epsilon$ odpowiadają produkcjom $A \to \epsilon$ gramatyk bezkontekstowych, reguły $A \to r : w$ — produkcjom $A \to w$, zaś reguły $A \to r_1 : B_1 \dots r_n : B_n$ (bez symboli $*$ ani $\sim$) — produkcjom $A \to B_1 \dots B_n$. Istotna różnica między regułami gramatyki bezkontekstowej a regułami FROG jest taka, że reguły FROG opisują dodatkowo role składniowe, podczas gdy dla gramatyk bezkontekstowych pojęcie ról składniowych nie jest w ogóle zdefiniowane.

Dodatkowe ograniczenia na szyk składników są opisywane przez reguły z symbolami $*$ i $\sim$:

1. Obecność symbolu $*$ przy symbolu nieterminalnym (np. A^*) oznacza, że dany symbol nieterminalny odnosi się do ciągłego składnika, czyli wyraże-

[4]Na podstawie [66], [15]

nia złożonego z kolejnych symboli terminalnych. Symbol $*$ może pojawić się zarówno po lewej, jak i po prawej stronie reguły.

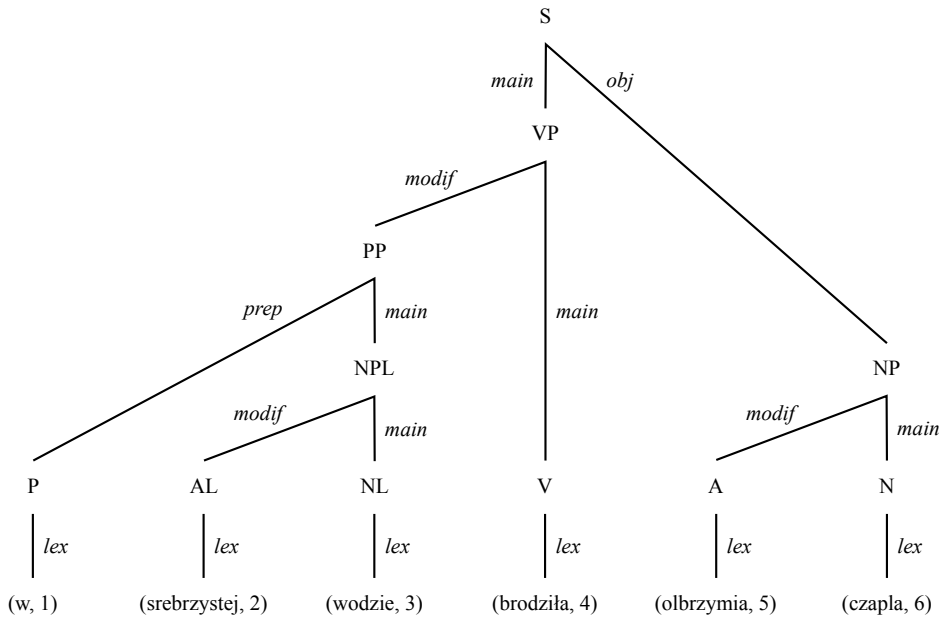
2. Obecność symbolu $*$ nad strzałką ($\xrightarrow{*}$) oznacza ograniczenie kolejności składników: wyrażenia po prawej stronie reguły muszą wystąpić w podanej kolejności, choć niekoniecznie muszą być ciągłe i niekoniecznie muszą występować bezpośrednio jedno po drugim.
   Dla reguły $A \xrightarrow{*} r_1 \ : \ B_1 \ldots r_n \ : \ B_n$ oznacza to, że ostatnie słowo wyrażenia kategorii B_i musi wystąpić przed pierwszym słowem wyrażenia kategorii B_{i+1} dla $i = 1, 2, \ldots, n-1$.
3. Symbol $\sim$ może pojawić się nad strzałką ($\xrightarrow{\sim}$) tylko wtedy, gdy po prawej stronie znajdują się dwa wyrażenia, i służy do wyrażenia *bezpośredniego poprzedzania*.
   Dla reguły $A \xrightarrow{\sim} r_1 : B_1 \, r_2 : B_2$ oznacza to, że pierwszy symbol terminalny wyrażenia kategorii B_2 musi nastąpić bezpośrednio po ostatnim symbolu terminalnym wyrażenia kategorii B_1.

Role składniowe służą do etykietowania krawędzi budowanych drzew składniowych. Role składniowe mogą oznaczać funkcje fraz w zdaniu, takie jak funkcja podmiotu, orzeczenia, dopełnienia czy modyfikatora.

Przykład 3.4 (FROG)**.** Przykład gramatyki FROG $G = (V, T, R, S, \mathcal{R})$:

$$
\begin{aligned}
V &= \{\, S\,,\ A\,,\ AL\,,\ N\,,\ NL\,,\ NP\,,\ NPL\,,\ P\,,\ PP\,,\ V\,,\ VP \,\}\,,\\
T &= \{\, \textit{brodziła}\,,\ \textit{czapla}\,,\ \textit{olbrzymia}\,,\ \textit{srebrzystej}\,,\ \textit{w}\,,\ \textit{wodzie} \,\}\,,\\
R &= \{\, S \to \textit{subj} : NP \ \textit{main} : VP\,,\\
&\qquad NP \xrightarrow{*} \textit{modif} : A \ \textit{main} : N\,, \qquad (3.1)\\
&\qquad NPL \to \textit{modif} : AL \ \textit{main} : NL\,,\\
&\qquad PP \xrightarrow{\sim} \textit{prep} : P \ \textit{main} : NPL^*, \qquad (3.2)\\
&\qquad VP \to \textit{main} : V \ \textit{modif} : PP\,,\\
&\qquad V \to \textit{lex} : \textit{brodziła}\,,\\
&\qquad AL \to \textit{lex} : \textit{srebrzystej}\,,\\
&\qquad N \to \textit{lex} : \textit{czapla}\,,\\
&\qquad NL \to \textit{lex} : \textit{wodzie}\,,\\
&\qquad P \to \textit{lex} : \textit{w}\,,\\
&\qquad A \to \textit{lex} : \textit{olbrzymia} \,\}\,,\\
&S \in V \text{ jest symbolem początkowym,}\\
\mathcal{R} &= \{\, \textit{lex}\,,\ \textit{main}\,,\ \textit{modif}\,,\ \textit{prep}\,,\ \textit{subj} \,\}\,.
\end{aligned}
$$

Gramatyka G generuje wszystkie poprawne polskie zdania złożone z wyrazów ze zbioru T takie, że:



Rysunek 3.5. Przykład drzewa generowanego przez FROG

— wyraz *olbrzymia* poprzedza wyraz *czapla* (ale niekoniecznie bezpośrednio) — zapewnia to symbol $*$ nad strzałką w regule (3.1),
— przedimek *w* poprzedza bezpośrednio frazę rzeczownikową *srebrzystej wodzie* (której szyk może być dowolny) — zapewnia to symbol $\sim$ nad strzałką w regule (3.2),
— fraza *srebrzystej wodzie* jest ciągła — wymusza to gwiazdka przy symbolu *NPL* w regule (3.2).

Język $L(G)$ zawiera zatem takie zdania, jak np.:
— *w srebrzystej wodzie brodziła olbrzymia czapla* (rys. 3.5),
— *olbrzymia w wodzie srebrzystej brodziła czapla*,
— *olbrzymia czapla brodziła w srebrzystej wodzie*.

Definicja 3.5 (funkcja δ dla gramatyki FROG)**.** Dla gramatyki FROG $G = (V, T, R, S, \mathcal{R})$ definiujemy funkcję $\delta\colon R \times \mathbb{N} \cup \{\rightarrow\} \rightarrow \{0, *, \sim\}$ następująco:
— $\delta(A^* \xrightarrow{(*)(\sim)} r_1 : B_1^{(*)} \dots r_n : B_n^{(*)}, 0) = *$,
— $\delta(A \xrightarrow{(*)(\sim)} r_1 : B_1^{(*)} \dots r_n : B_n^{(*)}, 0) = 0$,
— $\delta(A^{(*)} \xrightarrow{(*)(\sim)} r_1 : B_1^{(*)} \dots B_i^* \dots r_n : B_n^{(*)}, i) = *$,
— $\delta(A^{(*)} \xrightarrow{(*)(\sim)} r_1 : B_1^{(*)} \dots B_i \dots r_n : B_n^{(*)}, i) = 0$,
— $\delta(A^{(*)} \xrightarrow{*} r_1 : B_1^{(*)} \dots r_n : B_n^{(*)}, 0) = *$,
— $\delta(A^{(*)} \xrightarrow{\sim} r_1 : B_1^{(*)} \dots r_n : B_n^{(*)}, 0) = \sim$,
— $\delta(A^{(*)} \rightarrow r_1 : B_1^{(*)} \dots r_n : B_n^{(*)}, 0) = 0$.

Innymi słowy, funkcja $\delta(r, n)$ mówi, czy i jaki symbol ($*$, $\sim$) znajduje się przy n-tej zmiennej w regule $r \in R$.

Przykład 3.5 (funkcja δ)**.** Niech $G = (V, T, R, S, \mathcal{R})$ będzie FROG. Niech

$$(A^* \overset{\sim}{\rightarrow} q : B^* \, r : C) \in R$$

będzie regułą gramatyki G. Oznaczmy tę regułę przez ρ.

Wówczas:

— $\delta(\rho, 0) = *$, ponieważ przy symbolu A występuje gwiazdka $*$,

— $\delta(\rho, 1) = *$, ponieważ przy symbolu B występuje gwiazdka $*$,

— $\delta(\rho, 2) = 0$, ponieważ przy symbolu C nie ma żadnych dodatkowych oznaczeń,

— $\delta(\rho, \rightarrow) = \sim$, ponieważ nad strzałką występuje znak $\sim$.

Aby zdefiniować relację wyprowadzalności drzew w gramatyce FROG, trzeba wpierw wprowadzić definicję interwału:

Definicja 3.6 (interwał)**.** Zbiór $I \subset \mathbb{N}$ nazywamy *interwałem*, jeżeli jest zbiorem pustym lub składa się z kolejnych liczb naturalnych, czyli wtedy, gdy $I = \{i, i + 1, \ldots, i + k\}$ dla pewnych $i, k \in \mathbb{N}$.

Definicja 3.7 (wyprowadzalność drzew we FROG)**.** *Relację wyprowadzalności* $\vdash_G \subseteq V \times \mathcal{T}(T, V, \mathcal{R})$ dla gramatyki FROG $G = (V, T, R, S, \mathcal{R})$ definiujemy jako najmniejszą relację $\vdash \subseteq V \times \mathcal{T}(T, V, \mathcal{R})$ spełniającą następujące warunki:

(1) Jeżeli $(A \rightarrow \epsilon) \in R$, to $A \vdash A[]$.

(2) Jeżeli $(A \rightarrow r : w) \in R$, to $A \rightarrow A[r : (w, k)]$ dla dowolnego $k \in \mathbb{N}_+$.

(3) Jeżeli $\rho = \left(A^{(*)} \overset{(*)(\sim)}{\longrightarrow} r_1 : B_1^{(*)} \ldots r_n : B_n^{(*)}\right) \in R$ i istnieją $t_1, \ldots, t_n \in \mathcal{T}(T, V, \mathcal{R})$ takie, że $B_1 \vdash t_1, \ldots B_n \vdash t_n$ oraz $\sigma(t_i) \cap \sigma(t_j) = \emptyset$ dla $i \neq j$, to $A \vdash A[r_1 : t_1, \ldots, r_n : t_n]$, o ile spełnione są następujące warunki dotyczące szyku i nieciągłości:
   (i) jeśli $\delta(\rho, 0) = *$, to $\sigma(t_1) \cup \ldots \cup \sigma(t_n)$ jest interwałem,
   (ii) jeśli $\delta(\rho, i) = *$, to $\sigma(t_i)$ jest interwałem,
   (iii) jeśli $\delta(\rho, \rightarrow) = *$, to $\max(\sigma(t_i)) < \min(\sigma(t_j))$ dla $i < j$, $\sigma(t_i) \neq \emptyset$, $\sigma(t_j) \neq \emptyset$,
   (iv) jeśli $\delta(\rho, \rightarrow) = \sim$ oraz $\sigma(t_1) \neq \emptyset$ i $\sigma(t_2) \neq \emptyset$, to $\max(\sigma(t_1)) + 1 = \min(\sigma(t_2))$.

Definicja 3.8 (język drzew generowany przez FROG)**.** *Język drzew* generowany przez gramatykę FROG $G = (V, T, R, S, \mathcal{R})$ definiujemy jako:

$$L_{\mathrm{T}}(G) := \{t \in \mathcal{T}(T, V, \mathcal{R}) \colon S \vdash_G t\} \, .$$

Definicja 3.9 (wyprowadzalność łańcuchów we FROG)**.** Mówimy, że łańcuch $\omega \in T^*$ jest *wyprowadzalny* z symbolu $A \in V$ w gramatyce FROG

$G = (V, T, R, S, \mathcal{R})$, jeżeli istnieje wyprowadzalne z tego symbolu w tej gramatyce drzewo, którego plonem jest ω. Zapisujemy ten fakt jako $A \vdash_G \omega$ Innymi słowy, $A \vdash_G \omega$ wtedy i tylko wtedy, gdy $A \vdash_G t$ i $\text{yield}(t) = \omega$ dla pewnego $t \in \mathcal{T}(T, V, \mathcal{R})$.

Definicja 3.10 (język napisów generowany przez FROG)**.** *Język (napisów)* generowany przez gramatykę FROG $G = (V, T, R, S, \mathcal{R})$ definiujemy jako:

$$L(G) := \{\omega \in T^* \colon S \vdash_G \omega\} .$$

Moc generatywna gramatyk FROG jest większa niż (uporządkowanych) gramatyk bezkontekstowych, tj. istnieją gramatyki FROG generujące języki, które nie są bezkontekstowe. Dowód tego faktu można znaleźć w [15].

3.4.2. Gramatyki binarne generujące drzewa (TgBG)[5]

Kolejny formalizm opisujący zjawiska szyku swobodnego i nieciągłości syntaktycznych — gramatyki binarne generujące drzewa — stworzono w odpowiedzi na następujące postulaty:

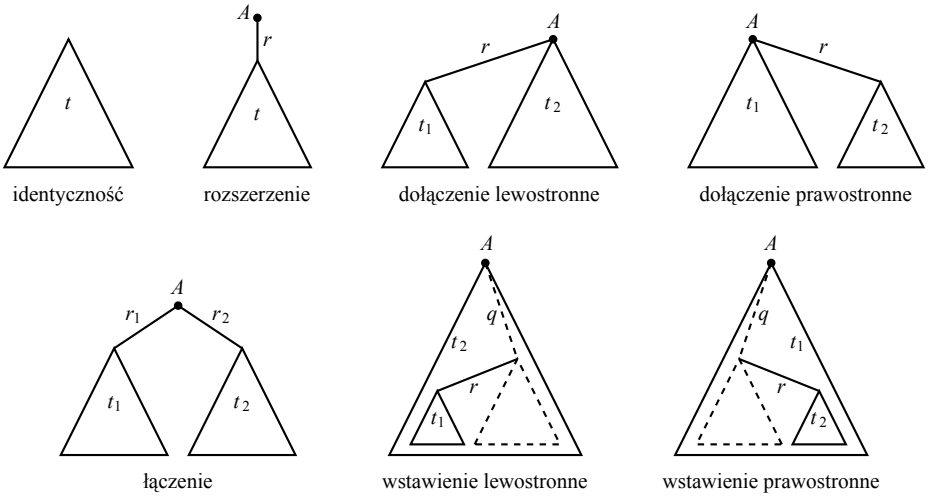
(1) jednorodność strategii parsingu dla języków różnych typów,
(2) reprezentacja drzew składniowych umożliwiająca dalsze przetwarzanie przez systemy tłumaczenia automatycznego korzystające z metod transferu,
(3) szybkie przetwarzanie,
(4) rozróżnienie między wyprowadzeniem (drzewem wyprowadzenia) a drzewem składniowym,
(5) płaska reprezentacja.

Gramatyki zależnościowe nie spełniają postulatu (2) [34]. Postulat (3) sugerują użycie rozszerzenia gramatyk bezkontekstowych, aby zachować co najwyżej sześcienną złożoność parsingu. Postulaty (4) i (5) są istotne ze względu na ułatwienie dalszego przetwarzania reprezentacji zdania (np. w celu zastosowania transferu do innego języka).

Powyższy zestaw postulatów spełniają unarne i binarne operacje na drzewach przedstawione w dalszej części niniejszego podrozdziału.

Gramatyki binarne generujące drzewa (ang. *tree-generating binary grammars*, TgBG) są formalizmem średniego poziomu — zajmują środkową pozycję na skali rozpiętej pomiędzy formalizmami wykorzystywanymi w płytkich parserach a bardziej skomplikowanymi gramatykami takimi jak HPSG [40] czy LFG [28].

[5]Na podstawie [14], [15]



Rysunek 3.6. Schematyczne przedstawienie wybranych operacji na drzewach

Operacje na drzewach

Operacja na drzewach to funkcja, która przyporządkowuje drzewu składniowemu lub parze drzew składniowych nowe drzewo składniowe. Czasami operacja na drzewach może nie dać się zastosować do danych argumentów; w takiej sytuacji wynik operacji będziemy oznaczać przez ∞.

Rozróżniamy unarne i binarne operacje na drzewach (w zależności od liczby argumentów). Rodzaje operacji na drzewach rozpatrywane w niniejszej pracy są zaczerpnięte z [15]. Operacje na drzewach rozważane w niniejszej pracy przedstawione są na rysunku 3.6.

Unarne operacje na drzewach są funkcjami postaci

$$I : \mathcal{T}(\Sigma, \mathcal{C}, \mathcal{R}) \rightarrow \mathcal{T}(\Sigma, \mathcal{C}, \mathcal{R}) \cup \{\infty\} \ . \tag{3.3}$$

Rozróżniamy dwa rodzaje unarnych operacji na drzewach: identyczność i rozszerzenie.

Operacja *identyczności* zdefiniowana jest następująco:

$$id(t) = t \ , \tag{3.4}$$

dla $t \in \mathcal{T}(\Sigma, \mathcal{C}, \mathcal{R})$. Wynikiem operacji jest niezmienione drzewo.

Operacja *rozszerzenia* zdefiniowana jest następująco:

$$ext(A, r)(t) = A[r : t] \ , \tag{3.5}$$

dla $A \in \mathcal{C}$, $r \in \mathcal{R}$, $t \in \mathcal{T}(\Sigma, \mathcal{C}, \mathcal{R})$. Operacja rozszerzenia dołącza drzewo t do węzła A za pomocą krawędzi etykietowanej rolą składniową r.

Binarne operacje na drzewach są funkcjami postaci

$$I : \mathcal{T}(\Sigma, \mathcal{C}, \mathcal{R})^2 \rightarrow \mathcal{T}(\Sigma, \mathcal{C}, \mathcal{R}) \cup \{\infty\} \ . \tag{3.6}$$

Zakładamy, że drzewa, które są argumentami operacji, mają rozłączne bazy. W przeciwnym wypadku (jeśli $\sigma(t_1) \cap \sigma(t_2) \neq \emptyset$) przyjmujemy $I(t_1, t_2) = \infty$. Rozróżniamy następujące binarne operacje na drzewach: łączenie, dołączenie lewostronne, dołączenie prawostronne, wstawienie lewostronne, wstawienie prawostronne.

Operacja *łączenia* zdefiniowana jest następująco:

$$cb(A, r_1, r_2)(t_1, t_2) = A[r_1 : t_1, r_2 : t_2] \,, \tag{3.7}$$

gdzie $A \in \mathcal{C}$, $r_1, r_2 \in \mathcal{R}$, $t_1, t_2 \in \mathcal{T}(\Sigma, \mathcal{C}, \mathcal{R})$. Operacja łączy dwa drzewa t_1 i t_2 dołączając je oba jako bezpośrednie poddrzewa do węzła A za pomocą krawędzi etykietowanych rolami składniowymi odpowiednio r_1 i r_2.

Operacje *dołączenia lewo-* i *prawostronnego* są zdefiniowane odpowiednio jako:

$$la(r)(t_1, A[q_1 : s_1, \ldots, q_n : s_n]) = A[r : t_1, q_1 : s_1, \ldots, q_n : s_n] \,, \tag{3.8}$$

$$ra(r)(A[q_1 : s_1, \ldots, q_n : s_n], t_2) = A[q_1 : s_1, \ldots, q_n : s_n, r : t_2] \,, \tag{3.9}$$

gdzie $A \in \mathcal{C}$, $r, q_1, \ldots, q_n \in \mathcal{R}$, $t_1, t_2, s_1, \ldots, s_n \in \mathcal{T}(\Sigma, \mathcal{C}, \mathcal{R})$.

Operacja dołączenia lewostronnego $la(r)(t_1, t_2)$ dołącza drzewo t_1 do korzenia drzewa t_2 jako jego bezpośrednie poddrzewo położone najbardziej po lewej stronie. Analogicznie, operacja dołączenia prawostronnego dołącza drzewo t_2 do korzenia drzewa t_1 jako jego bezpośrednie poddrzewo położone najbardziej po prawej stronie.

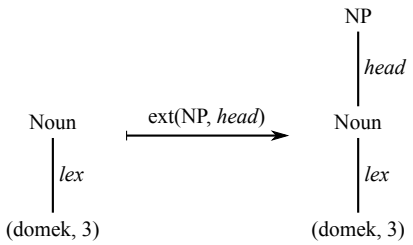
Operacje *wstawienia lewo-* i *prawostronnego* są zdefiniowane odpowiednio jako:

$$\begin{aligned} &li(q, r)(t_1, A[q_1 : s_1, \ldots, q : s_0, \ldots, q_n : s_n]) = \\ &\qquad = A[q_1 : s_1, \ldots, q : la(r)(t_1, s_0), \ldots, q_n : s_n] \,, \end{aligned} \tag{3.10}$$

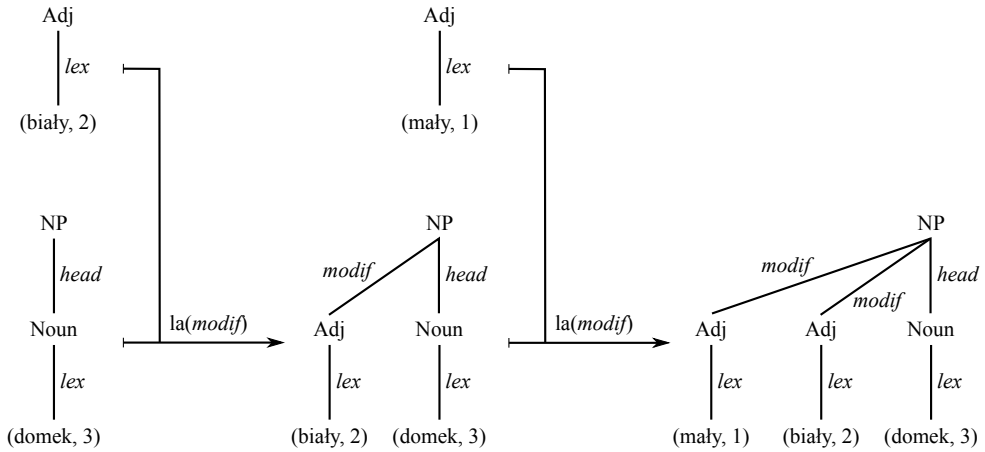
$$\begin{aligned} &ri(q, r)(A[q_1 : s_1, \ldots, q : s_0, \ldots, q_n : s_n], t_2) = \\ &\qquad = A[q_1 : s_1, \ldots, q : ra(r)(s_0, t_2), \ldots, q_n : s_n] \,, \end{aligned} \tag{3.11}$$

gdzie $A \in \mathcal{C}$, $r, q, q_1, \ldots, q_n \in \mathcal{R}$, $t_1, t_2, s_0, s_1, \ldots, s_n \in \mathcal{T}(\Sigma, \mathcal{C}, \mathcal{R})$. Operacja wstawienia lewostronnego $li(q, r)(t_1, t_2)$ dołącza lewostronnie drzewo t_1 do jedynego bezpośredniego poddrzewa drzewa t_2, którego rolą składniową jest q. Jeżeli drzewo t_2 nie posiada poddrzewa, którego rolą składniową jest q, lub jeśli posiada więcej niż jedno takie poddrzewo, to wynikiem operacji jest ∞. Powyższe stwierdzenia odnoszą się również do operacji wstawienia prawostronnego, która jest zdefiniowana w podobny sposób.

Różne operacje na drzewach umożliwiają przedstawienie różnych zjawisk gramatycznych. Operacja rozszerzenia może służyć na przykład do wyrażenia



Rysunek 3.7. Przykład operacji rozszerzenia



Rysunek 3.8. Przykład operacji dołączenia lewostronnego. Tutaj operacja została wykonana dwukrotnie

faktu, że fraza rzeczownikowa może składać się z pojedynczego rzeczownika (rys. 3.7).

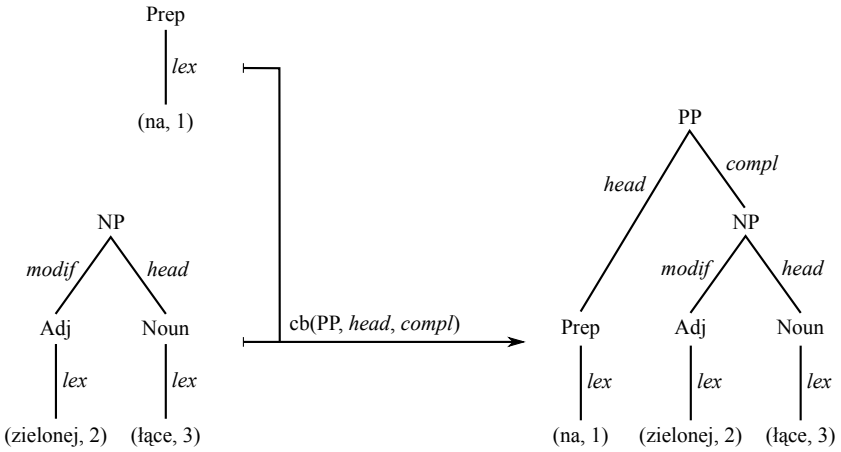
Operacje dołączenia (lewostronnego i prawostronnego) umożliwiają dołączanie szeregu modyfikatorów do danego symbolu (rys. 3.8). Efektem takiego zabiegu jest drzewo płaskie (postulat (5) ze strony 35).

Operacja łączenia może zostać użyta do wyrażenia kategorii gramatycznej, która składa się z dwóch części, np. fraza przyimkowa składa się z przyimka i frazy rzeczownikowej, zatem wynikiem operacji łączenia $cb(\,PP\,,\,head\,,\,compl\,)$ zastosowanej do dwóch drzew o korzeniach etykietowanych kategoriami *Prep* i *NP* jest drzewo frazy przyimkowej (rys. 3.9).

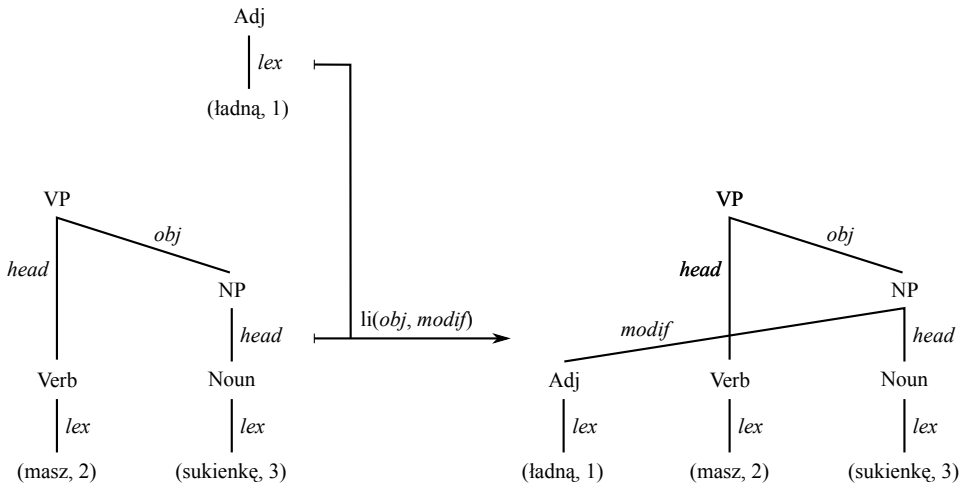
Operacja wstawienia pozwala na uwzględnianie nieciągłości syntaktycznych, np. przymiotnik opisujący rzeczownik można wstawić do frazy czasownikowej zawierającej ten rzeczownik używając operacji $li(\,obj\,,\,modif\,)$, nawet gdy przymiotnik nie sąsiaduje bezpośrednio z rzeczownikiem (rys. 3.10).

Definicje

Definicję gramatyki binarnej generującej drzewa podaję za [15].



Rysunek 3.9. Przykład operacji łączenia



Rysunek 3.10. Przykład operacji wstawienia lewostronnego

Definicja 3.11 (TgBG)**.** *Gramatyka binarna generująca drzewa* (TgBG) jest zdefiniowana jako szóstka $(T, Q, Q_s, \mathcal{C}, \mathcal{R}, R)$, gdzie:

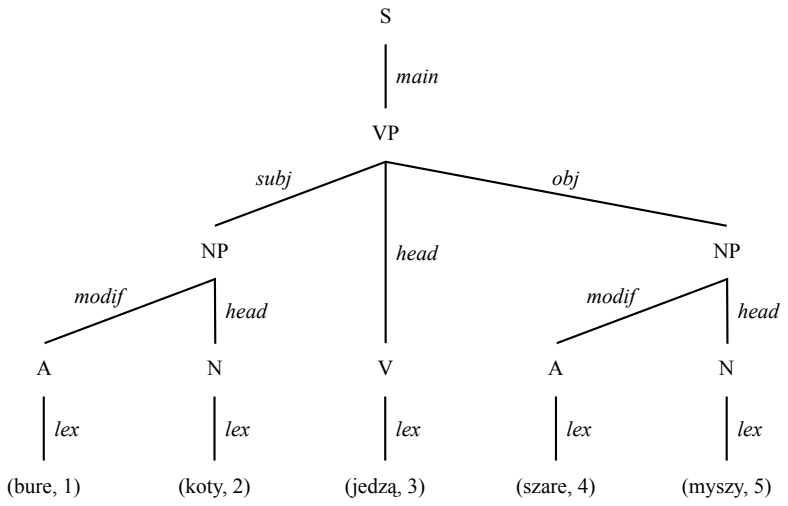
- T jest skończonym zbiorem *symboli terminalnych*,
- Q jest skończonym zbiorem *zmiennych*,
- $Q_s \subseteq Q$ jest zbiorem *zmiennych początkowych*,
- $\mathcal{C}$ jest skończonym zbiorem *kategorii*,
- $\mathcal{R}$ jest skończonym zbiorem *ról składniowych*,
- R jest skończonym zbiorem *reguł produkcji* postaci:
  - $x \to w : A : r$, $x \in Q$, $w \in T$, $A \in \mathcal{C}$, $r \in \mathcal{R}$ (*reguły leksykalne*), lub
  - $x \to y : I$, $x, y \in Q$, I jest unarną operacją na drzewach (*reguły unarne*), lub
  - $x \to y\, z : I$, $x, y, z \in Q$, I jest binarną operacją na drzewach (*reguły binarne*).

Koncepcja gramatyki binarnej generującej drzewa opiera się na pomyśle podobnym do gramatyk drzewiastych [47]: zmienne ze zbioru Q pełnią inną rolę niż kategorie ze zbioru $\mathcal{C}$. Kategorie są wykorzystywane do etykietowania węzłów generowanego drzewa składniowego, podczas gdy zmienne pełnią rolę pomocniczą w procesie budowy drzewa i nie są używane w docelowym drzewie.

Różnicę tę ilustruje następujący przykład 3.6:

Przykład 3.6 (TgBG)**.** Rozważmy następującą gramatykę binarną generującą drzewa:

$$
\begin{aligned}
G &= (T, Q, Q_s, \mathcal{C}, \mathcal{R}, R)\ , \\
T &= \{\, \mathit{szare}\,,\ \mathit{bure}\,,\ \mathit{czarne}\,,\ \mathit{psy}\,,\ \mathit{koty}\,,\ \mathit{myszy}\,,\ \mathit{jedzą}\,,\ \mathit{lubią}\,\}\ , \\
Q &= \{a, n, v, np, rvp, vp, s\}\ , \\
Q_s &= \{s\}\ , \\
\mathcal{C} &= \{A, N, V, NP, VP, S\}\ , \\
\mathcal{R} &= \{\, \mathit{lex}\,,\ \mathit{head}\,,\ \mathit{main}\,,\ \mathit{modif}\,,\ \mathit{subj}\,,\ \mathit{obj}\,\}\ , \\
R &= \{a \to \mathit{szare} : A : \mathit{lex}\ , \\
&\qquad a \to \mathit{bure} : A : \mathit{lex}\ , \\
&\qquad a \to \mathit{czarne} : A : \mathit{lex}\ , \\
&\qquad n \to \mathit{psy} : N : \mathit{lex}\ , \\
&\qquad n \to \mathit{koty} : N : \mathit{lex}\ , \\
&\qquad n \to \mathit{myszy} : N : \mathit{lex}\ , \\
&\qquad v \to \mathit{jedzą} : V : \mathit{lex}\ , \\
&\qquad v \to \mathit{lubią} : V : \mathit{lex}\ ,
\end{aligned}
$$



Rysunek 3.11. Przykład drzewa generowanego przez TgBG

$$
\begin{aligned}
&np \rightarrow n : xt(\, NP\,,\, head\,)\ ,\\
&np \rightarrow a\, np : la(\, modif\,)\ ,\\
&lvp \rightarrow v : xt(\, VP\,,\, head\,)\ ,\\
&lvp \rightarrow np\, v : cb(\, VP\,,\, obj\,,\, head\,)\ ,\\
&rvp \rightarrow v : xt(\, VP\,,\, head\,)\ ,\\
&rvp \rightarrow v\, np : cb(\, VP\,,\, head\,,\, obj\,)\ ,\\
&vp \rightarrow lvp\, np : ra(\, subj\,)\ ,\\
&vp \rightarrow np\, rvp : la(\, subj\,)\ ,\\
&s \rightarrow vp : xt(\, S\,,\, main\,)\}\ .
\end{aligned}
\tag{3.12}
$$

Przykładowe drzewo składniowe (rys. 3.11) generowane przez tę gramatykę:

$$
\begin{aligned}
t_1 = S[\, &main : VP[\, subj : NP[\, modif : A[\, lex : bure\,],\\
&head : N[\, lex : koty\,]],\ head : V[\, lex : jedzą\,],\\
&obj : NP[\, modif : A[\, lex : szare\,],\ head : N[\, lex : myszy\,]]]\ .
\end{aligned}
\tag{3.13}
$$

Plon drzewa t_1 należy do języka generowanego przez G:

$$
\mathrm{yield}(t_1) = bure\ koty\ jedzą\ szare\ myszy\ \in L(G)\ . \tag{3.14}
$$

Rozdział 4

Probabilistyczne gramatyki binarne generujące drzewa (PTgBG)

Probabilistyczne gramatyki binarne generujące drzewa (ang. *probabilistic tree-generating binary grammars*, PTgBG) są rozszerzeniem TgBG uzyskanym przez przyporządkowanie każdej regule gramatyki jej prawdopodobieństwa na podobnej zasadzie, jak probabilistyczne gramatyki bezkontekstowe (PCFG) są rozszerzeniem CFG. Motywacja dla stworzenia PTgBG jest analogiczna do motywacji dla stworzenia PCFG. Dla PCFG można zdefiniować pojęcia prawdopodobieństwa drzewa i prawdopodobieństwa zdania. To sprawia, że PCFG są narzędziem pomocnym w rozwiązywaniu wielu problemów dotyczących języków naturalnych:

— Algorytmy znajdowania najbardziej prawdopodobnego drzewa składniowego są przydatne do ujednoznaczniania znaczeń zdań [37].
— Obliczając prawdopodobieństwa zdań można uzyskać model języka [33].
— PCFG pozwalają modelować takie zjawiska językowe, jak brak płynności w mowie potocznej czy błędy gramatyczne [33].

Plaehn [37] opisuje probabilistyczną wersję DPSG (ang. *discontinuous phrase structure grammar*, 'gramatyka nieciągłych struktur frazowych'), formalizmu stworzonego przez Bunta [4, 5]. W swoim artykule nakreśla również szkic implementacji tablicowego algorytmu parsowania dla probabilistycznej wersji DCFG. Badania przedstawione w niniejszej pracy są analogiczne do badań Plaehna. PTgBG jest probabilistyczną wersją TgBG. Niniejsza praca przedstawia dowody słabej równoważności między PTgBG a PCFG i pokazuje, że dowolny algorytm parsowania PCFG [7, 30, 48, 65] może zostać użyty do parsowania PTgBG.

Podrozdział 4.1 zawiera definicje: probabilistycznej gramatyki binarnej generującej drzewa, a także pojęć wyprowadzalności, prawdopodobieństw zdań i drzew oraz języków generowanych przez PTgBG.

W podrozdziale 4.2 dowodzone są twierdzenia o zależnościach między PTgBG a PCFG. Pokazuje się, że PTgBG mają większą moc generatywną niż PCFG, jeżeli chodzi o języki drzew generowane przez oba formalizmy. Dowodzi się też, że jeżeli spełnione są odpowiednie warunki, to prawdopodobieństwa zdań liczone według PTgBG i według PCFG są równe.

4.1. Definicje

Definicja 4.1 (PTgBG)**.** Definiujemy *probabilistyczną gramatykę binarną generującą drzewa* (PTgBG) jako uporządkowaną ósemkę $(T, Q, Q_s, \mathcal{C}, \mathcal{R}, R, P, P_s)$, gdzie:

— $(T, Q, Q_s, \mathcal{C}, \mathcal{R}, R)$ jest TgBG,

— funkcja $P\colon R \to [0,1]$ (nazywana *prawdopodobieństwem reguły*) spełnia następujący warunek:

$$\sum_{\rho \in R\colon\, \rho = (x \to \ldots)} P(\rho) \;=\; 1 \tag{4.1}$$

dla każdej zmiennej $x \in Q$ (tj. dla dowolnego $x \in Q$ prawdopodobieństwa wszystkich reguł ze zmienną x po lewej stronie sumują się do 1).

— funkcja $P_s\colon Q_s \to [0,1]$ (nazywana *prawdopodobieństwem początkowym*) spełnia następujący warunek:

$$\sum_{x \in Q_s} P_s(x) = 1\ . \tag{4.2}$$

Definicja 4.2 (wyprowadzalność dla PTgBG)**.** Definiujemy relację *wyprowadzalności* $\vdash_G$ dla PTgBG jako najmniejszą relację $\vdash\, \in\, Q \times \mathcal{T}(T, \mathcal{C}, \mathcal{R})$ spełniającą następujące warunki:

— jeżeli $x \to w : A : r \in R$ to $x \vdash A[r : w]$,

— jeżeli $x \to y : I \in R$, $y \vdash t$, $t \in \mathcal{T}(T, \mathcal{C}, \mathcal{R})$ i $I(t) \neq \infty$ then $x \vdash I(t)$,

— jeżeli $x \to yz : I \in R$, $y \vdash t_1$, $z \vdash t_2$, $t_1, t_2 \in \mathcal{T}(T, \mathcal{C}, \mathcal{R})$ i $I(t_1, t_2) \neq \infty$, to $x \vdash I(t_1, t_2)$.

Dla dowolnej zmiennej $q \in Q$ będziemy pisać $q \vdash_G w$, jeżeli $w \in T^*$ i $\mathrm{yield}(t) = w$ dla pewnego $t \in \mathcal{T}(T, \mathcal{C}, \mathcal{R})$ z $q \vdash_G t$.

Definicja 4.3 (język drzew generowany przez PTgBG)**.** *Język drzew generowany przez PTgBG* $G = (T, Q, Q_s, \mathcal{C}, \mathcal{R}, R, P, P_s)$ definiujemy jako zbiór

$$L_{\mathrm{T}}(G) = \{t \in \mathcal{T}(T, \mathcal{C}, \mathcal{R})\colon \exists q \in Q_s\colon q \vdash_G t\}\ . \tag{4.3}$$

Definicja 4.4 (język generowany przez PTgBG)**.** *Język (napisów, czyli zdań) generowany przez PTgBG* $G = (T, Q, Q_s, \mathcal{C}, \mathcal{R}, R, P, P_s)$ definiujemy jako zbiór

$$L(G) = \{w \in T^*\colon \exists t \in L_{\mathrm{T}}(G)\colon w = \mathrm{yield}(t)\}\ . \tag{4.4}$$

Definicja 4.5 (prawdopodobieństwo drzewa dla PTgBG)**.** Niech będzie dana PTgBG $G = (T, Q, Q_s, \mathcal{C}, \mathcal{R}, R, P, P_s)$. Definiujemy funkcję $\mathbf{P}_G\colon \mathcal{T}(T, \mathcal{C}, \mathcal{R}) \times Q \to \mathbb{R}$ (nazywaną funkcją *prawdopodobieństwa drzewa*) w następujący sposób:

$$\mathbf{P}_G(t, x) = 0 \quad \text{if} \quad x \nvdash_G t\ , \tag{4.5}$$

$$\mathbf{P}_G(t,x) = P(x \to a : A : r) \quad \text{if} \quad t = A[r:a]\ , \tag{4.6}$$

$$\begin{aligned}\mathbf{P}_G(t,x) = &\sum_{t_1 \in \mathcal{T}(T,\mathcal{C},\mathcal{R}):\ t=I(t_1)} P(x \to y : I) \cdot \mathbf{P}_G(t_1,y) + \\ &+ \sum_{t_1,t_2 \in \mathcal{T}(T,\mathcal{C},\mathcal{R}):\ t=I(t_1,t_2)} P(x \to y\, z : I) \cdot \mathbf{P}_G(t_1,y) \cdot \mathbf{P}_G(t_2,z) \\ &\text{w przeciwnym wypadku.}\end{aligned} \tag{4.7}$$

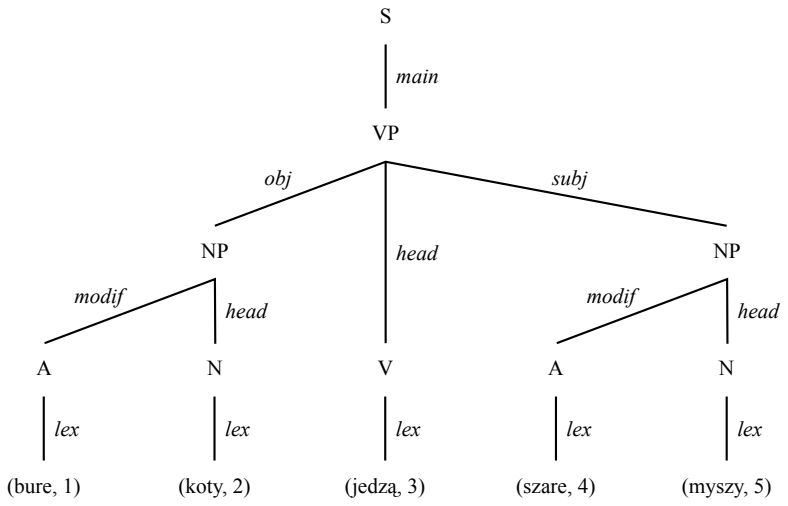
Jeżeli będzie jasno wynikać z kontekstu, o której gramatyce mowa, i nie będzie prowadzić to do nieporozumień, to będziemy pomijać symbol gramatyki i pisać po prostu $\mathbf{P}(t,x)$ dla $t \in \mathcal{T}(T,\mathcal{C},\mathcal{R})$, $x \in Q$.

Przykład 4.1 przedstawia probabilistyczną gramatykę binarną generującą drzewa, przykładowe drzewo składniowe generowane przez tę gramatykę oraz obliczenie jego prawdopodobieństwa.

Przykład 4.1 (PTgBG)**.** Niech $G = (T, Q, Q_s, \mathcal{C}, \mathcal{R}, R)$ będzie gramatyką z przykładu 3.6. Konstruujemy probabilistyczną gramatykę binarną generującą drzewa $G' = (T, Q, Q_s, \mathcal{C}, \mathcal{R}, R, P, P_s)$ na podstawie G definiując funkcje P i P_s w następujący sposób:

$$\begin{aligned}
&P(a \to \mathit{szare} : A : \mathit{lex}) = 0.2\ ,\\
&P(a \to \mathit{bure} : A : \mathit{lex}) = 0.3\ ,\\
&P(a \to \mathit{czarne} : A : \mathit{lex}) = 0.5\ ,\\
&P(n \to \mathit{psy} : N : \mathit{lex}) = 0.4\ ,\\
&P(n \to \mathit{koty} : N : \mathit{lex}) = 0.4\ ,\\
&P(n \to \mathit{myszy} : N : \mathit{lex}) = 0.2\ ,\\
&P(v \to \mathit{jedzą} : V : \mathit{lex}) = 0.7\ ,\\
&P(v \to \mathit{lubią} : V : \mathit{lex}) = 0.3\ ,\\
&P(np \to n : xt(\mathit{NP}, \mathit{head})) = 0.8\ ,\\
&P(np \to a\, np : la(\mathit{modif})) = 0.2\ ,\\
&P(lvp \to v : xt(\mathit{VP}, \mathit{head})) = 0.6\ ,\\
&P(lvp \to np\, v : cb(\mathit{VP}, \mathit{obj}, \mathit{head})) = 0.4\ ,\\
&P(rvp \to v : xt(\mathit{VP}, \mathit{head})) = 0.6\ ,\\
&P(rvp \to v\, np : cb(\mathit{VP}, \mathit{head}, \mathit{obj})) = 0.4\ ,\\
&P(vp \to lvp\, np : ra(\mathit{subj})) = 0.1\ ,\\
&P(vp \to np\, rvp : la(\mathit{subj})) = 0.9\ ,\\
&P(s \to vp : xt(S, \mathit{main})) = 1\ .
\end{aligned} \tag{4.8}$$

Niech t_1 będzie drzewem składniowym z przykładu 3.6. Ponieważ istnieje tylko jedno możliwe wyprowadzenie tego drzewa, możemy obliczyć prawdopo-



Rysunek 4.1. Przykład drzewa generowanego przez TgBG

dobieństwo drzewa po prostu mnożąc prawdopodobieństwa wszystkich reguł występujących w drzewie:

$$\begin{aligned}\mathbf{P}(t_1, s) &= 0.3 \cdot 0.4 \cdot 0.7 \cdot 0.2 \cdot 0.2 \cdot 0.8 \cdot 0.2 \cdot 0.8 \cdot 0.2 \cdot 0.4 \cdot 0.9 \cdot 1 = \\ &= 0.000030966 \ . \end{aligned} \tag{4.9}$$

Definicja 4.6 (drzewo Viterbiego)**.** Niech będzie dana probabilistyczna gramatyka binarna generująca drzewa $G = (T, Q, Q_s, \mathcal{C}, \mathcal{R}, R, P, P_s)$ oraz łańcuch $w \in L(G)$. *Drzewem Viterbiego* dla łańcucha w nazywamy to spośród drzew składniowych $t \in L_{\mathrm{T}}(G)$, $\mathrm{yield}(t) = w$, dla którego wyrażenie

$$\sum_{q \in Q} \mathbf{P}_G(t, q) \cdot P_s(q)$$

osiąga największą wartość.

Definicja 4.7 (prawdopodobieństwo łańcucha dla PTgBG)**.** Niech będzie dana PTgBG $G = (T, Q, Q_s, \mathcal{C}, \mathcal{R}, R, P, P_s)$. Definiujemy *prawdopodobieństwo łańcucha* jako funkcję:

$$\begin{aligned}&\mathbf{P}_G : \ L(G) \to \mathbb{R} \ , \\ &\qquad \mathbf{P}_G(s) = \sum_{t \in \mathcal{T}(T,\mathcal{C},\mathcal{R}) \colon \mathrm{yield}(t) = s;\, q \in Q_s} \mathbf{P}_G(t, q) \cdot P_s(q) \ . \end{aligned} \tag{4.10}$$

Przykład 4.2 (drzewo Viterbiego, prawdopodobieństwo łańcucha)**.** Niech G będzie gramatyką z przykładu 4.1. Zdanie *bure koty jedzą szare myszy* ma dwa możliwe drzewa składniowe: drzewo t_1 pokazane na rysunku 3.11 oraz drzewo t_2 pokazane na rysunku 4.1.

Z (4.9) wiemy, że

$$\mathbf{P}_G(t_1, s) = 0.000030966 \ .$$

Mamy również:

$$\begin{aligned}\mathbf{P}(t_2, s) &= 0.3 \cdot 0.4 \cdot 0.7 \cdot 0.2 \cdot 0.2 \cdot 0.8 \cdot 0.2 \cdot 0.8 \cdot 0.2 \cdot 0.4 \cdot 0.1 \cdot 1 = \\ &= 0.000003441 \ . \end{aligned} \tag{4.11}$$

Widzimy, że

$$\mathbf{P}_{G'}(t_1, s) => = \mathbf{P}_{G'}(t_2, s) \ ,$$

co oznacza, że t_1 jest drzewem Viterbiego dla łańcucha *bure koty jedzą szare myszy* .

Możemy również obliczyć prawdopodobieństwo łańcucha *bure koty jedzą szare myszy* ze wzoru (4.10):

$$\begin{aligned}&\mathbf{P}_{G'}(\textit{ bure koty jedzą szare myszy }) = \\ &\quad = \mathbf{P}_{G'}(t_1, s) \cdot P_s(s) + \mathbf{P}_{G'}(t_2, s) \cdot P_s(s) = \\ &\quad = 0.000030966 \cdot 1 + 0.000003441 \cdot 1 = \\ &\quad = 0.000034407 \ . \end{aligned} \tag{4.12}$$

4.2. Zależności między PTgBG a PCFG

W tym rozdziale porównamy PTgBG i PCFG ze względu na języki i drzewa, jakie generują.

Tabela 4.1 przedstawia zależności między PTgBG i PCFG. W kolumnie po lewej podano, czy na podstawie danej PCFG można skonstruować PTgBG, która generuje identyczne języki bądź daje takie same prawdopodobieństwa. Analogicznie, w kolumnie po prawej podano, czy na podstawie danej PTgBG można skonstruować odpowiednią PCFG.

Tabela 4.1. Zależności między PTgBG i PCFG

	PCFG → PTgBG	PTgBG → PCFG
język drzew	TAK (twierdzenie 4.1)	NIE (kontrprzykład 4.3)
prawdop. drzew	TAK (twierdzenie 4.1)	NIE (kontrprzykład 4.3)
język (napisów/zdań)	TAK (twierdzenie 4.3)	TAK (twierdzenie 4.4)
prawdop. zdań	TAK (twierdzenie 4.3)	TAK (twierdzenie 4.5)

4.2.1. Porównanie języków drzew generowanych przez PTgBG i PCFG

Twierdzenie 4.1 pokazuje, że dla dowolnej PCFG można skonstruować PTgBG, która generuje ten sam język drzew. Na dodatek, wyliczone prawdopodobieństwa drzew są w obu przypadkach takie same.

Twierdzenie 4.1. *Dla każdej PCFG $G = (V, T, R, S, P)$:*

1. *można skonstruować PTgBG G' taką, że $L_{\mathrm{T}}(G) = L_{\mathrm{T}}(G')$,*
2. *dla dowolnego drzewa $t \in L_{\mathrm{T}}(G)$: $\mathbf{P}_G(t) = \mathbf{P}_{G'}(t, \mathrm{root}(t))$.*[1]

Dowód.

Część 1. Pierwsza część dowodu oparta jest na dowodzie twierdzenia 6 z [15]. Niech $G' = (T, V', \{S\}, V, \{u\}, R', P', P_s')$. Tabela 4.2 pokazuje sposób konstrukcji G'. V' zawiera wszystkie zmienne z V oraz nowe zmienne utworzone dla każdej reguły postaci $A \to B_1 \dots B_n$, gdzie $A, B_1, \dots, B_n \in V$ (dla każdej reguły z R tworzony jest oddzielny zbiór nowych zmiennych). Zbiór reguł R' gramatyki PTgBG składa się z reguł utworzonych jak w tabeli 4.2. Tabela pokazuje również prawdopodobieństwa reguł przypisywane każdej regule (określają funkcję prawdopodobieństwa reguł P').

Tabela 4.2. Sposób konstrukcji PTgBG równoważnej danej PCFG

Reguła PCFG	Nowe zmienne PTgBG	Nowe reguły PTgBG $p \in R'$	$P'(p)$
$A \to w$	—	$A \to w : A : u$	$P(A \to w)$
$A \to B$	—	$A \to B : ext(A, u)$	$P(A \to B)$
$A \to B_1 B_2$	—	$A \to B_1 B_2 : cb(A, u, u)$	$P(A \to B_1 B_2)$
$A \to B_1 \dots B_n$	$x_1, \dots, x_n$	$x_1 \to B_1 B_2 : cb(A, u, u)$	1
		$x_2 \to x_1 B_3 : ra(u)$	1
		$\vdots$	$\vdots$
		$x_{n-2} \to x_{n-3} B_{n-1} : ra(u)$	1
		$A \to x_{n-2} B_n : ra(u)$	$P(A \to B_1 \dots B_n)$

Funkcja P' określona w ten sposób spełnia warunek (4.1):

— Dla każdego $A \in V$:

$$\begin{aligned}
&\sum_{w \in T,\, r \in \mathcal{R}} P'(A \to w : A : r) + \sum_{y \in V'} P'(A \to y : I) + \sum_{y,z \in V'} P'(A \to y\, z : I) = \\
&\quad = \sum_{w \in T} P'(A \to w : A : u) + \sum_{B \in V} P'(A \to B : ext(A, u)) + \\
&\qquad + \sum_{B_1, B_2 \in V} P'(A \to B_1\, B_2 : cb(A, u, u)) + \\
&\qquad + \sum_{x \in V' \setminus V,\, B \in V} P'(A \to x\, B : ra(u)) = \\
&\quad = \sum_{w \in T} P(A \to w) + \sum_{B \in V} P(A \to B) + \\
&\qquad + \sum_{B_1, B_2 \in V} P(A \to B_1\, B_2) + \sum_{B_1, \dots, B_{n-1}, B \in V} P(A \to B_1\, \dots\, B_{n-1}\, B) = \\
&\quad = 1\,. \qquad (4.13)
\end{aligned}$$

[1] G' jest konstruowana w ten sposób, że jej alfabet zmiennych zawiera jej zbiór kategorii.

— Dla każdego $x \in V' \setminus V$:

$$\sum_{w \in T,\, r \in \mathcal{R}} P'(x \to w : x : r) + \sum_{y \in V'} P'(x \to y : I) + \sum_{y,z \in V'} P'(x \to y\, z : I) =$$
$$= \sum_{B_1, B_2 \in V} P'(x \to B_1\, B_2 : cb(A, u, u)) + \sum_{y \in V' \setminus V,\, B \in V} P'(x \to y\, B : ra(u)) =$$
$$= 1 \,. \tag{4.14}$$

Zdefiniujmy teraz funkcję $P_s{}'$ na $\{S\}$ jako

$$P_s{}'(S) = 1 \,. \tag{4.15}$$

Oczywiste jest, że funkcja $P_s{}'$ spełnia warunek wyrażony w (4.2).

Część 2. Musimy teraz pokazać, że $\mathbf{P}_{G'}(t, \mathrm{root}(t)) = \mathbf{P}_G(t)$. Aby dowieść tego stwierdzenia, użyjemy indukcji strukturalnej.

Zauważmy, że jedyne operacje na drzewach użyte w regułach gramatyki G' to ext, cb i ra. Ogranicza to liczbę reguł i operacji na drzewach, które mogą zostać użyte przy budowaniu danego drzewa składniowego. Z tego powodu możemy uprościć sumy pojawiające się w (4.7) przy obliczaniu prawdopodobieństw drzew. Rozważmy różne przypadki w zależności od postaci, jaką może przybrać drzewo t.

Pierwszy przypadek: drzewo t jest postaci $t = A[u : w]$ dla pewnych $A \in V$, $w \in T$. Korzystamy z (4.6):

$$\begin{aligned} &\mathbf{P}_{G'}(A[u : w], \mathrm{root}(A[u : w])) = \\ &\quad = \mathbf{P}_{G'}(A[u : w], A) = \\ &\quad = \mathbf{P}_{G'}(A \to w : A : u) = \\ &\quad = \mathbf{P}_G(A \to w) = \\ &\quad = \mathbf{P}_G(A[u : w]) \,. \end{aligned} \tag{4.16}$$

Drugi przypadek: drzewo t jest postaci $t = A[u : t_1]$ dla pewnego $A \in V$ i pewnego drzewa $t_1 \in \mathcal{T}(T, V, \{u\})$. Jedyną operacją na drzewach, jaka może być użyta przy budowie drzewa t, jest xt, natomiast jedyną regułą, którą można zastosować jest $A \to \mathrm{root}(t_1) : ext(A, u)$:

$$\begin{aligned} &\mathbf{P}_{G'}(A[u : t_1], \mathrm{root}(A[u : t_1])) = \\ &\quad = \mathbf{P}_{G'}(ext(A, u)(t_1), A) = \\ &\quad = P'(A \to \mathrm{root}(t_1) : ext(A, u)) \cdot \mathbf{P}_{G'}(t_1, \mathrm{root}(t_1)) = \\ &\quad = P(A \to \mathrm{root}(t_1)) \cdot \mathbf{P}_G(t_1) = \\ &\quad = \mathbf{P}_G(A[u : t_1]) \,. \end{aligned} \tag{4.17}$$

Trzeci przypadek: drzewo t jest postaci $t = A[u : t_1, u : t_2]$ dla pewnego $A \in V$ i pewnych drzew $t_1, t_2 \in \mathcal{T}(T, V, \{u\})$. Istnieją co najwyżej dwie reguły, które mogą zostać użyte do zbudowania drzewa t. Jedną z nich jest $A \to \mathrm{root}(t_1)\,\mathrm{root}(t_2) : cb(A, u, u)$. Możemy jej użyć do obliczenia prawdopodobieństwa drzewa. Reguła $x \to \mathrm{root}(t_1)\,\mathrm{root}(t_2) : cb(A, u, u)$ (jeżeli taka reguła jest w zbiorze R') mogłaby również być wzięta pod uwagę, ale nie można jej użyć do obliczenia prawdopodobieństwa drzewa, ponieważ $\mathrm{root}(t) = A \neq x$. Stąd otrzymujemy:

$$\begin{aligned}
&\mathbf{P}_{G'}(A[u : t_1, u : t_2], \mathrm{root}(A[u : t_1, u : t_2])) = \\
&\quad = \mathbf{P}_{G'}(cb(A, u, u)(t_1, t_2), A) = \\
&\quad = P'(A \to \mathrm{root}(t_1)\,\mathrm{root}(t_2) : cb(A, u, u)) \cdot \\
&\qquad \cdot \mathbf{P}_{G'}(t_1, \mathrm{root}(t_1)) \cdot \mathbf{P}_{G'}(t_2, \mathrm{root}(t_2)) = \\
&\quad = P(A \to t_1 t_2) \cdot \mathbf{P}_G(t_1) \cdot \mathbf{P}_G(t_2) = \\
&\quad = \mathbf{P}_G(A[u : t_1, u : t_2])\ . \qquad (4.18)
\end{aligned}$$

Ostatni przypadek, gdy $\mathrm{root}(t)$ ma więcej niż dwóch synów, jest najbardziej skomplikowany. Użyjemy tu lematu 4.2, którego dowód jest przedstawiony w dalszej części pracy. Zauważmy również, że jedyną operacją na drzewach, jaka może zostać użyta do budowy drzewa t jest ra, a jedyną regułą, którą można zastosować, jest $A \to x_{n-2}\,\mathrm{root}(t_n) : ra(u)$. Obliczenie prawdopodobieństwa drzewa wymaga wyliczenia prawdopodobieństw drzew dla szeregu drzew, z których każde może być jednoznacznie przedstawione jako wynik operacji na drzewach. W rezultacie otrzymujemy:

$$\begin{aligned}
&\mathbf{P}_{G'}(A[u : t_1, \ldots, u : t_n], \mathrm{root}(A[u : t_1, \ldots, u : t_n])) = \\
&\quad = \mathbf{P}_{G'}(ra(u)(A[u : t_1, \ldots, u : t_{n-1}], t_n), A) = \\
&\quad = P'(A \to x_{n-2}\,\mathrm{root}(t_n) : ra(u)) \cdot \\
&\qquad \cdot \mathbf{P}_{G'}(A[u : t_1, \ldots, u : t_{n-1}], x_{n-2}) \cdot \mathbf{P}_{G'}(t_n, \mathrm{root}(t_n)) = \\
&\quad = P'(A \to x_{n-2}\,\mathrm{root}(t_n) : ra(u)) \cdot \prod_{i=1}^{n-3} P'(x_{i+1} \to x_i\,\mathrm{root}(t_{i+2}) : ra(u)) \cdot \\
&\qquad \cdot \mathbf{P}_{G'}(A[u : t_1, u : t_2], x_1) \cdot \prod_{i=1}^{n-3} \mathbf{P}_{G'}(t_{i+2}, \mathrm{root}(t_{i+2})) \cdot \mathbf{P}_{G'}(t_n, \mathrm{root}(t_n)) = \\
&\quad = P'(A \to x_{n-2}\,\mathrm{root}(t_n) : ra(u)) \cdot \prod_{i=2}^{n-2} P'(x_i \to x_{i-1}\,\mathrm{root}(t_{i+1}) : ra(u)) \cdot \\
&\qquad \cdot P'(x_1 \to \mathrm{root}(t_1)\,\mathrm{root}(t_2) : cb(A, u, u)) \cdot \mathbf{P}_{G'}(t_1, \mathrm{root}(t_1)) \cdot \\
&\qquad \cdot \mathbf{P}_{G'}(t_2, \mathrm{root}(t_2)) \cdot \prod_{i=3}^{n-1} \mathbf{P}_{G'}(t_i, \mathrm{root}(t_i)) \cdot \mathbf{P}_{G'}(t_n, \mathrm{root}(t_n)) =
\end{aligned}$$

$$
\begin{aligned}
&= P'(A \to x_{n-2}\,\mathrm{root}(t_n) : ra(u)) \cdot \\
&\quad \cdot\ P'(x_{n-2} \to x_{n-3}\,\mathrm{root}(t_{n-1}) : ra(u)) \cdot \ldots \cdot \\
&\quad \cdot \ldots \cdot P'(x_2 \to x_1\,\mathrm{root}(t_3) : ra(u)) \cdot \\
&\quad \cdot\ P'(x_1 \to \mathrm{root}(t_1)\,\mathrm{root}(t_2) : cb(A,u,u)) \cdot \\
&\quad \cdot\ \mathbf{P}_{G'}(t_1, \mathrm{root}(t_1)) \cdot \ldots \cdot \mathbf{P}_{G'}(t_n, \mathrm{root}(t_n)) = \\
&= P(A \to \mathrm{root}(t_1)\ \ldots\ \mathrm{root}(t_n)) \cdot \mathbf{P}_G(t_1) \cdot \ldots \cdot \mathbf{P}_G(t_n) = \\
&= \mathbf{P}_G(A[u:t_1, \ldots, u:t_n])\ . \qquad (4.19)
\end{aligned}
$$

Rozważywszy wszystkie możliwe przypadki, pokazaliśmy, że dla dowolnego drzewa $t \in L_{\mathrm{T}}(G)$:

$$\mathbf{P}_{G'}(t, \mathrm{root}(t)) = \mathbf{P}_G(t) \qquad (4.20)$$

□

Lemat 4.2. *Niech $G' = (T, V', \{S\}, V, \{u\}, R', P', P'_s)$ będzie probabilistyczną gramatyką binarną generującą drzewa (PTgBG) zdefiniowaną tak, jak w twierdzeniu 4.1. Niech*

$$
\begin{aligned}
&t_1, \ldots, t_{n+2} \in \mathcal{T}(T, V, \{u\})\ , \\
&A, \mathrm{root}(t_1), \ldots, \mathrm{root}(t_{n+2}) \in V\ , \\
&x_1, \ldots, x_{n+1} \in V'\ . \qquad (4.21)
\end{aligned}
$$

Wówczas

$$
\begin{aligned}
&\mathbf{P}(A[u:t_1, \ldots, u:t_{n+2}], x_{n+1}) = \\
&\quad = \prod_{i=1}^{n} P(x_{i+1} \to x_i\,\mathrm{root}(t_{i+2}) : ra(u)) \cdot \\
&\qquad \cdot\ \mathbf{P}(A[u:t_1, u:t_2], x_1) \cdot \prod_{i=1}^{n} \mathbf{P}(t_{i+2}, \mathrm{root}(t_{i+2})) \qquad (4.22)
\end{aligned}
$$

dla dowolnego $n \in \mathbb{N}_+$.

Dowód. Ponieważ jedynymi operacjami na drzewach występującymi w regułach gramatyki G' są xt, ct i ra, więc każde drzewo t takie, że $\mathrm{root}(t)$ ma więcej niż dwóch synów, może powstać tylko jako wynik operacji ra. Sposób konstrukcji gramatyki G' zapewnia również, że istnieje zawsze tylko jedna reguła produkcji, która może zostać użyta do obliczenia prawdopodobieństwa drzewa dla każdego takiego drzewa t.

Tezy lematu dowiedziemy za pomocą zasady indukcji matematycznej.

Dla $n = 1$ mamy (ze wzoru (4.7)):

$$
\begin{aligned}
&\mathbf{P}(A[u:t_1, u:t_2, u:t_3], x_2) = \\
&\quad = \mathbf{P}(ra(u)(A[u:t_1, u:t_2], t_3), x_2) =
\end{aligned}
$$

$$= P(x_2 \to x_1\,\mathrm{root}(t_3) : ra(u))\cdot$$
$$\cdot\,\mathbf{P}(A[u:t_1, u:t_2], x_1)\cdot\mathbf{P}(t_3, \mathrm{root}(t_3))\ . \tag{4.23}$$

Przypuśćmy teraz, że

$$\mathbf{P}(A[u:t_1,\ldots,u:t_{n+2}], x_{n+1}) =$$
$$= \prod_{i=1}^{n} P(x_{i+1} \to x_i\,\mathrm{root}(t_{i+2}) : ra(u))\cdot$$
$$\cdot\,\mathbf{P}(A[u:t_1, u:t_2], x_1)\cdot\prod_{i=1}^{n}\mathbf{P}(t_{i+2}, \mathrm{root}(t_{i+2}))\ . \tag{4.24}$$

Wówczas

$$\begin{aligned}
&\mathbf{P}(A[u:t_1,\ldots,u:t_{n+2}, u:t_{n+3}], x_{n+2}) =\\
&\quad= \mathbf{P}(ra(u)(A[u:t_1,\ldots,u:t_{n+2}], t_{n+3}), x_{n+2}) =\\
&\quad= P(x_{n+2} \to x_{n+1}\,\mathrm{root}(t_{n+3}) : ra(u))\cdot\\
&\qquad\cdot\,\mathbf{P}(A[u:t_1,\ldots,u:t_{n+2}], x_{n+1})\cdot\mathbf{P}(t_{n+3}, \mathrm{root}(t_{n+3})) =\\
&\quad= P(x_{n+2} \to x_{n+1}\,\mathrm{root}(t_{n+3}) : ra(u))\cdot\prod_{i=1}^{n} P(x_{i+1} \to x_i\,\mathrm{root}(t_{i+2}) : ra(u))\cdot\\
&\qquad\cdot\,\mathbf{P}(A[u:t_1, u:t_2], x_1)\cdot\prod_{i=1}^{n}\mathbf{P}(t_{i+2}, \mathrm{root}(t_{i+2}))\cdot\mathbf{P}(t_{n+3}, \mathrm{root}(t_{n+3})) =\\
&\quad= \prod_{i=1}^{n+1} P(x_{i+1} \to x_i\,\mathrm{root}(t_{i+2}) : ra(u))\cdot\\
&\qquad\cdot\,\mathbf{P}(A[u:t_1, u:t_2], x_1)\cdot\prod_{i=1}^{n+1}\mathbf{P}(t_{i+2}, \mathrm{root}(t_{i+2}))\ .
\end{aligned} \tag{4.25}$$

Na mocy zasady indukcji matematycznej otrzymujemy tezę lematu. □

Zależność w drugą stronę (tj. że dla dowolnej PTgBG można skonstruować PCFG, która generuje ten sam język drzew) nie zachodzi. Pokazuje to następujący kontrprzykład oparty na przykładzie dla gramatyk nieprobabilistycznych opisanym w [15].

Przykład 4.3. Niech $G = (\{a\}, \{x, y\}, \{x\}, \{A\}, \{u\}, R, P, P_s)$ będzie probabilistyczną gramatyką binarną generującą drzewa, gdzie $R = \{x \to a : A : u, y \to a : A : u, x \to x\,y : ra(u)\}$. Prawdopodobieństwa reguł i prawdopodobieństwa początkowe mogą być dowolnie określone. W gramatyce G liczba bezpośrednich poddrzew generowanego drzewa jest nieograniczona (reguła $x \to x\,y : ra(u)$ może zostać zastosowana nieskończoną liczbę razy). W każdej PCFG liczba bezpośrednich poddrzew generowanego drzewa jest ograniczona przez liczbę symboli po prawej stronie reguły. Nie może zatem istnieć żadna PCFG, która generowałaby wszystkie drzewa, które mogą być wygenerowane przez G.

4.2.2. Porównanie języków napisów generowanych przez PTgBG i PCFG

Twierdzenie 4.3 pokazuje, że dla dowolnej PCFG można skonstruować PTgBG, która generuje wszystkie napisy generowane przez PCFG. Prawdopodobieństwa napisów liczone za pomocą skonstruowanej gramatyki PTgBG są takie same jak liczone za pomocą PCFG.

Twierdzenie 4.3. *Dla dowolnej PCFG $G = (V, T, R, S, P)$:*

1. *można skonstruować PTgBG G' taką, że $L(G) = L(G')$,*
2. *dla dowolnego łańcucha (zdania) $w \in L(G)$: $\mathbf{P}_G(w) = \mathbf{P}_{G'}(w)$.*

Twierdzenie 4.4 pokazuje, że dla każdej PTgBG można skonstruować PCFG, która generuje wszystkie napisy generowane przez PTgBG.

Twierdzenie 4.4. *Dla dowolnej PTgBG $G = (T, Q, Q_s, \mathcal{C}, \mathcal{R}, R, P, P_s)$ można skonstruować PCFG G' taką, że $L(G) = L(G')$.*

Z twierdzeń 4.3 i 4.4 płynie wniosek, że PCFG i PTgBG są równoważne pod względem napisów, jakie generują.

Twierdzenie 4.5 pokazuje, że dla każdej PTgBG spełniającej pewne warunki można skonstruować PCFG, która generuje wszystkie łańcuchy generowane przez tę PTgBG, a na dodatek wyliczone prawdopodobieństwa łańcuchów są w obu przypadkach takie same.

Twierdzenie 4.5. *Dla każdej PTgBG $G = (T, Q, Q_s, \mathcal{C}, \mathcal{R}, R, P, P_s)$, która nie używa operacji li ani ri:*

1. *możemy skonstruować PCFG G' taką, że $L(G) = L(G')$,*
2. *dla każdego łańcucha (zdania) $w \in L(G)$: $\mathbf{P}_G(w) = \mathbf{P}_{G'}(w)$.*

Dowód.

Część 1. Konstruujemy PCFG $G' = (V, T, R', S, P')$, w której:
— S jest specjalnym dodatkowym symbolem,
— $V = Q \cup \{S\}$,
— T jest tym samym alfabetem symboli końcowych, co w gramatyce G.

Tabela 4.3 pokazuje, jak skonstruować reguły i ich prawdopodobieństwa dla gramatyki G'.

Dodatkowo tworzymy specjalne reguły z symbolem początkowym S gramatyki PCFG po lewej stronie. Dla każdej zmiennej $q \in Q_s$ tworzymy regułę $S \to q$ o prawdopodobieństwie

$$P'(S \to q) = P_s(q) \; . \tag{4.26}$$

Tabela 4.3. Sposób konstrukcji PCFG równoważnej danej PTgBG

Reguły PTgBG	Nowe reguły PCFG $p \in R'$	$P'(p)$
$x \to w : A : r$	$x \to w$	$\sum_{A \in Q,\, r \in \mathcal{R}} P(x \to w : A : r)$
$x \to y : id$ $x \to y : ext(A, r)$	$x \to y$	$P(x \to y : id) +$ $+ \sum_{A \in Q,\, r \in \mathcal{R}} P(x \to y : ext(A, r))$
$x \to y\, z : la(A, r)$ $x \to y\, z : ra(A, r)$ $x \to y\, z : cb(A, r_1, r_2)$	$x \to y\, z$	$\sum_{A \in Q,\, r \in \mathcal{R}} P(x \to y\, z : la(A, r)) +$ $+ \sum_{A \in Q,\, r \in \mathcal{R}} P(x \to y\, z : ra(A, r)) +$ $+ \sum_{A \in Q,\, r_1, r_2 \in \mathcal{R}} P(x \to y\, z : cb(A, r_1, r_2))$

Prawdopodobieństwa reguł przedstawione w tabeli 4.3 są konstruowane w ten sposób, żeby dla każdej zmiennej $x \in Q$ był spełniony następujący warunek:

$$\begin{aligned}
&\sum_{(x \to \omega) \in R'} P'(x \to \omega) = \\
&= \sum_{w \in T} P'(x \to w) + \sum_{y \in Q} P'(x \to y) + \sum_{y,z \in Q} P'(x \to y\, z) = \\
&= \sum_{(x \to w : A : r) \in R} P(x \to w : A : r) + \sum_{(x \to y : I) \in R} P(x \to y : I) + \\
&\quad + \sum_{(x \to yz : I) \in R} P(x \to yz : I) \;=\; 1 \,. \qquad (4.27)
\end{aligned}$$

Mamy też

$$\sum_{(S \to \omega) \in R'} P'(S \to \omega) = \sum_{q \in Q_s} P_s(q) = 1 \,. \qquad (4.28)$$

Dlatego funkcja P' spełnia warunek na prawdopodobieństwo reguły dla PCFG:

$$\sum_{(A \to \omega) \in R'} P'(A \to \omega) = 1 \quad \text{dla każdego } A \in V \,. \qquad (4.29)$$

Część 2. Aby dowieść drugiej części twierdzenia, będziemy potrzebować następującego lematu:

Lemat 4.6. *Niech G i G' będą jak w dowodzie twierdzenia 4.5. Wówczas*

$$\sum_{t' \in \mathcal{T}(T,V,\{u\}):\, \mathrm{yield}(t') = w,\, \mathrm{root}(t') = x} \mathbf{P}_{G'}(t') \;=\; \sum_{t \in \mathcal{T}(T,\mathcal{C},\mathcal{R}):\, \mathrm{yield}(t) = w} \mathbf{P}_G(t, x) \qquad (4.30)$$

dla dowolnej zmiennej $x \in Q$ i każdego łańcucha $w \in T^+$.

Dowód.

Część 1. Najpierw pokażemy, że dla dowolnej zmiennej $x \in Q$ i symbolu terminalnego $a \in T$:

$$\sum_{t' \in \mathcal{T}(T,V,\{u\}):\, \mathrm{yield}(t') = a,\, \mathrm{root}(t') = x} \mathbf{P}_{G'}(t') \;=\; \sum_{t \in \mathcal{T}(T,\mathcal{C},\mathcal{R}):\, \mathrm{yield}(t) = a} \mathbf{P}_G(t, x) \,. \qquad (4.31)$$

Ponieważ a jest pojedynczym symbolem terminalnym, jedynie reguły unarne mogą zostać użyte do budowy drzewa składniowego, zarówno po lewej, jak i po prawej stronie.

Niech $|t|$ oznacza liczbę reguł użytych do zbudowanie drzewa t, tj. długość wyprowadzenia odpowiadającego drzewu t.

Dla wyprowadzeń o długości $|t| = 1$ otrzymujemy

$$\begin{aligned}
&\sum_{t\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t)=a,\, \mathrm{root}(t)=x,\, |t|=1} \mathbf{P}_{G'}(t) = \sum_{t=x[u:a]} \mathbf{P}_{G'}(t) = \\
&\quad = \mathbf{P}_{G'}(x[u:a]) = P'(x \to a) = \\
&\quad = \sum_{A\in Q,\, r\in\mathcal{R}} P(x \to a : A : r) = \sum_{A\in Q,\, r\in\mathcal{R}} \mathbf{P}_G(A[r:a], x) = \\
&\quad = \sum_{t=A[r:a]\colon A\in Q,\, r\in\mathcal{R}} \mathbf{P}_G(t, x) = \sum_{t\in\mathcal{T}(T,\mathcal{C},\mathcal{R})\colon \mathrm{yield}(t)=a,\, |t|=1} \mathbf{P}_G(t, x) \qquad (4.32)
\end{aligned}$$

Przypuśćmy, że następująca zależność zachodzi dla dowolnego wyprowadzenia o długości co najwyżej k:

$$\sum_{t\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t)=a,\, \mathrm{root}(t)=x,\, |t|\leqslant k} \mathbf{P}_{G'}(t') = \sum_{t\in\mathcal{T}(T,\mathcal{C},\mathcal{R})\colon \mathrm{yield}(t)=a,\, |t|\leqslant k} \mathbf{P}_G(t, x)\ . \qquad (4.33)$$

Wówczas dla dowolnego wyprowadzenia o długości co najwyżej $k+1$ otrzymujemy

$$\begin{aligned}
&\sum_{t\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t)=w,\, \mathrm{root}(t)=x,\, |t|\leqslant k+1} \mathbf{P}_{G'}(t) = \\
&\quad = \sum_{t=x[u:w]} \mathbf{P}_{G'}(t) + \sum_{t=x[u:t_1]\colon \mathrm{yield}(t_1)=w,\, |t_1|\leqslant k} \mathbf{P}_{G'}(t) = \\
&\quad = \mathbf{P}_{G'}(x[u:w]) + \sum_{t_1\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t_1)=w,\, |t_1|\leqslant k} P'(x \to \mathrm{root}(t_1)) \cdot \mathbf{P}_{G'}(t_1) = \\
&\quad = P'(x \to w) + \\
&\qquad + \sum_{y\in Q} \left(P'(x \to y) \cdot \sum_{t_1\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t_1)=w,\, \mathrm{root}(t_1)=y,\, |t_1|\leqslant k} \mathbf{P}_{G'}(t_1) \right) = \\
&\quad = \sum_{A\in Q,\, r\in\mathcal{R}} P(x \to w : A : r) + \sum_{y\in Q} \Bigg(\Bigg(P(x \to y : id) + \\
&\qquad + \sum_{A\in Q,\, r\in\mathcal{R}} P(x \to y : ext(A, r)) \Bigg) \cdot \\
&\qquad \cdot \sum_{t_1\in\mathcal{T}(T,\mathcal{C},\mathcal{R})\colon \mathrm{yield}(t_1)=w,\, |t_1|\leqslant k} \mathbf{P}_G(t_1, y) \Bigg) = \\
&\quad = \sum_{A\in Q,\, r\in\mathcal{R}} P(x \to w : A : r) + \\
&\qquad + \sum_{y\in Q} \left(\left(\sum_{I \text{ unary}} P(x \to y : I) \right) \cdot \sum_{t_1\in\mathcal{T}(T,\mathcal{C},\mathcal{R})\colon \mathrm{yield}(t_1)=w,\, |t_1|\leqslant k} \mathbf{P}_G(t_1, y) \right) =
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{A\in Q,\, r\in\mathcal{R}} \mathbf{P}_G(A[r:w], x) + \\
&\quad + \sum_{t=A[r:t_1]\colon A\in Q,\, r\in\mathcal{R},\, \mathrm{yield}(t_1)=w} \;\sum_{t_1\in\mathcal{T}(T,\mathcal{C},\mathcal{R})\colon t=I(t_1),\, |t_1|\leqslant k} P(x \to y : I) \cdot \\
&\quad \cdot \mathbf{P}_G(t_1, y) = \\
&= \sum_{t=A[r:w]\colon A\in Q,\, r\in\mathcal{R}} \mathbf{P}_G(t, x) + \sum_{t=A[r:t_1]\colon A\in Q,\, r\in\mathcal{R},\, \mathrm{yield}(t_1)=w,\, |t_1|\leqslant k} \mathbf{P}_G(t, x) = \\
&= \sum_{t\in\mathcal{T}(T,\mathcal{C},\mathcal{R})\colon \mathrm{yield}(t)=w,\, |t|\leqslant k+1} \mathbf{P}_G(t, x)
\end{aligned} \tag{4.34}
$$

Na mocy zasady indukcji matematycznej otrzymujemy

$$
\sum_{t\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t)=a,\, \mathrm{root}(t)=x,\, |t|\leqslant n} \mathbf{P}_{G'}(t) \;=\; \sum_{t\in\mathcal{T}(T,\mathcal{C},\mathcal{R})\colon \mathrm{yield}(t)=a,\, |t|\leqslant n} \mathbf{P}_G(t, x) \tag{4.35}
$$

dla każdego $n \in \mathbb{N}_+$ i dowolnego $a \in T$.

Ponieważ każde wyprowadzenie jest skończone, otrzymujemy

$$
\sum_{t\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t)=a,\, \mathrm{root}(t)=x} \mathbf{P}_{G'}(t) \;=\; \sum_{t\in\mathcal{T}(T,\mathcal{C},\mathcal{R})\colon \mathrm{yield}(t)=a} \mathbf{P}_G(t, x) \tag{4.36}
$$

dla dowolnego $a \in T$, co odpowiada przypadkowi tezy lematu dla $w = a$, $|w| = 1$.

Część 2. Teraz użyjemy indukcji po długości łańcucha w, aby dowieść tezy lematu.

Przypuśćmy, że

$$
\sum_{t\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t)=w,\, \mathrm{root}(t)=x} \mathbf{P}_{G'}(t) \;=\; \sum_{t\in\mathcal{T}(T,\mathcal{C},\mathcal{R})\colon \mathrm{yield}(t)=w} \mathbf{P}_G(t, x) \tag{4.37}
$$

dla każdego $x \in Q$ i $w \in T^+$, $|w| \leqslant k$.

Wówczas dla dowolnego $1 < |w| \leqslant k + 1$:

$$
\begin{aligned}
&\sum_{t\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t)=w,\, \mathrm{root}(t)=x} \mathbf{P}_{G'}(t) = \\
&= \sum_{t=x[u:t_1]\colon \mathrm{yield}(t_1)=w} \mathbf{P}_{G'}(t) + \sum_{t=x[u:t_1,u:t_2]\colon \mathrm{yield}(t_1)\mathrm{yield}(t_2)=w} \mathbf{P}_{G'}(t) = \\
&= \sum_{t_1\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t_1)=w} P'(x \to \mathrm{root}(t_1)) \cdot \mathbf{P}_{G'}(t_1) + \\
&\quad + \sum_{t_1,t_2\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t_1)\mathrm{yield}(t_2)=w} P'(x \to \mathrm{root}(t_1)\,\mathrm{root}(t_2)) \cdot \\
&\quad \cdot \mathbf{P}_{G'}(t_1) \cdot \mathbf{P}_{G'}(t_2) =
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{y \in Q} \left(P'(x \to y) \cdot \sum_{t_1 \in \mathcal{T}(T,V,\{u\}):\, \mathrm{yield}(t_1)=w,\, \mathrm{root}(t_1)=y} \mathbf{P}_{G'}(t_1) \right) + \\
&\quad + \sum_{y,z \in Q,\, w_1 w_2 = w} \left(P'(x \to y\, z) \cdot \sum_{t_1 \in \mathcal{T}(T,V,\{u\}):\, \mathrm{yield}(t_1)=w_1,\, \mathrm{root}(t_1)=y} \mathbf{P}_{G'}(t_1) \cdot \right. \\
&\quad \left. \cdot \sum_{t_2 \in \mathcal{T}(T,V,\{u\}):\, \mathrm{yield}(t_2)=w_2,\, \mathrm{root}(t_2)=z} \mathbf{P}_{G'}(t_1) \right) = \\
&= \sum_{y \in Q} \left(\left(P(x \to y : id) + \sum_{A \in Q,\, r \in \mathcal{R}} P(x \to y : ext(A,r)) \right) \cdot \right. \\
&\quad \left. \cdot \sum_{t_1 \in \mathcal{T}(T,V,\{u\}):\, \mathrm{yield}(t_1)=w,\, \mathrm{root}(t_1)=y} \mathbf{P}_{G'}(t_1) \right) + \\
&\quad + \sum_{y,z \in Q,\, w_1 w_2 = w} \left(\left(\sum_{A \in Q,\, r \in \mathcal{R}} P(x \to y\, z : la(A,r)) + \right. \right. \\
&\quad + \sum_{A \in Q,\, r \in \mathcal{R}} P(x \to y\, z : ra(A,r)) + \\
&\quad \left. \left. + \sum_{A \in Q,\, r_1, r_2 \in \mathcal{R}} P(x \to y\, z : cb(A,r_1,r_2)) \right) \cdot \right. \\
&\quad \left. \cdot \sum_{t_1 \in \mathcal{T}(T,V,\{u\}):\, \mathrm{yield}(t_1)=w_1} \mathbf{P}_G(t_1, y) \cdot \sum_{t_2 \in \mathcal{T}(T,V,\{u\}):\, \mathrm{yield}(t_2)=w_2} \mathbf{P}_G(t_1, z) \right) = \\
&= \sum_{y \in Q} \left(\left(\sum_{I \text{ unary}} P(x \to y : I) \right) \cdot \sum_{t_1 \in \mathcal{T}(T,V,\{u\}):\, \mathrm{yield}(t_1)=w,\, \mathrm{root}(t_1)=y} \mathbf{P}_{G'}(t_1) \right) + \\
&\quad + \sum_{y,z \in Q,\, w_1 w_2 = w} \left(\left(\sum_{I \text{ binarne}} P(x \to y\, z) \right) \cdot \right. \\
&\quad \left. \cdot \sum_{t_1 \in \mathcal{T}(T,V,\{u\}):\, \mathrm{yield}(t_1)=w_1} \mathbf{P}_G(t_1, y) \cdot \sum_{t_2 \in \mathcal{T}(T,V,\{u\}):\, \mathrm{yield}(t_2)=w_2} \mathbf{P}_G(t_1, z) \right) = \\
&= \sum_{t = A[r:t_1]:\, A \in Q, r \in \mathcal{R}, \mathrm{yield}(t_1)=w} \; \sum_{t_1 \in \mathcal{T}(T,\mathcal{C},\mathcal{R}):\, t = I(t_1)} P(x \to y : I) \cdot \mathbf{P}_G(t_1, y) + \\
&\quad + \sum_{t = A[r_1:t_1, r_2:t_2]:\, A \in Q, r_1, r_2 \in \mathcal{R}, \mathrm{yield}(t_1)\mathrm{yield}(t_2)=w} \; \sum_{t_1, t_2 \in \mathcal{T}(T,\mathcal{C},\mathcal{R}):\, t = I(t_1,t_2)} \\
&\qquad P(x \to y\, z : I) \cdot \mathbf{P}_G(t_1, y) \cdot \mathbf{P}_G(t_2, z) = \\
&= \sum_{t = A[r:t_1]:\, A \in Q, r \in \mathcal{R}, \mathrm{yield}(t_1)=w} P_G(t, x) + \\
&\quad + \sum_{t = A[r_1:t_1, r_2:t_2]:\, A \in Q, r_1, r_2 \in \mathcal{R}, \mathrm{yield}(t_1)\mathrm{yield}(t_2)=w} P_G(t, x) = \\
&= \sum_{t \in \mathcal{T}(T,\mathcal{C},\mathcal{R}):\, \mathrm{yield}(t)=w} \mathbf{P}_G(t, x)\ . \qquad (4.38)
\end{aligned}
$$

Na mocy zasady indukcji otrzymujemy tezę lematu. □

Na mocy lematu 4.6, dla dowolnego łańcucha (zdania) w mamy:

$$\begin{aligned}
\mathbf{P}_{G'}(w) \;&=\; \sum_{t\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t)=w,\,\mathrm{root}(t)=S} \mathbf{P}_{G'}(t) = \\
&= \sum_{t'\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t')=w,\,\mathrm{root}(t')\in Q_s} \mathbf{P}_{G'}(S[u:t']) = \\
&= \sum_{t'\in\mathcal{T}(T,V,\{u\})\colon \mathrm{yield}(t')=w,\,\mathrm{root}(t')\in Q_s} \mathbf{P}_{G'}(t')\cdot P'(S\to\mathrm{root}(t')) = \\
&= \sum_{t\in\mathcal{T}(T,\mathcal{C},\mathcal{R})\colon \mathrm{yield}(t)=w,\,q\in Q_s} \mathbf{P}_{G}(t,q)\cdot P_s(q) \;=\; \mathbf{P}_G(w) \qquad (4.39)
\end{aligned}$$

To dowodzi drugiej części twierdzenia. □

Rozdział 5

Istniejące wydajne algorytmy parsowania wykorzystujące wagi

Opracowano wiele algorytmów parsingu probabilistycznych gramatyk bezkontekstowych. Stolcke [61] zaadaptował w tym celu parser Earleya. Do parsowania PCFG w postaci normalnej Chomsky'ego można również użyć probabilistycznej wersji algorytmu Cocke'a-Youngera-Kasamiego [27]. Algorytmy te są przedstawione w podrozdziale 5.1.

Aby przyspieszyć proces znajdowania drzewa Viterbiego, można zastosować algorytmy przeszukiwania stosowane w teorii grafów (podrozdział 5.2): Caraballo i Charniak [6] używają do tego celu algorytmu *best-first*, natomiast Ratnaparkhi [44] i Roark [45] — strategii *beam search*. Klein i Manning [30] przedstawiają wydajny algorytm znajdowania w PCFG drzewa Viterbiego dla danego zdania za pomocą algorytmu opartego na algorytmie A* znajdowania najkrótszej drogi w grafie.

Do parsowania gramatyk probabilistycznych można wykorzystać też metody typu *coarse-to-fine*, które wykorzystują gramatyki o różnym stopniu skomplikowania (podrozdział 5.3), wykorzystywane w tłumaczeniu automatycznym metody typu *cube pruning* (podrozdział 5.4) i inne (podrozdział 5.5).

5.1. Algorytmy klasyczne

Do parsowania probabilistycznych gramatyk bezkontekstowych można wykorzystać modyfikacje klasycznych algorytmów parsowania CFG, takich jak parser Earleya czy parser CYK.

5.1.1. Algorytm Earleya

W parserze Earleya symbole łańcucha wejściowego $w = w_1 \dots w_n$ przetwarzane są kolejno od lewej do prawej. W tym celu wykorzystuje się $(n + 1)$-elementową tablicę, która w czasie działania algorytmu jest sukcesywnie zapełniana od lewej do prawej. Dla każdej pozycji symbolu w łańcuchu wejściowym tablica zawiera listę stanów reprezentujących wygenerowane do tej pory częściowe drzewa składniowe. Gdy algorytm dociera do końca łańcucha,

tablica zawiera zakodowane wszystkie możliwe drzewa składniowe dla tego łańcucha.

Główny mechanizm algorytmu zawierają trzy procedury, które stosowane są w zależności od rodzaju aktualnie przetwarzanego stanu: przewidywanie, wczytywanie i uzupełnianie.

Modyfikacja algorytmu Earleya do parsowania probabilistycznych gramatyk bezkontekstowych została opracowana przez Stolckego [61]. W tej wersji algorytmu tablica oprócz stanów parsera przechowuje prawdopodobieństwo Viterbiego każdego stanu, czyli cząstkowe prawdopodobieństwo maksymalne obliczone na danym etapie działania algorytmu.

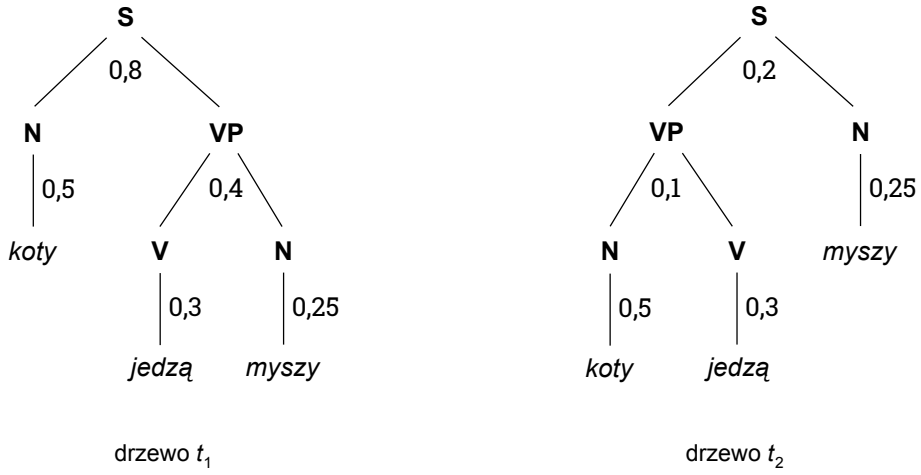
Kiedy wykonywana jest jedna z trzech procedur algorytmu, oblicza się nową wartość prawdopodobieństwa Viterbiego na podstawie prawdopodobieństw stanów będących argumentami procedur. Gdy algorytm dotrze do końca łańcucha wejściowego, maksymalne prawdopodobieństwo łańcucha wejściowego można odczytać jako prawdopodobieństwo Viterbiego odpowiedniego stanu. Aby odnaleźć drzewo Viterbiego danego łańcucha należy na zakończenie algorytmu wykonać jeszcze jedną dodatkową operację, a mianowicie prześledzić kolejne stany, które złożyły się na taką wartość prawdopodobieństwa łańcucha.

Złożoność czasowa algorytmu Earleya dla PCFG zależy w sposób sześcienny zarówno od długości łańcucha, jak i rozmiaru gramatyki i wynosi $O(n^3 \cdot m^3)$ dla łańcucha o długości n i gramatyki o m symbolach pomocniczych.

5.1.2. Algorytm CYK[1]

Algorytm Cocke'a-Youngera-Kasamiego jest jednym z najbardziej popularnych algorytmów parsingu dla gramatyk bezkontekstowych. Opiera się on na paradygmacie programowania dynamicznego. Algorytm CYK korzysta z faktu, że jeżeli łańcuch w można rozłożyć na dwa podłańcuchy $w = xy$, to drzewa składniowe podłańcuchów można wykorzystać do znalezienia drzewa składniowego łańcucha w. Główną strukturą, na której działa algorytm, jest trójwymiarowa tablica o wymiarach $n \times n \times m$, gdzie n jest długością parsowanego łańcucha, zaś m jest liczbą symboli pomocniczych gramatyki. W tablicy przechowuje się informacje, z których symboli gramatyki można wyprowadzić dane podłańcuchy łańcucha wejściowego oraz z których poddrzew można skonstruować drzewa składniowe tych podłańcuchów. Przechowywane są również wskaźniki, które umożliwiają rekonstrukcję drzew składniowych z odpowiednich poddrzew.

[1]Na podstawie[27].



Rysunek 5.1. Dwa różne drzewa wyprowadzenia łańcucha *koty jedzą myszy* . Drzewo t_1 ma prawdopodobieństwo 0.012. Drzewo t_2 ma prawdopodobieństwo 0.00075. Drzewo t_1 jest zatem drzewem Viterbiego tego łańcucha.

W wersji algorytmu CYK dla PCFG oprócz powyższych informacji tablica przechowuje cząstkowe prawdopodobieństwa wyprowadzeń poszczególnych podłańcuchów. Po zakończeniu działania algorytmu w odpowiedniej komórce można znaleźć maksymalne prawdopodobieństwo drzewa składniowego oraz zrekonstruować drzewo Viterbiego wejściowego łańcucha.

Przykład 5.1 (algorytm CYK parsowania probabilistycznych gramatyk bezkontekstowych)**.** Niech dana będzie probabilistyczna gramatyka bezkontekstowa w postaci normalnej Chomsky'ego z przykładu 2.20 na stronie 20.

Niech wejściowym łańcuchem będzie zdanie $w =$ *koty jedzą myszy* . To zdanie posiada dwa różne drzewa wyprowadzenia (rys. 5.1).

Na początku tworzymy pustą trójwymiarową tablicę t o wymiarach $|w| \times |w| \times |V|$, czyli $3 \times 3 \times 4$:

	1 (S,N,VP,V)	2 (S,N,VP,V)	3 (S,N,VP,V)
1	$(0,0,0,0)$	$(0,0,0,0)$	$(0,0,0,0)$
wskaźniki	(—,—,—,—)	(—,—,—,—)	(—,—,—,—)
2	$(0,0,0,0)$	$(0,0,0,0)$	$(0,0,0,0)$
wskaźniki	(—,—,—,—)	(—,—,—,—)	(—,—,—,—)
3	$(0,0,0,0)$	$(0,0,0,0)$	$(0,0,0,0)$
wskaźniki	(—,—,—,—)	(—,—,—,—)	(—,—,—,—)
	koty	*jedzą*	*myszy*

Teraz znajdujemy reguły, których następniki są symbolami końcowymi występującymi w łańcuchu wejściowym:

$$P(\, N \,\rightarrow\, \textit{koty}\,) = 0.5\;,$$
$$P(\, V \,\rightarrow\, \textit{jedzą}\,) = 0.3\;,$$
$$P(\, N \,\rightarrow\, \textit{myszy}\,) = 0.25\;.$$

Prawdopodobieństwa tych reguł wpisujemy w odpowiednie miejsca tablicy:

	1 (S,**N**,VP,V)	2 (S,N,VP,**V**)	3 (S,**N**,VP,V)
1	$(0, \mathbf{0.5}, 0, 0)$	$(0,0,0,0)$	$(0,0,0,0)$
wskaźniki	(—,/,—,—)	(—,—,—,—)	(—,—,—,—)
2	$(0,0,0,0)$	$(0,0,0,\mathbf{0.3})$	$(0,0,0,0)$
wskaźniki	(—,—,—,—)	(—,—,—,/)	(—,—,—,—)
3	$(0,0,0,0)$	$(0,0,0,0)$	$(0,\mathbf{0.25},0,0)$
wskaźniki	(—,—,—,—)	(—,—,—,—)	(—,/,—,—)
	koty	*jedzą*	*myszy*

Kolejnym krokiem jest znalezienie symboli, które rozwijają się w podłańcuchy długości 2:

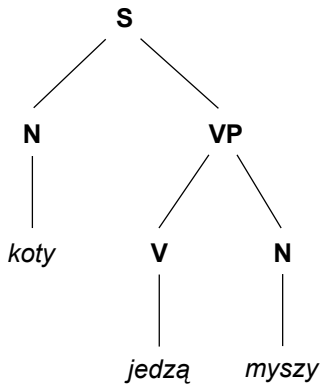
$$VP \,\rightarrow\, N\ V\ ,$$
$$p := 0.5 \cdot 0.3 \cdot P(\, VP \,\rightarrow\, N\ V\,) = 0.5 \cdot 0.3 \cdot 0.1 = 0.015\;;$$
$$VP \,\rightarrow\, V\ N\ ,$$
$$p := 0.3 \cdot 0.25 \cdot P(\, VP \,\rightarrow\, V\ N\,) = 0.3 \cdot 0.25 \cdot 0.4 = 0.03\;.$$

Znaleźliśmy dwa takie symbole. Aktualizujemy wartości w tablicy i przypisujemy odpowiednie wskaźniki:

	1 (S,*N*,VP,V)	2 (S,N,**VP**,*V*)	3 (S,*N*,**VP**,V)
1	$(0, \textit{0.5}, 0, 0)$	$(0,0,\mathbf{0.015},0)$	$(0,0,0,0)$
wskaźniki	(—,/,—,—)	(—,—,**[1,1,N;2,2,V]**,—)	(—,—,—,—)
2	$(0,0,0,0)$	$(0,0,0,\textit{0.3})$	$(0,0,\mathbf{0.03},0)$
wskaźniki	(—,—,—,—)	(—,—,—,/)	(—,—,**[2,2,V;3,3,N]**,—)
3	$(0,0,0,0)$	$(0,0,0,0)$	$(0,\textit{0.25},0,0)$
wskaźniki	(—,—,—,—)	(—,—,—,—)	(—,/,—,—)
	koty	*jedzą*	*myszy*

W ostatnim kroku znajdujemy symbole, które rozwijają się w podłańcuchy długości 3, czyli w naszym przypadku — w cały łańcuch. Znajdujemy dwie pasujące reguły, których poprzednikiem jest symbol S:

$$S \,\rightarrow\, N\ VP\,,$$



Rysunek 5.2. Drzewo Viterbiego uzyskane za pomocą algorytmu CYK.

$$p_1 := 0.5 \cdot 0.03 \cdot P(\, S \;\rightarrow\; N \;\; VP\,) = 0.5 \cdot 0.03 \cdot 0.8 = 0.012;$$

$S \;\rightarrow\; VP \;\; N\,,$

$$p_2 := 0.015 \cdot 0.25 \cdot P(\, S \;\rightarrow\; VP \;\; N\,) = 0.015 \cdot 0.25 \cdot 0.2 = 0.00075.$$

Ponieważ $p_1 > p_2$, zatem w komórce $t_{1,3,S}$ znajdzie się wartość p_1 oraz wskaźnik na poddrzewa o korzeniach w N i VP:

	1 (S,*N*,VP,V)	2 (S,N,VP,V)	3 (**S**,N,*VP*,V)
1	$(0, \mathit{0.5}, 0, 0)$	$(0, 0, 0.015, 0)$	$(\mathbf{0.012}, 0, 0, 0)$
wskaźniki	(—,/,—,—)	(—,—,[1,1,N;2,2,V],—)	([**1,1,N;2,3,VP**],—,—,—)
2	$(0, 0, 0, 0)$	$(0, 0, 0, 0.3)$	$(0, 0, \mathit{0.03}, 0)$
wskaźniki	(—,—,—,—)	(—,—,—,/)	(—,—,[2,2,V;3,3,N],—)
3	$(0, 0, 0, 0)$	$(0, 0, 0, 0)$	$(0, 0.25, 0, 0)$
wskaźniki	(—,—,—,—)	(—,—,—,—)	(—,/,—,—)
	koty	*jedzą*	*myszy*

Teraz możemy odczytać drzewo Viterbiego podążając za wskaźnikami z komórki $t_{1,3,S}$ (rys. 5.2).

5.2. Algorytmy oparte na algorytmach grafowych

Przestrzeń przeszukiwania możliwych drzew wyprowadzeń dla danego łańcucha można traktować jako hipergraf z wagami. Wystarczy potraktować łańcuchy jako wierzchołki hipergrafu, stosowane do nich reguły produkcji jako krawędzie, a prawdopodobieństwa reguł (a dokładnie odwrotności logarytmów prawdopodobieństw) — jako wagi krawędzi. Wówczas drzewa wyprowadzeń stają się ścieżkami w tym hipergrafie.

W takiej interpretacji zagadnienie znalezienia drzewa Viterbiego dla danego łańcucha można zastąpić problemem znalezienia najkrótszej ścieżki w odpowiednim hipergrafie. Z tego powodu do parsowania gramatyk probabilistycznych (ogólniej: gramatyk z wagami) można wykorzystać odpowiednio zmodyfikowane algorytmy grafowe służące do znajdowania najkrótszych ścieżek.

5.2.1. *Best-first*

Metoda *best-first* ('najpierw najlepszy') to algorytm przeszukiwania grafu polegający na tym, że najbardziej obiecujące wierzchołki bądź krawędzie odwiedza się w pierwszej kolejności. Wykorzystuje się przy tym daną z góry funkcję oceny (*figure-of-merit*, 'miarę wartości'), która dla każdego wierzchołka określa, jak bardzo wart jest on odwiedzenia.

W przypadku parsingu tablicowego oznacza to, że funkcja oceny decyduje o kolejności przetwarzania krawędzi znajdujących się w agendzie. Funkcje oceny powiązane są z prawdopodobieństwami reguł. Można w tym celu wykorzystać takie miary, jak prawdopodobieństwo wewnętrzne podłańcucha, unormowane prawdopodobieństwo wewnętrzne czy oszacowania trigramowe. Szczegółową dyskusję stosowanych rozwiązać można znaleźć u Caraballo i Charniaka [6].

Zaletą metod *best-first* jest istotne zmniejszenie nakładu pracy wykonywanej przez algorytm dzięki temu, że składniki o niskiej mierze wartości nie są przetwarzane. Wadą jest to, że algorytm nie daje gwarancji znalezienia drzewa Viterbiego, ponieważ jego elementy mogą się znajdować wśród odrzuconych przez algorytm (nie zawsze rozwiązanie globalnie optymalne jest również lokalnie najlepsze).

Asymptotyczna złożoność czasowa zależy od zastosowanej funkcji oceny, ale w ogólnym przypadku nie jest niższa niż sześcienna względem długości łańcucha.

5.2.2. *Beam search*

Ratnaparkhi [44] i Roark [45] opisują strategię *beam search* ('przeszukiwanie wiązkowe'). Jest to rozszerzenie metody *best-first*, które polega na tym, że ogranicza się z góry liczbę ścieżek („szerokość wiązki"), które brane są pod uwagę jako najbardziej obiecujące.

W przypadku algorytmów parsingu tablicowego realizowane jest to w ten sposób, że liczba elementów przechowywanych w agendzie jest ograniczona. Elementy o najmniejszej wartości funkcji oceny są odrzucane podczas dodawania nowo skonstruowanych krawędzi do agendy.

Podobnie jak w przypadku strategii *best-first*, i z tego samego powodu, algorytm nie zawsze znajduje drzewo Viterbiego danego łańcucha.

Złożoność czasowa algorytmu zależy od szerokości wiązki — im węższa wiązka, tym mniejsza złożoność algorytmu.

5.2.3. Algorytm A*

Algorytm A* — opis działania w ogólnym przypadku

Algorytm A* służy do znajdowania najkrótszej ścieżki między dwoma danymi wierzchołkami grafu z wagami, grafu skierowanego z wagami bądź multigrafu skierowanego z wagami. W celu znalezienia najkrótszej ścieżki algorytm A* wykorzystuje dodatkową funkcję $h\colon U \to \mathbb{R}$ (gdzie U oznacza zbiór wierzchołków grafu), nazywaną *heurystyką* lub *funkcją oceny*. Heurystyka $h(x)$ określa, jaka jest przewidywana odległość od wierzchołka x do wierzchołka docelowego. Na ogół jako wartość funkcji h przyjmuje się pewne oszacowanie odległości między x a wierzchołkiem docelowym, które może być dane z góry lub po prostu łatwe do obliczenia. Właściwości algorytmu zależą od rodzaju użytej heurystyki.

Algorytm A* zawsze znajduje najkrótszą ścieżkę między danymi dwoma wierzchołkami grafu pod warunkiem, że funkcja heurystyki h spełnia warunek *dopuszczalności*: $h(x) \leqslant d(x, y) + h(y)$ dla dowolnych dwóch wierzchołków x, y (gdzie $d(x, y)$ oznacza rzeczywistą odległość między wierzchołkami x i y).

Zasada działania algorytmu A* opiera się na dążeniu do maksymalizacji funkcji

$$f(x) = g(x) + h(x)\ , \tag{5.1}$$

gdzie $g(x)$ oznacza długość do tej pory znalezionej ścieżki z wierzchołka początkowego do wierzchołka x, a $h(x)$ jest wartością heurystyki dla wierzchołka x.

Pseudokod algorytmu A* dla grafów jest przedstawiony w ramce zatytułowanej „Algorytm 1”.

Algorytm A* w parsingu

Algorytm A* może służyć do parsowania probabilistycznych gramatyk bezkontekstowych. Szczegółowy opis wykorzystania algorytmu A* w parsingu PCFG wraz z propozycjami heurystyk podali Klein i Manning [30].

Algorytm parsingu A* jest odmianą parsera tablicowego. Charakterystycznym elementem, który odróżnia go od innych algorytmów tablicowych, jest to, że w pierwszej kolejności z agendy wyjmowane są elementy o największej wartości funkcji f (ze wzoru (5.1)). Agendę dla parsera A* można zaimplementować jako kolejkę priorytetową.

Algorytm 1: A*

wejście: graf (graf skierowany, multigraf skierowany) z wagami Γ o zbiorze wierzchołków U, zbiorze krawędzi E i funkcji wagi W; wierzchołki u i v, między którymi mamy znaleźć najkrótszą drogę; funkcja heurystyki $h\colon U \to \mathbb{R}$

zbiór $C := \emptyset$;
zbiór $D := \{u\}$;
$g(u) := 0$;
$f(u) := g(u) + h(u) = 0 + h(u) = h(u)$;
dopóki $D \neq \emptyset$ **wykonuj:**
  $x_0 := \arg\min_{x \in D} f(x)$; `/* znajdź w zbiorze` D `wierzchołek, dla którego wartość funkcji` f `jest najmniejsza */`
  jeżeli $x_0 = v$ **to:**
    zakończ i zwróć *znalezioną ścieżkę (podążając za* wskaźnik*ami)* ;
  $D := D \setminus \{x_0\}$;
  $C := C \cup \{x_0\}$;
  dla każdego *wierzchołka* $y \notin C$ *sąsiadującego z* x_0 **wykonaj:**
    e := (najkrótsza — w przypadku multigrafu) krawędź łącząca wierzchołek x_0 z wierzchołkiem y;
    g' := $g(x_0) + W($e$)$;
    k **jeżeli** $y \notin D$ **to:**
      $D := D \cup \{y\}$;
      wskaźnik$(y) := x_0$;
      $g(y) :=$ g';
      $f(y) := g(y) + h(y)$;
    jeżeli $y \in D$ *oraz* g' $< g(y)$ **to:**
      wskaźnik$(y) := x_0$;
      $g(y) :=$ g';
      $f(y) := g(y) + h(y)$;
zakończ i zwróć *komunikat: nie znaleziono ścieżki z* u *do* v ;

Na podobnych zasadach opierają się algorytmy przeszukiwania *best-first* i *beam search*, ale w przeciwieństwie do nich algorytm A* gwarantuje znalezienie drzewa Viterbiego. Złożoność czasowa znalezienia jakiegokolwiek poprawnego drzewa składniowego wejściowego łańcucha jest asymptotycznie sześcienna względem długości łańcucha. Jeżeli sześcienna złożoność ma zostać zachowana dla problemu znalezienia drzewa Viterbiego łańcucha, musi być spełniony warunek, by dla każdej krawędzi wszystkie krawędzie, które służą do jej budowy, zostały wyjęte z agendy przed wyjęciem tejże krawędzi. W przeciwnym wypadku mogłaby zajść konieczność przebudowania całego drzewa, a w konsekwencji wzrosłaby złożoność czasowa znajdowania drzewa Viterbiego dla danego łańcucha.

Aby zapewnić, by ten warunek został spełniony, w algorytmie parsingu A* stosuje się odpowiednie oszacowania, na których opiera się definicje heurystyk. Możemy przy tym wyróżnić heurystyki oparte na oszacowaniach kontekstu oraz heurystyki oparte na rzutowaniu gramatyki, a także różnorodne ich kombinacje.

Heurystyki oparte na oszacowaniach kontekstu to funkcje szacujące cząstkowe prawdopodobieństwo Viterbiego na podstawie wiedzy o sąsiedztwie bieżącej krawędzi. Możliwe jest tu całe spektrum oszacowań począwszy od oszacowania NULL, które odpowiada zerowej wiedzy o kontekście (jest najprostsze do obliczenia, ale w praktyce bezużyteczne), aż do oszacowania TRUE, które jest najdokładniejsze, gdyż odpowiada pełnej wiedzy o kontekście, ale za to jest niepraktyczne, ponieważ jest równoważne wykonywaniu całego algorytmu dla każdej krawędzi z osobna. Ponadto istnieje cały wachlarz oszacowań pośrednich, które biorą pod uwagę np. liczbę symboli po prawej i lewej stronie bieżącej krawędzi czy etykiety jej sąsiadów.

Heurystyki oparte na rzutowaniu gramatyki korzystają z uproszczonej gramatyki (łatwiejszej do parsowania) do obliczenia wartości heurystyki. Przypomina to pod pewnymi względami opisane w kolejnym podrozdziale (5.3) algorytmy *coarse-to-fine*.

Przykład 5.2 (parsing wykorzystujący algorytm A* z heurystyką SX)**.** Niech dana będzie gramatyka bezkontekstowa z przykładu 2.20 na stronie 20 oraz łańcuch *koty jedzą myszy* .

Ponieważ stosując podejście wstępujące możemy w prosty sposób obliczać dokładną wartość wagi β, zamiast oszacowania b będziemy stosować dokładną wartość β.

Najpierw przygotowujemy pustą tablicę i agendę. Do agendy wkładamy krawędzie *koty* $[0,1]$, *jedzą* $[1,2]$ oraz *myszy* $[2,3]$. Dla każdej krawędzi obliczamy wartości β, a i $\beta+a$. Wartości wagi β dla każdej z krawędzi *koty* $[0,1]$,

jedzą $[1,2]$ oraz *myszy* $[2,3]$ są równe 0, ponieważ krawędzie te odpowiadają symbolom terminalnym, a prawdopodobieństwo wyprowadzenia symbolu terminalnego z tego samego symbolu terminalnego wynosi 1.

Wartości $a(e)$ obliczamy jako logarytm największego z prawdopodobieństw drzew wyprowadzenia, które zawierają krawędź e.

Aktualna zawartość agendy uwidoczniona jest w poniższej tabeli (wytłuszczono nowo dodane krawędzie):

agenda			
krawędź e	$\beta(e)$	$a(e)$	$(\beta + a)(e)$
***koty* [0,1]**	$\log(1) = 0$	$\log(0.0056) = -4.16$	$\mathbf{-4.16}$
***jedzą* [1,2]**	$\log(1) = 0$	$\log(0.0024) = -5.38$	-5.38
***myszy* [2,3]**	$\log(1) = 0$	$\log(0.0028) = -5.16$	-5.16

Jako pierwszą z agendy wyjmujemy krawędź *koty* $[0,1]$, ponieważ ma największą wartość oszacowania $\beta+a$. Dodajemy tę krawędź do tablicy. Teraz zawartość tablicy jest następująca (wytłuszczono nowo dodaną krawędź):

tablica	
krawędź e	$\beta(e)$
***koty* [0,1]**	$\log(1) = 0$

Wypiszmy teraz wszystkie reguły gramatyki, które po prawej stronie mają symbol *koty*:

— $N \rightarrow$ *koty*.

Znaleźliśmy tylko jedną taką regułę. Może być ona zastosowana do przetwarzanej aktualnie krawędzi, ponieważ jest to reguła unarna (ma po prawej stronie tylko jeden symbol).

Konstruujemy zatem nową krawędź $N\,[0,1]$. Dodajemy ją do agendy i obliczamy wartość oszacowania:

$$\beta(N\,[0,1]) = \log P(N \rightarrow \textit{koty}) = \log(0.5) = -1\ .$$

Wartość $a(N\,[0,1])$ równa jest logarytmowi prawdopodobieństwa najbardziej prawdopodobnego drzewa wyprowadzenia łańcucha $N\,c\,d$ z symbolu początkowego S gramatyki, gdzie c i d są dowolnymi symbolami końcowymi.

Poniższa tabela przedstawia zawartość agendy po wykonaniu tego kroku:

agenda			
krawędź e	$\beta(e)$	$a(e)$	$(\beta + a)(e)$
jedzą $[1,2]$	$\log(1) = 0$	$\log(0.0024) = -5.38$	-5.38
myszy $[2,3]$	$\log(1) = 0$	$\log(0.0028) = -5.16$	-5.16
$\boldsymbol{N}$ **[0,1]**	$\log(0.5) = -1.00$	$\log(0.112) = -3.16$	$\mathbf{-4.16}$

Znów wyjmujemy z agendy tę krawędź, dla której wartość $\beta + a$ jest największa, czyli tym razem jest to $N\,[0,1]$. Krawędź $N\,[0,1]$ wstawiamy do tablicy:

tablica	
krawędź e	$\beta(e)$
koty $[0,1]$	$\log(1) = 0$
$\boldsymbol{N}$ **[0,1]**	$\log(0.5) = -1.00$

Ponieważ żadna z produkcji gramatyki G nie ma po prawej stronie pojedynczego symbolu N, nie zostają utworzone żadne nowe krawędzie, które można by było dorzucić do agendy.

Teraz w agendzie ponownie znajdujemy krawędź o największej wadze:

agenda			
krawędź e	$\beta(e)$	$a(e)$	$(\beta + a)(e)$
jedzą $[1,2]$	$\log(1) = 0$	$\log(0.0024) = -5.38$	-5.38
myszy $[2,3]$	$\log(1) = 0$	$\log(0.0028) = -5.16$	$\mathbf{-5.16}$

Wyjmujemy ją z agendy i wstawiamy do tablicy:

tablica	
krawędź e	$\beta(e)$
koty $[0,1]$	$\log(1) = 0$
$N\,[0,1]$	$\log(0.5) = -1.00$
myszy **[2,3]**	$\log(1) = 0$

Znajdujemy reguły, które mają symbol *myszy* po prawej stronie:

— $N \rightarrow$ *myszy* .

Znów istnieje tylko jedna taka reguła. Konstruujemy nową krawędź $N\,[2,3]$, dodajemy ją do agendy i obliczamy dla niej wartość oszacowania:

$$\beta(\,N\,[2,3]) = \log P(\,N \rightarrow \textit{myszy}\,) = \log(0.25) = -2\ .$$

Zawartość agendy po tym kroku:

agenda			
krawędź e	$\beta(e)$	$a(e)$	$(\beta + a)(e)$
jedzą $[1,2]$	$\log(1) = 0$	$\log(0.0024) = -5.38$	-5.38
$\boldsymbol{N}$ **[2,3]**	$\log(0.25) = -2.00$	$\log(0.112) = -3.16$	$\mathbf{-5.16}$

Wyjmujemy krawędź $N\,[2,3]$, ponieważ ma największą wagę, i wstawiamy ją do tablicy:

tablica	
krawędź e	$\beta(e)$
koty $[0,1]$	$\log(1) = 0$
$N\,[0,1]$	$\log(0.5) = -1.00$
myszy $[2,3]$	$\log(1) = 0$
$\boldsymbol{N}$ **[2,3]**	$\log(0.25) = -2.00$

Nie są tworzone żadne nowe krawędzie. Zawartość agendy:

agenda			
krawędź e	$\beta(e)$	$a(e)$	$(\beta + a)(e)$
jedzą $[1,2]$	$\log(1) = 0$	$\log(0.0024) = -5.38$	**-5.38**

Wyjmujemy z agendy jedyną krawędź i wstawiamy ją do tablicy:

tablica	
krawędź e	$\beta(e)$
koty $[0,1]$	$\log(1) = 0$
$N\,[0,1]$	$\log(0.5) = -1.00$
jedzą **[1,2]**	$\log(1) = 0$
myszy $[2,3]$	$\log(1) = 0$
$N\,[2,3]$	$\log(0.25) = -2.00$

Istnieją dwie reguły z symbolem *jedzą* po prawej stronie:

— $V \rightarrow$ *jedzą*,

— $VP \rightarrow$ *jedzą*.

Każda z nich może być zastosowana do krawędzi *jedzą* $[1,2]$. Konstruujemy nowe krawędzie: $V\,[1,2]$ i $VP\,[1,2]$. Dodajemy je do agendy i obliczamy wartości oszacowań:

$$\beta(\,V\,[1,2]) = \log P(\,V \rightarrow \textit{jedzą}\,) = \log(0.3) = -1.74\,,$$
$$\beta(\,VP\,[1,2]) = \log P(\,VP \rightarrow \textit{jedzą}\,) = \log(0.35) = -1.51\,.$$

Wartość oszacowania $a(\,VP\,[1,2])$ wynosi $-\infty$, ponieważ nie istnieje żadne drzewo składniowe łańcucha $a\,VP\,b$ z symbolu początkowego S (gdzie $a, b \in T$).

agenda			
krawędź e	$\beta(e)$	$a(e)$	$(\beta + a)(e)$
$\boldsymbol{V}$ **[1,2]**	$\log(0.3) = -1.74$	$\log(0.08) = -3.64$	$\mathbf{-5.38}$
$\boldsymbol{VP}$ **[1,2]**	$\log(0.35) = -1.51$	$\log(0) = -\infty$	$-\infty$

Wyjmujemy z agendy krawędź, dla której wartość $\beta + a$ jest największa, czyli tym razem jest to jedyna krawędź, dla której wartość ta jest różna od $-\infty$: krawędź $V\,[1,2]$. Wstawiamy ją do tablicy:

tablica	
krawędź e	$\beta(e)$
koty $[0,1]$	$\log(1) = 0$
$N\,[0,1]$	$\log(0.5) = -1.00$
jedzą $[1,2]$	$\log(1) = 0$
$\boldsymbol{V}$ **[1,2]**	$\log(0.3) = -1.74$
myszy $[2,3]$	$\log(1) = 0$
$N\,[2,3]$	$\log(0.25) = -2.00$

Istnieją dwie reguły, które można zastosować do krawędzi znajdujących się w agendzie:

(a) $VP \rightarrow N\ V$,
(b) $VP \rightarrow V\ N$.

Możemy zastosować każdą z nich, ponieważ w agendzie znajduje się zarówno krawędź N po lewej stronie krawędzi V (reguła (a)), jak i krawędź N po prawej stronie krawędzi V (reguła (b)). Tworzymy nowe krawędzie: $VP\,[0,2]$ i $VP\,[1,3]$. Obliczamy dla nich wartość β:

$$\begin{aligned}
\beta(\,VP\,[0,2]) &= \log\left(P(\,VP \rightarrow N\ V\,) \cdot P(\,N \rightarrow \textit{koty}\,) \cdot P(\,V \rightarrow \textit{jedzą}\,)\right) = \\
&= \log P(\,VP \rightarrow N\ V\,) + \log P(\,N \rightarrow \textit{koty}\,) + \log P(\,V \rightarrow \textit{jedzą}\,) = \\
&= \log 0.1 + \beta(\,N\,[0,1]) + \beta(\,V\,[1,2]) = \\
&= -3.32 - 1.00 - 1.74 = -6.06\ , \\
\beta(\,VP\,[1,3]) &= \log\left(P(\,VP \rightarrow V\ N\,) \cdot P(\,V \rightarrow \textit{jedzą}\,) \cdot P(\,N \rightarrow \textit{myszy}\,)\right) = \\
&= \log P(\,VP \rightarrow V\ N\,) + \log P(\,V \rightarrow \textit{jedzą}\,) + \log P(\,N \rightarrow \textit{myszy}\,) = \\
&= \log 0.4 + \beta(\,V\,[1,2]) + \beta(\,N\,[2,3]) = \\
&= -1.32 - 1.74 - 2.00 = -5.06\ .
\end{aligned}$$

Wstawiamy je do agendy:

agenda			
krawędź e	$\beta(e)$	$a(e)$	$(\beta + a)(e)$
$VP\,[1,2]$	$\log(0.35) = -1.51$	$\log(0) = -\infty$	$-\infty$
$\boldsymbol{VP}$ **[0,2]**	$\log(0.015) = -6.06$	$\log(0.1) = -3.32$	-9.38
$\boldsymbol{VP}$ **[1,3]**	$\log(0.03) = -5.06$	$\log(0.4) = -1.32$	$\mathbf{-6.38}$

Wybieramy z agendy krawędź o największej wadze i wstawiamy ją do tablicy:

tablica	
krawędź e	$\beta(e)$
koty $[0,1]$	$\log(1) = 0$
$N\,[0,1]$	$\log(0.5) = -1.00$
jedzą $[1,2]$	$\log(1) = 0$
$V\,[1,2]$	$\log(0.3) = -1.74$
$\boldsymbol{VP}$ **[1,3]**	$\log(0.03) = -5.06$
myszy $[2,3]$	$\log(1) = 0$
$N\,[2,3]$	$\log(0.25) = -2.00$

Istnieje tylko jedna reguła, którą można zastosować do bieżącej krawędzi i krawędzi znajdujących się w agendzie. Tą regułą jest $S \rightarrow N\ VP$. Zastosowanie tej reguły powoduje utworzenie krawędzi $S\,[0,3]$ zawierającej symbol początkowy gramatyki, więc można zakończyć działanie algorytmu. Znaleziona wartość

$$\begin{aligned}\beta(\,S\,[0,3]) &= \log P(\,S \rightarrow N\ VP\,) + \beta(\,N\,[0,1]) + \beta(\,VP\,[1,3]) = \\ &= -0.32 - 1.00 - 5.06 = -6.38\,.\end{aligned}$$

jest logarytmem prawdopodobieństwa Viterbiego łańcucha *koty jedzą myszy* (por. wynik uzyskany algorytmem CYK w przykładzie 5.1):

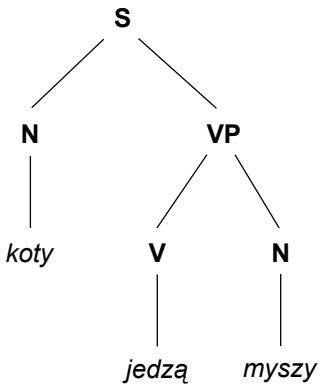
$$\log 0.012 = -6.38.$$

Ostateczna zawartość tablicy po zakończeniu działania algorytmu A*:

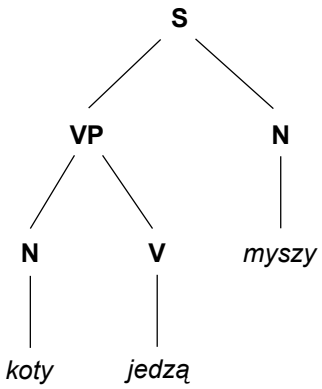
tablica	
krawędź e	$\beta(e)$
koty $[0,1]$	$\log(1) = 0$
$N\,[0,1]$	$\log(0.5) = -1.00$
$\boldsymbol{S}$ **[0,3]**	$\log(0.012) = -6.38$
jedzą $[1,2]$	$\log(1) = 0$
$V\,[1,2]$	$\log(0.3) = -1.74$
$VP\,[1,3]$	$\log(0.03) = -5.06$
myszy $[2,3]$	$\log(1) = 0$
$N\,[2,3]$	$\log(0.25) = -2.00$

Przeglądając krawędzie zawarte w tablicy i reguły, które zostały użyte do ich utworzenia, możemy odtworzyć drzewo Viterbiego łańcucha *koty jedzą myszy* (rys. 5.3). Nie różni się ono niczym od drzewa uzyskanego za pomocą algorytmu CYK.

Widać, że część krawędzi tworzących alternatywne drzewo wyprowadzenia rozważanego łańcucha (rys. 5.4) nie była przetwarzana przez algorytm A*, czego nie można powiedzieć o algorytmie CYK.



Rysunek 5.3. Drzewo Viterbiego łańcucha *koty jedzą myszy* uzyskane za pomocą algorytmu A*.



Rysunek 5.4. Drzewo wyprowadzenia łańcucha *koty jedzą myszy* , które nie jest drzewem Viterbiego.

5.3. *Coarse-to-fine*

Coarse-to-fine jest grupą algorytmów parsingu probabilistycznych gramatyk bezkontekstowych. Ogólnie rzecz ujmując, metody *coarse-to-fine* polegają na tym, że do parsowania używanych jest dwie (lub więcej) gramatyk: najpierw prostsza gramatyka jest używana po to, żeby ograniczyć przestrzeń poszukiwań poprzez odrzucenie najmniej obiecujących interpretacji; następnie bardziej szczegółowa gramatyka używana jest do parsingu właściwego.

Metody *coarse-to-fine* były przede wszystkim rozwijane przez Charniaka. Jego *Maximum-Entropy-Inspired Parser* [7] działa dwufazowo: w pierwszym przebiegu generowane są drzewa składniowe danego zdania, w drugim przebiegu są one weryfikowane przy użyciu skonstruowanego modelu probabilistycznego. Ten pomysł był następnie udoskonalany przez Charniaka. W [9] opisuje on parser, który w pierwszym przebiegu zwraca 50 najlepszych drzew składniowych dla danego zdania. W kolejnym przebiegu algorytmu z tych 50 drzew wybierane jest najlepsze za pomocą bardziej szczegółowej gramatyki. Z kolei [8] zawiera opis wielopoziomowego parsera typu *coarse-to-fine*, w którym korzysta się z szeregu coraz dokładniejszych probabilistycznych gramatyk bezkontekstowych. Las drzew składniowych uzyskanych w procesie parsingu każdej z nich służy do ograniczenia przestrzeni poszukiwań dla kolejnych przebiegów.

Zaletą podejścia *coarse to fine* jest zwiększenie wydajności parsingu przy jednoczesnym zachowaniu jego jakości. Parsing z wykorzystaniem prostszej gramatyki przebiega szybciej, natomiast gramatyka dokładniejsza umożliwia bardziej precyzyjny wybór właściwego drzewa składniowego.

5.4. *Cube pruning* i pochodne

Metody parsingu typu *cube pruning* zostały opracowane głównie z myślą o tłumaczeniu automatycznym wykorzystującym gramatyki synchroniczne. Gramatyki synchroniczne to gramatyki, które opisują dwa języki jednocześnie — dzięki temu można opisywać zależności, jakie występują przy tłumaczeniu poszczególnych fraz.

Algorytm *cube pruning* został przez Chianga [10] jako jedna z metod stosowania modelu języka do wyboru najlepszego drzewa składniowego w fazie dekodowania w systemie tłumaczenia automatycznego wykorzystującym synchroniczną gramatykę bezkontekstową. Chiang zauważył mianowicie, że możliwe interpretacje dla danego fragmentu zdania tworzą trójwymiarową kostkę, a ich obliczone wagi są do siebie proporcjonalne. Aby zatem odrzucić te interpretacje, które są najmniej obiecujące (mają najmniejszą wagę), nie

trzeba obliczać wag dla wszystkich możliwości, lecz wystarczy obliczyć wagi tylko w tym narożniku „kostki", w którym są one najmniejsze, a pozostałą część „kostki" odrzucić (stąd nazwa). Metoda *cube pruning* nie ogranicza się do tłumaczenia automatycznego, lecz może być wykorzystana w parsingu typu *coarse-to-fine* w fazie ograniczania przestrzeni poszukiwań.

Algorytm *coarse-to-fine* doczekał się wielu odmian, wśród których można wymienić: *cube growing* [22], *early pruning* [42], *closing spans* [46] czy *pervasive laziness* [41].

Hopkins i Langmead [20] pokazali, że algorytm *cube pruning* jest równoważny użyciu na odpowiedniej przestrzeni poszukiwań algorytmu A* z odpowiednią heurystyką.

5.5. Inne podejścia

Tsuruoka i Tsujii [65] zauważyli, że parser A* zaproponowany przez Kleina i Manninga implementuje agendę za pomocą kolejki priorytetowej, dlatego w praktyce nie zawsze jest szybki. Zamiast tego proponują iteracyjne rozszerzenie parsera CYK. W każdym kroku algorytmu ustalana jest pewna wartość progowa. Elementy, dla których funkcja oceny jest mniejsza od wartości progowej, są odrzucane. W kolejnych krokach wartości progowe są coraz mniejsze, a parsing jest powtarzany aż do skutku — póki nie zostanie znalezione drzewo Viterbiego dla wejściowego łańcucha.

Schmid [48] z kolei proponuje zastosować w parsingu wektory bitowe — struktury danych przechowujące tablice wartości logicznych i umożliwiające efektywne operacje na tych wartościach. Uważa, że dzięki temu podstawowe operacje parsingu mogą być wykonywane równolegle. To podejście wydaje się obiecujące zwłaszcza wtedy, jeżeli trzeba znaleźć wszystkie możliwe drzewa składniowe (wraz z ich prawdopodobieństwami) dla danego łańcucha wejściowego, a nie tylko najbardziej prawdopodobne z nich.

Rozdział 6

Autorska implementacja algorytmu parsowania dla języków o szyku swobodnym

Gramatyki bezkontekstowe, opisane w podrozdziale 2.2, są potężnym narzędziem opisu zarówno języków naturalnych, jak i sztucznych. Ich prostota, a jednocześnie uniwersalność sprawiają, że znajdują zastosowanie w przetwarzaniu języka naturalnego. Idea gramatyk bezkontekstowych została rozwinięta przez opracowanie probabilistycznych gramatyk bezkontekstowych, opisanych w podrozdziale 2.4. Probabilistyczne gramatyki bezkontekstowe korzystają z dobrodziejstw rachunku prawdopodobieństwa, dzięki czemu można między innymi tworzyć modele języka czy określać najlepszą interpretację zdania (wykorzystując pojęcie drzewa Viterbiego).

Probabilistyczne gramatyki bezkontekstowe są dobrze opisane i opracowano dla nich szereg algorytmów parsingu. Ich opisowi poświęcony jest rozdział 5. Jednak w niektórych zastosowaniach wygodniej jest używać gramatyk, w których zamiast miar probabilistycznych używa się wag nieprobabilistycznych (lub innego rodzaju nagród/kar). Gramatyki z wagami są szczególnie popularne w zastosowaniach związanych z przetwarzaniem języka naturalnego. Przykładem gramatyki z wagami jest WCDG (*weighted constraint dependency grammar*, 'ograniczone gramatyki zależnościowe z wagami') — formalizm opracowany przez Heineckego [18] i Schrödera [49]. W wagi można również wyposażyć przedstawione w sekcji 3.4.2 gramatyki binarne generujące drzewa. Gramatyki z wagami nie są tak dobrze zbadane jak gramatyki probabilistyczne, a ponadto są znacznie trudniejsze do parsowania. Algorytmy parsowania PCFG zazwyczaj korzystają z faktu, że wagi reguł są probabilistyczne, ponieważ odwołują się do takich pojęć jak prawdopodobieństwo zdania, prawdopodobieństwo łańcucha, prawdopodobieństwo wyprowadzenia czy prawdopodobieństwo drzewa składniowego, które trudno poprawnie zdefiniować dla gramatyk nieprobabilistycznych. Trudności rosną zwłaszcza wtedy, gdy wszystkie wagi są dodatnie, ponieważ wówczas nie mogą być traktowane jako logarytmy prawdopodobieństwa, a zatem nie mogą być w prosty sposób przekształcone na wagi probabilistyczne.

Wydaje się zatem uzasadnioną potrzeba szukania sposobów konwersji gramatyk nieprobabilistycznych (takich jak np. gramatyki z wagami) na grama-

tyki probabilistyczne. Podrozdział 6.2 przedstawia sposoby uzyskiwania wag probabilistycznych dla TgBG z wagami, co umożliwia konwersję z TgBG z wagami do PTgBG. W połączeniu z faktem równoważności między PCFG a PTgBG, pokazanym w podrozdziale 4.2, uzyskujemy, że przedstawione w rozdziale 5 metody parsingu PCFG mogą być z powodzeniem zastosowane w parserze wykorzystującym gramatyki TgBG z wagami (po uprzedniej konwersji do gramatyki probabilistycznej). Takim parserem jest parser Gobio, który jest opisany w podrozdziale 6.1.

Niniejszy rozdział przedstawia również proces adaptacji parsera Gobio do systemu PSI-Toolkit (podrozdziały 6.3 i 6.4). Zaadaptowany parser został poddany optymalizacji w celu poprawy wydajności parsingu. Ów proces optymalizacji jest opisany w podrozdziale 6.5.

6.1. Parser Gobio

Gobio jest głębokim parserem języka naturalnego, który w przeciwieństwie do parserów płytkich daje w wyniku działania pełną strukturę składniową zdania i oznacza role składniowe poszczególnych składników. Wynik parsingu przedstawiany jest w postaci drzewa składniowego, które reprezentuje zależności pomiędzy poszczególnymi składnikami. Gobio jest rodzajem parsera tablicowego, który wykorzystuje wariant algorytmu Cocke'a-Youngera-Kasamiego. Gobio korzysta z gramatyk TgBG z wagami.

Gobio pierwotnie opracowano jako głęboki parser dla języka niemieckiego w systemie tłumaczenia automatycznego Translatica [23]. Może być używany do parsowania różnych języków naturalnych. Dla parsera Gobio opracowano zestawy reguł dla różnych języków, w tym polskiego, niemieckiego, rosyjskiego i angielskiego. Reguły i wagi dla niektórych języków (polski, rosyjski) były tworzone ręcznie, natomiast dla innych języków (niemiecki, angielski) zostały automatycznie uzyskane na podstawie korpusów.

Ponieważ Gobio został przeznaczony dla konkretnego systemu tłumaczenia maszynowego, jest on mocno zintegrowany z innymi mechanizmami i zależny od formatów systemu Translatica, takich jak segmenter, lematyzator lub tagsety.

6.2. Uzyskiwanie wag dla parsera na podstawie korpusu[1]

6.2.1. Uczenie probabilistycznych gramatyk bezkontekstowych

Gramatykę bezkontekstową można łatwo przekształcić w PCFG przypisując jej regułom produkcji pewne prawdopodobieństwa. Należy jednak zwrócić uwagę, że nie każdy wybór wag probabilistycznych jest równie użyteczny w zastosowaniach. Dlatego ważną kwestią jest określenie prawdopodobieństw reguł w sposób odzwierciedlający cechy języka naturalnego modelowanego przez daną gramatykę bezkontekstową. Cel ten można osiągnąć poprzez proces nazywany uczeniem probabilistycznej gramatyki bezkontekstowej. Zakładamy, że wagi reguł będą właściwe, jeżeli będą maksymalizowały obliczone prawdopodobieństwo danych treningowych. W zależności od dostępnych danych istnieją różne sposoby wykonania tego zadania. Jeśli dysponujemy korpusem sparsowanym według danej gramatyki bezkontekstowej lub bankiem drzew, którego reprezentacja odpowiada rozważanej gramatyce, możemy określić prawdopodobieństwo reguły $A \to w$ przy użyciu następującego wzoru:

$$P(A \to w) = \frac{C(A \to w)}{\sum_{u \in (V \cup T)^*} C(A \to u)} \,, \tag{6.1}$$

gdzie $C(r)$ oznacza liczbę wystąpień reguły r w korpusie.

Gdy taki sparsowany korpus nie jest dostępny lub gdy formalizm użyty w banku drzew jest niezgodny z daną gramatyką, wówczas nie możemy bezpośrednio obliczyć wartości za pomocą podanego wyżej wzoru. Zamiast tego możemy wykorzystać iteracyjny algorytm opisany przez Manninga i Schützego [33] lub użyć algorytmów ewolucyjnych. Schröder [50] opisuje, jak zastosować algorytmy ewolucyjne dla WCDG, natomiast Tenerowicz [64] przedstawia wyniki zastosowania algorytmów ewolucyjnych w gramatykach w kontekście tłumaczenia maszynowego.

6.2.2. *British National Corpus* jako źródło pozyskiwania wag parsera

British National Corpus (BNC, 'Brytyjski Korpus Narodowy') jest jednojęzycznym synchronicznym ogólnym korpusem dla brytyjskiej odmiany języka angielskiego. Zawiera około stu milionów słów, zarówno z języka pisanego, jak i mówionego, co odpowiada około pięciu milionom zdań. BNC został opracowany w celu reprezentowania szerokiego zakresu współczesnego (XX w.) języka angielskiego. Próbki języka pisanego (90% całkowitej liczby

[1] Na podstawie [55].

próbek językowych w korpusie) pochodzą m.in. z gazet, czasopism, literatury naukowej, literatury pięknej, listów i szkolnych wypracowań. Próbki języka mówionego (pozostałe 10%) zawierają transkrypcje codziennych rozmów, audycji radiowych i spotkań biznesowych. BNC zawiera adnotacje dotyczące części mowy pochodzące z POS-taggera CLAWS.

BNC został stworzony w latach 1991—1994. Najnowsze wydanie — BNC XML Edition — zostało opublikowane w 2007 roku; właśnie to wydanie służyło jako podstawa dla naszych badań. Względem poprzednich wydań w BNC XML Edition poprawiono niektóre błędy, dodano informacje o częściach mowy i morfologii, a także zastosowano popularny format XML, co ułatwia pracę z korpusem. BNC jest dystrybuowany w postaci ponad czterech tysięcy plików, z których każdy zawiera pojedynczy dokument BNC.

6.2.3. Uzyskiwanie wag dla parsera Gobio

Parser Gobio używał gramatyki TgBG wyposażonej w ręcznie przypisane nieprobabilistyczne wagi. Chcieliśmy uzyskać na jej podstawie gramatykę probabilistyczną (PTgBG), a następnie dokonać ewaluacji parsera i porównać wyniki uzyskane za pomocą nowej gramatyki z wynikami uzyskiwanymi przy użyciu dotychczasowej gramatyki. Ocenie miały również podlegać wyniki tłumaczenia automatycznego dokonanego na podstawie parsingu. Chcieliśmy znaleźć odpowiedź na pytanie, jak zastąpienie ręcznie opracowanych wag automatycznie wygenerowanymi wagami probabilistycznymi wpłynie na jakość parsingu i tłumaczenia automatycznego.

Naszym celem nie było uzyskanie optymalnej strategii ujednoznaczniania możliwych interpretacji składniowych, ale raczej znalezienie takiej gramatyki probabilistycznej, która da wyniki parsowania możliwie zbliżone do wyników parsowania uzyskanych przy użyciu gramatyki wyjściowej.

Do uczenia gramatyki wykorzystano duży fragment korpusu BNC, zawierający około 270 000 zdań (ponad 5 mln słów). Jako gramatyki źródłowej użyliśmy własnoręcznie skonstruowanej gramatyki dla języka angielskiego parsera Gobio. Jest to zbinaryzowana gramatyka TgBG wyposażona w nieprobabilistyczne wagi (dodatnie jako nagrody lub ujemne jako kary) dołączone do niektórych reguł (większość reguł ma wagę o wartości domyślnej równej zero). Parser używa tej gramatyki do parsowania angielskich zdań, a wyniki są używane do wyboru najwłaściwszego rozbioru składniowego ze wszystkich możliwych. Następnie wybrane drzewo składniowe jest używane w procesie tłumaczenia.

Musieliśmy wiedzieć, które reguły istniejącej gramatyki nieprobabilistycznej (i jak często) są wykorzystywane przez parser podczas parsowania typo-

wych zdań. W tym celu wyposażyliśmy parser w możliwość wypisywania reguł stosowanych do budowy docelowego („najwłaściwszego") drzewa składniowego.

W pierwszym etapie eksperymentu wybrany fragment British National Corpus został sparsowany za pomocą parsera opartego na gramatyce nieprobabilistycznej. Wszystkie reguły użyte do budowy docelowego drzewa składniowego każdego zdania zostały zapisane i zachowane w bazie danych. Pozwoliło nam to skorzystać ze wzoru (6.1), aby utworzyć nowe wagi probabilistyczne.

Stworzyliśmy nową gramatykę probabilistyczną w następujący sposób: symbole końcowe i pomocnicze, a także zbiór reguł nowej gramatyki zostały zaczerpnięte bezpośrednio z istniejącej gramatyki nieprobabilistycznej. Dla reguł zawartych w bazie danych, czyli tych, które zostały wykorzystane do budowy drzew składniowych, przypisaliśmy prawdopodobieństwa obliczone ze wzoru (6.1). Niektóre reguły nie zostały użyte w ogóle, otrzymały one zerowe prawdopodobieństwa.

Pseudokod omówionego algorytmu podany jest w ramce zatytułowanej „Algorytm 2".

Algorytm 2: Uzyskiwanie wag probabilistycznych na podstawie korpusu

```
wejście: zbiór zdań corpus
dla każdej reguły left → right w gramatyce wykonaj:
    P(left → right) := 0;
dla każdego zdania sentence w korpusie corpus wykonaj:
    parse_tree := wynik parsingu zdania sentence;
    dla każdej reguły l → r w drzewie parse_tree wykonaj:
        jeżeli count(l) istnieje, to:
            count(l) := count(l) + 1;
        w przeciwnym wypadku:
            count(l) := 1;
        jeżeli count(l → r) istnieje, to:
            count(l → r) := count(l → r) + 1;
        w przeciwnym wypadku:
            count(l → r) := 1;
dla każdej reguły left → right w gramatyce wykonaj:
    jeżeli count(left → right) > 0, to:
        P(left → right) := count(left → right) / count(left);
```

Do testów wykorzystaliśmy system tłumaczenia automatycznego Transla-

tica. Można w nim używać zewnętrznego zestawu wag, więc byliśmy w stanie wykorzystać parser ze starą nieprobabilistyczną gramatyką, jak również nową gramatyką probabilistyczną. Przetłumaczyliśmy zestaw 108 przykładowych zdań angielskich na język polski przy użyciu obu gramatyk i porównaliśmy wyniki tłumaczenia w celu ustalenia, w jaki sposób zmiana wag reguł wpłynęła na jakość tłumaczenia.

W celu porównania jakości tłumaczenia przy użyciu starego parsera i nowego, w którym wykorzystano gramatykę probabilistyczną z wagami uzyskanymi z korpusu, przeprowadzono szereg testów. Zestaw przykładowych zdań w języku angielskim przetłumaczono na język polski za pomocą parserów opartych na następujących gramatykach:

— starej gramatyce z ręcznie przygotowanymi dodatnimi wagami,
— gramatyce z wagami probabilistycznymi uzyskanymi na podstawie korpusu BNC,
— gramatyce z losowymi ujemnymi wagami z zakresu między -10 a 0,
— gramatyce ze wszystkimi wagami równymi zeru.

Porównanie jakości wynikowego tłumaczenia z różnych parserów przedstawia tabela 6.1. Jakość tłumaczenia oceniana była przez człowieka.

Tabela 6.1. Porównanie wyników tłumaczenia za pomocą parserów wykorzystujących gramatyki z różnymi wagami (OLD — starą gramatykę nieprobabilistyczną, PROB — gramatykę z wagami probabilistycznymi uzyskanymi na podstawie BNC, RAND — gramatykę z losowymi wagami, ZERO — gramatykę z wszystkimi wagami równymi zeru)

	PROB	RAND	ZERO
Łącznie przetłumaczonych zdań	108	108	108
Zdań przetłumaczonych tak samo jak OLD	65	64	74
Zdań przetłumaczonych lepiej niż OLD	18	18	15
Zdań przetłumaczonych gorzej niż OLD	25	26	19
Jeśli PROB i RAND się różnią, lepsze jest...	7	4	

Wyniki ewaluacji pokazują, że uzyskane tłumaczenia różniły się nieznacznie. Około 60% zdań zostało przetłumaczonych dokładnie tak samo. Dalszych 17% zdań zostało przetłumaczonych przez nowy probabilistyczny parser lepiej niż przez dotychczasowy. 23% zdań zostało przetłumaczonych gorzej. Główną przyczyną tego ostatniego faktu było niewłaściwe tłumaczenie przyimka *of* (który można przetłumaczyć na język polski jako oznaczenie dopełniacza bądź jako przyimek *z*) przez nowy parser, co spowodowało 14 z 25 przypadków gorszego tłumaczenia. Może to wskazywać, że nowa gramatyka potrzebuje ręcznego dostrojenia, aby mogła konkurować z dotychczasową.

Najmniej różnic względem wyjściowego tłumaczenia wykazał parser oparty na gramatyce, w której wszystkie wagi były równe zeru. Głównym powodem jest zapewne fakt, że większość reguł starego parsera miała wagę właśnie zerową (a tylko niewielka część reguł miała niezerowe wagi). Z tego powodu do wyjściowej gramatyki była bardziej podobna gramatyka z wagami zero niż gramatyka probabilistyczna. Zgodnie z oczekiwaniami, zdania zostały lepiej przetłumaczone przy użyciu gramatyki probabilistycznej niż przy użyciu losowych wag. Zaskakującym wnioskiem natomiast jest fakt, że tłumaczenie z wykorzystaniem ręcznie przypisanych wag jest tylko nieznacznie lepsze niż to, w którym wagi w ogóle nie były wykorzystywane.

6.2.4. Wnioski

Może dziwić fakt, że parsing i tłumaczenie uzyskane za pomocą uzyskanej gramatyki probabilistycznej są gorsze niż parsing i tłumaczenie korzystające z wyjściowej gramatyki. Główną przyczyną wydaje się być fakt, że korpusem treningowym był niedoskonały bank drzew — uzyskany za pomocą parsera, który miał dopiero zostać „poprawiony" w trakcie eksperymentu. Gdyby jako korpusu treningowego użyć banku drzew poprawnie oznaczonego, przygotowanego lub choćby sprawdzonego ręcznie, to wyniki byłyby zapewne lepsze.

Nasz eksperyment pokazał, że przypisanie wag probabilistycznych do gramatyki bezkontekstowej (lub równoważnej) z ręcznie przypisanymi wagami można uzyskać bez większych trudności. Istniejące nieprobabilistyczne wagi reguł (nagrody lub kary) można zastąpić przez wagi probabilistyczne przy użyciu metod uczenia gramatyki. Otrzymana gramatyka probabilistyczna może być używana do celów parsingu z podobną skutecznością jak wyjściowa gramatyka — a przy odpowiednim doborze danych treningowych — zapewne wyższą.

6.3. PSI-Toolkit — system przetwarzania języka naturalnego[2]

PSI-Toolkit jest zestawem narzędzi do automatycznego przetwarzania języka naturalnego, stworzonym w Pracowni Systemów Informacyjnych (stąd nazwa) na Wydziale Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu w ramach grantu Ministerstwa Nauki i Szkolnictwa Wyższego (nr N N516 480540).

Na PSI-Toolkit składają się narzędzia (procesory) trojakiego rodzaju: narzędzia do odczytu (procesory odczytu, *readers*), narzędzia do zapisu (pro-

[2]Na podstawie [16, 24].

cesory zapisu, *writers*) i adnotatory (*annotators*). Przetwarzanie w ramach PSI-Toolkitu odbywa się w sposób łańcuchowy: najpierw procesor odczytu wczytuje dane z zewnętrznego źródła (którym może być na przykład plik XML albo zwykły plik tekstowy) do kraty, następnie na kracie operuje kolejno jeden lub więcej adnotatorów, na koniec wyniki są wyświetlane na ekranie bądź zapisywane do pliku w żądanym formacie za pomocą odpowiedniego procesora zapisu. Krata zawiera wszystkie informacje dotyczące aktualnego stanu przetwarzanych danych. Adnotatorami mogą być różne narzędzia przetwarzania języka, takie jak: segmentery, tokenizatory, lematyzatory, analizatory składniowe płytkie i głębokie, narzędzia do tłumaczenia automatycznego regułowego i statystycznego czy narzędzia do post-edycji. Są wśród nich narzędzia obsługujące poszczególne języki, m.in. polski, angielski, niemiecki, francuski, hiszpański i rosyjski, a także narzędzia niezależne językowo. Narzędzia te mogą być łączone ze sobą w celu wykorzystania ich do złożonych zadań. Na przykład łańcuch narzędzi segmenter — tokenizator — lematyzator — parser — translator realizuje zadanie tłumaczenia maszynowego.

System wspiera różne formaty wejścia i wyjścia, takie jak format Narodowego Korpusu Języka Polskiego, pliki HTML, RTF, LaTeX (za pomocą deformatera wzorowanego na Apertium) czy format UTT.

PSI-Toolkit jest projektem otwartoźródłowym (*open-source*) i udostępnionym publicznie na licencji GPL (rdzeń systemu oraz moduły stworzone w ramach grantu są udostępnione na licencji LGPL).

Zestaw narzędzi PSI-Toolkit posiada budowę modułową, która pozwala na tworzenie i dodawanie nowych procesorów. Przy wykonywaniu konkretnego zadania (np. lematyzacji) użytkownik ma możliwość wyboru narzędzia (np. może wybrać dowolny spośród dostępnych lematyzatorów), które uważa za najbardziej odpowiednie.

System może być obsługiwany dwojako: za pomocą narzędzi konsolowych (z wiersza poleceń) bądź przez przeglądarkę w postaci serwisu internetowego pod adresem `http://psi-toolkit.wmi.amu.edu.pl`. PSI-Toolkit jest też dostępny w postaci pakietów instalacyjnych dla popularnych dystrybucji Linuxa (Ubuntu, Debian, Linux Mint, Arch Linux).

6.4. Adaptacja parsera Gobio do systemu PSI-Toolkit[3]

PSI-Toolkit zawiera oprócz Gobio dwa inne parsery: Puddle i Link Grammar Parser. Puddle to płytki parser oparty na parserze ♠ (SPEJD) pierwotnie opracowanym w Instytucie Podstaw Informatyki PAN [39]. Link Gram-

[3]Na podstawie [56].

mar Parser to parser opracowany na Carnegie Mellon University [60]; korzysta z gramatyki *link grammar*, w której wynik parsingu ma postać powiązań między słowami w danym zdaniu. Parser ten posiada również opcję generowania wyjścia w formie prostego drzewa składniowego. Oba parsery są wielojęzyczne: mogą parsować różne języki, pod warunkiem, że są dostępne reguły parsingu dla tych języków. Nie opracowano jednak reguł Link Grammar Parsera dla języka polskiego. Żaden z tych parserów nie daje szczegółowego drzewa składniowego — są to parsery płytkie.

Chcieliśmy dołączyć parser Gobio do zestawu narzędzi PSI-Toolkitu, ponieważ oprócz parserów płytkich w zestawie narzędzi języka naturalnego przydatny jest parser głęboki. Potrzebowaliśmy również parsera Gobio w projekcie PSI-Toolkit, ponieważ planowaliśmy udostępnić w ramach tego zestawu narzędzi moduł tłumaczenia automatycznego systemu Translatica. Z tego powodu konieczna była adaptacja parsera Gobio do struktur PSI-Toolkitu.

6.4.1. Najważniejsze zadania i ich rozwiązania

Aby operacja dołączenia parsera Gobio do PSI-Toolkitu się powiodła, musieliśmy przezwyciężyć kilka trudności. Oba systemy zostały napisane głównie w języku C++, ale znacznie różnią się od siebie wewnętrzną budową. Używają różnych struktur danych i różnych tagsetów. Translatica jest systemem, którego głównym przeznaczeniem jest tłumaczenie maszynowe, dlatego wszystkie czynności przygotowawcze, takie jak podział tekstu na zdania, tokenizacja czy analiza morfologiczna odbywają się wewnątrz systemu i są realizowane przez narzędzia ściśle powiązane ze sobą i z systemem. Trzeba było oddzielić te narzędzia od właściwego parsera, aby móc wyodrębnić parser Gobio z Translatiki i zaadaptować go do modularnego systemu, jakim jest PSI-Toolkit.

PSI-krata a tablica Gobio

PSI-krata jest podstawową strukturą danych używaną w systemie PSI-Toolkit. Jest to rodzaj kraty słownej (*word lattice*) używanej w przetwarzaniu języka naturalnego [11]. Składa się z wierzchołków, które oznaczają miejsca w tekście wejściowym, oraz łączących je krawędzi, które odnoszą się do odpowiednich fragmentów tekstu. W systemie PSI-Toolkit przetwarzanie tekstu odbywa się w sposób następujący: na początku *reader* wczytuje tekst to kraty (w postaci krawędzi), a następnie kolejne adnotatory dodają swoje krawędzie do kraty. Na końcu *writer* wypisuje całą zawartość kraty lub odpowiednią jej część w formacie wskazanym przez użytkownika. Podstawową jednostką PSI-kraty jest pojedynczy symbol — wierzchołki kraty odpowiadają „punktom" między symbolami.

Parser Gobio wykorzystuje odmienną strukturę danych: tablicę (Gobio jest parserem tablicowym, *chart parser*). Jest to struktura skądinąd podobna do kraty, ale podstawowymi jednostkami są poszczególne słowa, a wierzchołki tablicy odpowiadają granicom słów.

W każdej krawędź PSI-kraty przechowywane są następujące informacje:

a) Wierzchołki, które są końcami krawędzi (początek krawędzi i jej koniec). Odpowiadają początkowi i końcowi fragmentu tekstu, do którego dana krawędź się odnosi.
b) Różnorodne adnotacje powiązane z krawędzią. Może to być kategoria gramatyczna krawędzi, tekst do niej przypisany czy adnotacje specyficzne dla poszczególnych procesorów, np. szczegóły analizy morfologicznej — rodzaj, przypadek, osoba, liczba, czas itp. Są przechowane w postaci łańcuchów znakowych.
c) Oznaczenia warstw. Wyrażają one pewne meta-informacje, takie jak np. typ krawędzi czy nazwę procesora, który dodał krawędź do kraty.
d) Podziały krawędzi. Wskazują, których krawędzi składowych użyto do konstrukcji danej krawędzi.
e) Wagi — wartości zmiennoprzecinkowe powiązane z krawędzią bądź poszczególnymi jej podziałami. Mogą oznaczać np. wagi reguł użytych podczas parsingu.

Krawędzie w tablicy używanej przez parser Gobio w systemie Translatica również przechowują podobne informacje, ale w inny sposób. Końce krawędzi odnoszą się do granic słów, a nie do granic znaków, jak w PSI-kracie. Nie występują oznaczenia warstw (nie ma takiej potrzeby, ponieważ Translatica nie jest systemem wielozadaniowym, który wykorzystuje różne procesory). Adnotacje są przechowywane w macierzach atrybutów i wartości (*AV-matrices*) zakodowane jako liczby całkowite; ich faktyczne wartości można odczytać korzystając ze specjalnych struktur nazywanych rejestratorami (*registrars*).

Aby parser Gobio mógł operować na PSI-kracie tak, jakby to była tablica z Translatiki, stworzona została specjalna struktura pośrednicząca (*PSI-lattice wrapper*). Zamienia ona na bieżąco wszystkie adnotacje przetwarzanych krawędzi, korzystając ze specjalnie stworzonych konwerterów (*AV-AI converter*). Wadą tego rozwiązania jest pewne spowolnienie działania parsera (część czasu działania parsera tracona jest na konwersję), jednak było ono konieczne, ponieważ parser Gobio korzysta z faktu, że adnotacje przechowywane są jako wartości liczbowe. Alternatywą byłoby przebudowanie całej struktury parsera.

Tagsety

W momencie pisania niniejszej pracy w PSI-Toolkicie dostępne są dwa analizatory morfologiczne. Jednym z nich jest Morfologik, lematyzator stworzony przez Miłkowskiego [35]. Morfologik korzysta z tagsetu opartego na tagsecie Narodowego Korpusu Języka Polskiego [38]. Drugim lematyzatorem jest Lamerlemma, prosty analizator morfologiczny opracowany specjalnie na potrzeby systemu PSI-Toolkit. Lamerlemma używa tego samego tagsetu, co Morfologik.

Gobio używa własnego tagsetu, który różni się znacząco od tagsetu Morfologika. Tagset Gobio posiada możliwość opisywania informacji leksykalnych, takich jak np. walencje czasowników. Ani Morfologik, ani Lamerlemma nie dostarczają takich informacji leksykalnych.

Aby umożliwić obsługę różnych tagsetów, potrzebny był konwerter pomiędzy tagami dostarczanymi przez Morfologik a tagami Gobio/Translatiki. W tym celu stworzyliśmy uniwersalny konwerter tagsetów, który konwertuje tagi według dostarczonych wcześniej reguł. Konwerter tagsetów jest niezależnym procesorem PSI-Toolkitu — może być łączony z różnymi innymi adnotatorami. Umożliwi to w przyszłości rozbudowę systemu i pracę z różnymi tagsetami.

Plik reguł konwertera tagsetów zawiera oprócz reguł konwersji specyfikację oznaczeń warstw, które wskazują które krawędzie kraty powinny zostać poddane konwersji. Rozróżniamy dwa rodzaje reguł: proste reguły podstawieniowe oraz reguły warunkowe. Reguły podstawieniowe pozwalają zastępować nazwy kategorii, atrybutów i wartości. Składnia reguł warunkowych umożliwia natomiast specyfikację bardziej zaawansowanych zależności między kategoriami, wartościami atrybutów, a nawet fragmentami tekstu odpowiadającymi danym krawędziom.

Poniższy listing przedstawia przykładowy plik reguł dla konwertera tagsetów:

```
# reguły konwersji tagsetu Morfologika do tagsetu Gobio

# oznaczenie warstwy źródłowej
@source morfologik-tagset

# oznaczenie warstwy docelowej
@target gobio-tagset

# oznaczenia warstw podlegające kopiowaniu
@tags   lexeme  form
```

```
# reguły podstawiania
@cat    adj     przym
@attr   number  L
@val    sg      1
@val    pl      2

# reguły warunkowe

# Jeżeli wyraz ma postać "niż"
# i został oznaczony kategorią "rzecz",
# to zduplikuj krawędź
# i jednej z kopii nadaj kategorię "rzecz",
# a drugiej "przyim".
CAT=rzecz, niż >> CAT=rzecz|przyim

# Jeżeli krawędź posiada kategorię "rzecz" oraz atrybuty:
# "L" o wartości "1" i "Rp" o wartości "mo",
# to ustaw wartość atrybutu "R" na "mż",
# zaś wartość atrybutu "R1" na równą wartości atrybutu "Rp".
CAT=rzecz, L=1, Rp=mo >> R=mż, R1=$Rp
```

Wiersze rozpoczynające się od znaku `#` są traktowane jak komentarze. Wiersze rozpoczynające się od `@source` i `@target` wskazują odpowiednio tagsety źródłowy i docelowy (a właściwie ich oznaczenia warstw). Wiersz rozpoczynający się od `@tags` wskazuje oznaczenia warstw, które powinny zostać zachowane przy kopiowaniu krawędzi. Pozostałe wiersze zaczynające się od znaku `@` zawierają reguły podstawieniowe. Słowa kluczowe `@cat`, `@attr` i `@val` oznaczają, że reguły odnoszą się odpowiednio do kategorii, atrybutów i wartości.

Niepuste wiersze, które zaczynają się od znaków innych niż `@` i `#` zawierają reguły warunkowe. Każda taka reguła składa się z części zawierającej warunki i części zawierającej polecenia. Blok warunków jest oddzielony od bloku poleceń przez `>>`.

Poszczególne warunki oddzielone są przecinkami. Jeżeli warunek ma postać pojedynczego wyrazu, to jest spełniony, jeżeli fragment tekstu odpowiadający badanej krawędzi jest tym wyrazem. Warunek może mieć też postać równości. Jest ona spełniona, jeżeli wartość atrybutu po lewej stronie jest równa wartości znajdującej się po prawej stronie. Jeżeli po lewej stronie równości znajduje się słowo kluczowe `CAT`, to równość jest spełniona, gdy kategoria krawędzi jest równa wartości po prawej stronie znaku równości.

Polecenia są oddzielone przecinkami. Wszystkie mają postać instrukcji przypisania. Symbol `$` po prawej instrukcji przypisania oznacza, że atrybutowi po lewej stronie należy przypisać wartość atrybutu poprzedzonego

symbolem `$`. Słowo kluczowe `CAT` odnosi się natomiast do kategorii, podobnie jak w bloku warunków. Czasami w poleceniach występuje kilka wartości oddzielonych pionową kreską (`|`). Oznacza to, że krawędź należy skopiować i w każdej kopii krawędzi zastosować inną wartość.

Ten system, choć nieskomplikowany, pozwala tworzyć zaawansowane reguły konwersji. Dzięki niemu można pomyślnie przekonwertować każdą adnotację wyrażoną przy pomocy tagsetu Morfologika na tagset Gobio/Translatiki.

Ponieważ lematyzatory dostępne w systemie PSI-Toolkit nie dostarczają informacji leksykalnych, potrzebny był prosty leksykon oraz słownik walencyjny dla systemu PSI-Toolkit. Tak powstało narzędzie *mapper* — uniwersalny leksykon wykonujący ogólne zadania mapujące. *Mapper* jest niezależnym adnotatorem PSI-Toolkitu, który potrafi generować walencje czasowników, jeżeli otrzyma na wejściu słownik walencyjny.

Innym uniwersalnym narzędziem, które zostało stworzone dla systemu PSI-Toolkit w związku z potrzebą dostarczenia parserowi Gobio informacji leksykalnych, jest *joiner*. *Joiner* tworzy produkt kartezjański określonych dwóch zbiorów krawędzi kraty. W tym przypadku umożliwia budowanie krawędzi, które zawierają zarówno formy pochodzące od lematyzatora, jak i walencje tworzone przez *mapper*. Te krawędzie stanowią właściwe dane wejściowe dla parsera Gobio.

Segmentacja, tokenizacja, lematyzacja

PSI-Toolkit ma strukturę modułową. Za procesy, które trzeba wykonać przed rozpoczęciem właściwego parsowania, czyli segmentację, tokenizację i lematyzację, odpowiedzialne są niezależne adnotatory. Jest to elastyczne podejście, ponieważ pozwala użytkownikowi wybrać spośród dostępnych narzędzi ten moduł, który wydaje mu się najodpowiedniejszy.

W systemie Translatica zastosowano odmienne podejście. Translatica jest systemem monolitycznym, którego głównym zadaniem jest tłumaczenie automatyczne. Segmenter, tokenizator i lematyzator Translatiki są integralnymi częściami systemu.

Podczas procesu adaptacji trzeba było oddzielić właściwy parser Gobio od tych dodatkowych narzędzi, żeby w systemie PSI-Toolkit można było używać innych segmenterów, tokenizatorów i lematyzatorów w połączeniu z parserem Gobio.

Wyniki parsingu

W parserze Gobio wynik parsowania tworzony jest w następujący sposób: najpierw zdanie wejściowe jest wstawiane do tablicy. Następnie moduł nazy-

wany kombinatorem próbuje połączyć krawędzie z tablicy zgodnie z regułami gramatyki tak, aby utworzyły nowe krawędzie, które następnie są wstawiane do tablicy. Proces ten powtarza się do wyczerpania krawędzi w agendzie. W ten sposób tworzony jest las, który zawiera wszystkie możliwe drzewa składniowe wyjściowego zdania. Na koniec moduł nazywany *chooserem* wybiera krawędzie, z których tworzy ostateczne drzewo składniowe. Nie jest ono wstawiane do tablicy, lecz przechowywane w oddzielnej strukturze danych.

Ponieważ ideą PSI-Toolkitu jest przechowywanie w kracie wszystkich etapów działania procesorów, zarówno pośrednich jak i końcowych, potrzebny był mechanizm, który wstawia otrzymane ostateczne drzewo składniowe do kraty. W tym celu stosowany jest mechanizm, który odnajduje w kracie krawędzie źródłowe, na których oparto ostateczne drzewo składniowe, i przyporządkowuje je krawędziom tego drzewa.

Tłumaczenie automatyczne

Wyniki parsingu Gobio mogą być wykorzystane do przeprowadzenia tłumaczenia maszynowego przez PSI-Toolkit. W systemie PSI-Toolkit dostępne są dwa narzędzia systemu służące do tłumaczenia: Bonsai i Transferer. Bonsai jest narzędziem do tłumaczenia wykorzystującym paradygmat składniowo-statystyczny. Transferer z kolei stanowi adaptację systemu transferu Translatiki, służy do przeprowadzania tłumaczenia regułowego. Każde z tych narzędzi można połączyć z parserem Gobio, aby stworzyć system tłumaczenia automatycznego. Dla Bonsai można wykorzystać Gobio do trenowania danych. Dla Transferera wynik parsingu Gobio jest źródłem informacji o strukturze składniowej tłumaczonego zdania.

6.4.2. Wnioski

Adaptacja pojedynczego narzędzia na moduł systemu przetwarzania języka naturalnego może wymagać nakładu pracy, ale przynosi pewne korzyści:

— Można rozbudować system i zwiększyć jego funkcjonalność.
— Sposoby użycia różnych narzędzi stają się zunifikowane.
— Narzędzia w ramach jednego modułowego systemu można ze sobą łączyć w łatwy sposób.
— System modułowy można lepiej skonfigurować pod potrzeby użytkownika.

6.5. Optymalizacja parsera Gobio w systemie PSI-Toolkit

Ceną adaptacji parsera Gobio do systemu PSI-Toolkit było obniżenie wydajności parsera. Można wymienić kilka przyczyn takiego stanu rzeczy. Po pierwsze, oryginalna wersja parsera Gobio została napisana do pracy na określonych strukturach danych (struktury systemu Translatica) i zoptymalizowana pod kątem tychże struktur. Postulat zbytniego nieingerowania w kod parsera podczas adaptacji spowodował, że mechanizmy adaptujące parser zostały obudowane dookoła niego, co było przyczyną narzutów czasowych podczas wykonywania parsingu. Podczas działania parsera w systemie PSI-Toolkit nieustannie była dokonywana konwersja pomiędzy strukturami danych Gobio a strukturami danych PSI-Toolkitu. Inną przyczyną spadku wydajności jest struktura kraty stosowana w PSI-Toolkicie — ponieważ jej jednostką podstawową są pojedyncze znaki, przetworzenie tekstu wymaga więcej operacji niż w przypadku tablicy, która operuje na całych wyrazach.

Z powyższych powodów po adaptacji parsera Gobio do systemu PSI-Toolkit konieczna była optymalizacja parsera w ramach nowego środowiska. Dokonaliśmy też optymalizacji wewnątrz samego systemu PSI-Toolkit.

6.5.1. Optymalizacja systemu PSI-Toolkit

Aby dowiedzieć się, które elementy systemu można poprawić, by zwiększyć wydajność parsera, system został poddany profilowaniu za pomocą narzędzia Gprof [13]. Najpierw poddaliśmy profilowaniu operację tokenizacji przy użyciu tokenizatora PSI-Toolkitu. Ponieważ tokenizacja jest nieskomplikowanym zadaniem, oczekiwaliśmy, że ujawnią się cechy samej kraty, które można zoptymalizować. Zadaniem testowym była tokenizacja 500 przykładowych zdań.

W wyniku profilowania zauważyliśmy przede wszystkim, że kluczowym miejscem jest operacja dodawania krawędzi do kraty, która wykonuje się wielokrotnie.

Pierwszym krokiem optymalizacji było zastąpienie dwukierunkowej tablicy asocjacyjnej `boost::bimap` przez jednokierunkową tablicę asocjacyjną `std::unordered_map`. Owa tablica asocjacyjna służy do przechowywania informacji dotyczących masek oznaczeń warstw (`LayerTagMask`), co służy usprawnieniu odczytywania informacji o krawędziach zawartych w kracie wychodzących z lub wchodzących do danego wierzchołka kraty. Okazało się, że można w tym miejscu zastąpić tablicę dwukierunkową przez tablicę jednokierunkową, ponieważ za odwzorowanie w drugą stronę może odpowiadać

odpowiednio skonstruowany wektor. Przyspieszyło to wykonywanie się tokenizacji o około 22%.

Kolejnym krokiem optymalizacji było usprawnienie operacji na maskach oznaczeń warstw. Poprzednio maski były konstruowane na bieżąco na żądanie za każdym razem, gdy były potrzebne. Okazało się, że zdecydowana większość wywołań konstruujących maski to wywołania kilku najpopularniejszych masek. Skonstruowanie tych najpopularniejszych masek zawczasu i odwoływanie się do tychże już skonstruowanych masek skróciło czas wykonywania się testowej tokenizacji o kolejne 20%.

Następny krok optymalizacji kraty to uproszczenie sposobu konstruowania masek złożonych z pojedynczego oznaczenia warstwy (*singleton masks*). Wydzielenie kodu tworzącego ten szczególny rodzaj masek do oddzielnej procedury spowodowało przyspieszenie działania testowego programu o dalszy 1%.

W dalszej kolejności został zoptymalizowany proces obsługi tagów płaszczyzn (*plane tags*). Tagi płaszczyzn to tagi, za pomocą których krata dzielona jest na niezależne płaszczyzny, co umożliwia m.in. proces tłumaczenia. Sprawdzanie, czy dodanie nowej krawędzi powinno spowodować wydzielenie nowej płaszczyzny, było poprzednio wykonywane przy użyciu ogólnych funkcji obliczających przekroje i sumy zbiorów znaczników warstw. Wydzielenie procedury dedykowanej do sprawdzania tagów płaszczyzn spowodowało poprawienie wydajności czasowej o kolejne 3%.

Ostatnim krokiem optymalizacji kraty było zastąpienie dynamicznej tablicy bitowej `boost::dynamic_bitset` przez tablicę bitową o stałym rozmiarze `std::bitset`. Owa tablica bitowa służy do przechowywania informacji o zbiorach znaczników warstw. Pierwotnie została zaprojektowana jako tablica bitowa dynamicznie zmieniająca rozmiar, aby móc przechowywać dowolną liczbę znaczników warstw. Okazało się, że w praktyce liczba używanych znaczników warstw jest ograniczona (rzadko kiedy przekracza 20), a zatem pierwotny rozmiar tej tablicy można ustawić od początku na odpowiednio dużą wartość (np. 64), co wyeliminuje konieczność aktualizacji jej rozmiaru. W tej sytuacji możliwe stało się użycie tablicy bitowej `std::bitset`, która jest szybsza niż `boost::dynamic_bitset`. Co więcej, koszt tej operacji, jakim jest zwiększenie złożoności pamięciowej, w praktyce jest pomijalny ze względu na niewielki rozmiar potrzebnych tablic. Przyspieszyło to wykonywanie się testowej tokenizacji o dalsze 19%.

Wszystkie te optymalizacje ogólnie spowodowały wzrost wydajności czasowej obserwowany na zestawie testowym łącznie o 52%, czyli o ponad połowę.

6.5.2. Optymalizacja parsera Gobio

W celu optymalizacji samego parsera Gobio również użyliśmy profilera Gprof. Tym razem jednak procedurą testową był parsing przykładowych 100 zdań języka polskiego.

Po przeprowadzeniu profilowania okazało się, że lwią część czasu parsingu (około 90%) zajmuje proces konwersji między obiektami `AnnotationItem` a `av_matrix`, które odpowiedzialne są sa przechowywanie informacji na temat krawędzi.

Proces optymalizacji kraty przedstawiony w poprzedniej sekcji (6.5.1) przyniósł poprawę wydajności czasowej o około 10%.

Klasa `AnnotationItem`, wykorzystywana intensywnie przez parser, oparta była o dynamiczną tablicę bitową `boost::dynamic_bitset`, podobnie jak zbiory znaczników warstw. Z tego powodu również tutaj dokonaliśmy zastąpienia dynamicznej tablicy bitowej jej odpowiednikiem o stałym rozmiarze, tablicą `std::bitset`. Przyczyniło się to do zmniejszenia czasu parsingu zestawu testowego o 9%.

W kolejnym kroku zredukowana została częstotliwość aktualizacji rozmiaru wektora stanowiącego główny element klasy `av_matrix`. Ustawianie odpowiednio dużego rozmiaru owego wektora już na etapie jego konstrukcji spowodowało przyspieszenie parsingu zestawu testowego o kolejne 9%.

Ostatni i najważniejszy krok stanowiło zredukowanie konwersji między `AnnotationItem` a `av_matrix`. Idealnym rozwiązaniem byłoby całkowite zastąpienie pochodzącej z systemu Translatica klasy `av_matrix` przez zaprojektowaną dla PSI-Toolkitu klasę `AnnotationItem`. Takie rozwiązanie nie wchodziło w grę, ponieważ mechanizm Gobio w zbyt dużym stopniu polega na pewnych cechach klasy `av_matrix`, takich jak np. przechowywanie informacji o kategoriach i atrybutach krawędzi jako liczb całkowitych. Poza tym, `av_matrix` jest nie tyle samodzielną klasą, co szablonem klas. Dlatego dokonaliśmy pozornego scalenia klas `AnnotationItem` i `av_matrix` w inny sposób. Najpierw została wyodrębniona klasa odpowiadająca specjalizacji szablonu `av_matrix<zvalue, int>` (jako najistotniejszej specjalizacji szablonu `av_matrix`). Następnie zastąpiła ona szablon `av_matrix` w tych miejscach, w których była użyta jej specjalizacja `av_matrix<zvalue, int>`. W dalszej kolejności `av_matrix<zvalue, int>` została uczyniona częścią składową klasy `AnnotationItem`. Zostały wystawione odpowiednie metody do komunikacji z obiektem `av_matrix<zvalue, int>` wewnątrz obiektu klasy `AnnotationItem`, dzięki czemu klasa `AnnotationItem` może symulować klasę `av_matrix<zvalue, int>`. Następnie zastąpiliśmy wystąpienia klasy `av_matrix<zvalue, int>` wystąpieniami klasy `AnnotationItem`. Do-

dana została również metoda „autokonwersji", która jest wywoływana wtedy, kiedy ma zostać dokonana konwersja między klasą `AnnotationItem` jako taką a klasą `AnnotationItem`, która symuluje klasę `av_matrix<zvalue, int>`. W ogólności konwersji nie można było pominąć, żeby nie zakłócić działania parsera, gdy operuje on na innych specjalizacjach szablonu `av_matrix`. Wykonanie całej tej operacji poskutkowało zwiększeniem szybkości obserwowanej podczas parsingu testowego zestawu zdań aż o 79% w porównaniu z poprzednim etapem optymalizacji.

Łącznie cały proces optymalizacji przyczynił się do ponad 6-krotnego (czyli dokładnie o 84%) zwiększenia szybkości parsera na zestawie testowym.

6.5.3. Ewaluacja — porównanie wydajności parsera Gobio na różnych etapach optymalizacji

Ewaluacja wydajności parsera Gobio i systemu PSI-Toolkit została przeprowadzona na komputerze z procesorem Intel Core i5-2450M o częstotliwości taktowania 2.5 GHz i pamięci RAM o rozmiarze 4 GB.

Do ewaluacji wydajności systemu PSI-Toolkit użyliśmy zestawu 500 zdań języka polskiego. Z tego zestawu wydzieliliśmy następnie 100 zdań, które posłużyły do ewaluacji parsera Gobio. Na każdym etapie optymalizacji wykonywany był zarówno test wydajności systemu PSI-Toolkit, jak i test wydajności parsera Gobio.

Do testowania wydajności systemu PSI-Toolkit został wybrany proces tokenizacji, ponieważ nie jest to skomplikowany proces i jako taki pozwala ocenić wydajność działania samego frameworku (a nie poszczególnych adnotatorów). Test polegał na kilkakrotnej tokenizacji testowego zestawu 500 zdań, a następnie obliczeniu średniego czasu wykonania tego zadania.

Ewaluację parsera Gobio przeprowadziliśmy uruchamiając kilkakrotnie na każdym etapie optymalizacji zadanie parsingu 100 testowych zdań za pomocą parsera Gobio wbudowanego w PSI-Toolkit, a następnie obliczając średni czas wykonania zadania.

Porównanie obliczonych średnich czasów tokenizacji i parsingu za pomocą systemu PSI-Toolkit i parsera Gobio przedstawia tabela 6.2.

Jak widać, dzięki optymalizacji udało się uzyskać ponad dwukrotne (dokładnie o 52%) zwiększenie wydajności systemu PSI-Toolkit oraz ponad sześciokrotny (dokładnie o 84%) wzrost wydajności samego parsera Gobio.

Dla porównania, parser Gobio w oryginalnym systemie tłumaczenia automatycznego Translatica parsuje ten sam zestaw 100 zdań w czasie 13 s.

Tabela 6.2. Porównanie czasów tokenizacji i parsingu przykładowych zestawu zdań na poszczególnych etapach optymalizacji systemu PSI-Toolkit i parsera Gobio

etap optymalizacji		czas [s]	
		tokenizacji 500 zdań	parsingu 100 zdań
0	przed optymalizacją	28.2	1782
1	zastąpienie `boost::bimap` przez `std::unordered_map`	21.9	1777
2	usprawnienie operacji na maskach	17.5	1771
3	uproszczenie konstrukcji pojedynczych masek	17.3	1722
4	optymalizacja obsługi płaszczyzn	16.7	1718
5	zastąpienie `boost::dynamic_bitset` w `LayerTagCollection` przez `std::bitset`	13.6	1596
6	zastąpienie `boost::dynamic_bitset` w `AnnotationItem` przez `std::bitset`	13.3	1446
7	ograniczenie aktualizacji rozmiaru wektora	13.3	1323
8	scalenie `av_matrix` z `AnnotationItem`	13.3	282

6.5.4. Wnioski

Niska wydajność parsera przed optymalizacją wynikała głównie ze sposobu, w jaki parser został wstawiony do zestawu narzędzi przetwarzania języka naturalnego PSI-Toolkit. Struktury danych, z których korzystały oba systemy, różniły się na tyle, że potrzebna była konwersja między nimi. Odbywanie się konwersji na każdym kroku algorytmu powodowało olbrzymi narzut obliczeniowy podczas parsingu. Zmiany w sposobie połączenia parsera z toolkitem, które ograniczyły nadmiarową konwersję, oraz szereg innych drobniejszych usprawnień poskutkowały ponad sześciokrotnym wzrostem wydajności parsera.

Z drugiej strony, czas parsingu przy użyciu parsera Gobio wewnątrz systemu tłumaczenia automatycznego Translatica jest ponad dwudziestokrotnie niższy. Częściowo przyczyną takiego stanu rzeczy jest narzut czasowy spowodowany koniecznością konwersji między strukturami danych Gobio i PSI-Toolkitu. Dodatkowo część mocy obliczeniowej zostaje poświęcona na wstawienie ostatecznych wyników parsingu do kraty (W Translatice ostateczne drzewo składniowe nie było stawiane do tablicy). Inną przyczyną takiej różnicy w wydajności jest fakt, że parser Gobio został pierwotnie napisany i zoptymalizowany dla systemu tłumaczenia maszynowego Translatica. Struktury danych używane przez system Translatica są wydajniejsze niż struktury danych używane przez system PSI-Toolkit, ponieważ są wyspecjalizowane

do jednego zadania: parsingu wykorzystywanego w tłumaczeniu automatycznym. Ceną za uniwersalność zestawu narzędzi PSI-Toolkit jest jego wydajność. Również parsing za pomocą parsera Gobio, mimo podjętych prób optymalizacji, przebiega sprawniej wewnątrz macierzystego systemu. Wygląda na to, że parser Gobio mógłby i w systemie PSI-Toolkit działać wydajniej, lecz wymagałoby to poważnej ingerencji w strukturę samego parsera, a być może i w strukturę całego systemu PSI-Toolkit.

Podstawowe założenie budowy systemu PSI-Toolkit było takie, że wszystkie operacje wykonywane w systemie mają być wykonywane bezpośrednio na kracie. Miało to na celu wprowadzenie porządku w strukturę systemu, a także miało to ułatwiać przyszłą rozbudowę toolkitu, m.in. tworzenie nowych narzędzi i dodawanie ich do systemu. Okazuje się jednak, że takie rozwiązanie posiada istotne wady. Założenie, że wszystkie operacje muszą być wykonywane z użyciem kraty, ogranicza możliwości optymalizacji poszczególnych narzędzi. Ponadto istniejące narzędzia, zoptymalizowane pod ich dotychczasowe środowisko, mogą nie dać się w łatwy sposób zaadaptować do toolkitu, zwłaszcza jeśli w sposób istotny są zależne od zastosowanych w nich struktur danych. Co więcej, jeżeli takie narzędzie zostało zoptymalizowane pod kątem dotychczasowego środowiska, to spadek wydajności po adaptacji narzędzia do nowego środowiska może być znaczący.

Być może zamiast konstruować zestaw narzędzi do przetwarzania języka naturalnego bazując na założeniu, że wszystkie operacje powinny być wykonywane na jednej strukturze danych, należałoby raczej zdecydować się na inną koncepcję współdziałania różnych procesorów: aby procesory komunikowały się między sobą, przekazując sobie jedynie wyniki pośrednie w ujednoliconej formie. Rozwiązanie tego rodzaju zastosowano m.in. w projekcie Apache UIMA [1], gdzie poszczególne narzędzia komunikują się za pomocą plików XML. Dzięki temu proces adaptacji narzędzia do toolkitu byłby jeszcze łatwiejszy, a ponadto poszczególne narzędzia mogłyby być lepiej zoptymalizowane, co przyczyniłoby się do zwiększenia ich wydajności.

Rozdział 7

Podsumowanie

W niniejszej pracy przedstawione zostały rozwiązania dotyczące analizy składniowej języków o szyku swobodnym. W szczególności zostały przedstawione formalizmy opisu takich języków oraz algorytmy parsingu. Autorski wkład stanowi ujednolicenie opisu tych formalizmów i przedstawienie ich w sposób spójny za pomocą wprowadzonego aparatu matematycznego.

Zaproponowałem autorską koncepcję probabilistycznej gramatyki binarnej generującej drzewa (PTgBG) — formalizmu służącego do opisu języków o szyku swobodnym i nieciągłościach syntaktycznych, który wykorzystuje rachunek prawdopodobieństwa. Pokazałem pewnego rodzaju równoważność gramatyk PTgBG i PCFG (probabilistycznych gramatyk bezkontekstowych). Gramatyki PTgBG i PCFG są równoważne w tym sensie, że generują te same klasy języków napisów. Ponadto pod pewnymi warunkami, dla dowolnego zdania, które może być wygenerowane przez te gramatyki, prawdopodobieństwa tego zdania obliczone przy użyciu PTgBG i przy użyciu PCFG są sobie równe.

Pokazałem, że można bez większych trudności uzyskać na podstawie korpusu wagi probabilistyczne dla gramatyki bezkontekstowej (lub równoważnej) z ręcznie przypisanymi wagami i że nie trzeba do tego celu dysponować sparsowanym korpusem bądź innym bankiem drzew.

Opisałem parser, który wykorzystuje formalizm PTgBG. Parser został zaadaptowany i włączony do zestawu narzędzi przetwarzania języka naturalnego, a następnie zoptymalizowany pod względem wydajności.

Adaptacja pojedynczego narzędzia i włączenie go do zestawu narzędzi do przetwarzania języka naturalnego może przynieść pewne korzyści. Ułatwia rozbudowę systemu i zwiększanie jego funkcjonalności. Unifikuje sposoby używania różnych narzędzi. Pozwala na łatwe łączenie różnych narzędzi w celu wykonania złożonych zadań. Ponadto system modułowy można lepiej skonfigurować pod potrzeby użytkownika.

Okazuje się jednak, że za te korzyści trzeba zapłacić pewną cenę. Konieczność dostosowania narzędzia do systemu przetwarzania języka naturalnego ogranicza możliwości optymalizacji tego narzędzia. Skutkuje to na ogół

spadkiem wydajności narzędzia względem jego pierwotnej (oddzielnej) implementacji. Także pewne założenia poczynione przy konstruowaniu systemu mają istotny wpływ na wydajność narzędzi, które wchodzą w jego skład. Na przykład postulat używania konkretnej struktury danych może ograniczyć możliwości optymalizacji i utrudnić dostosowanie narzędzia do systemu. Ponadto narzędzia już zoptymalizowane pod kątem pewnego środowiska po włączeniu w struktury docelowego systemu mogą stracić swoją właściwość wysokiej wydajności, ponieważ dokonane uprzednio optymalizacje mogą okazać się nieprzydatne w nowej strukturze danych.

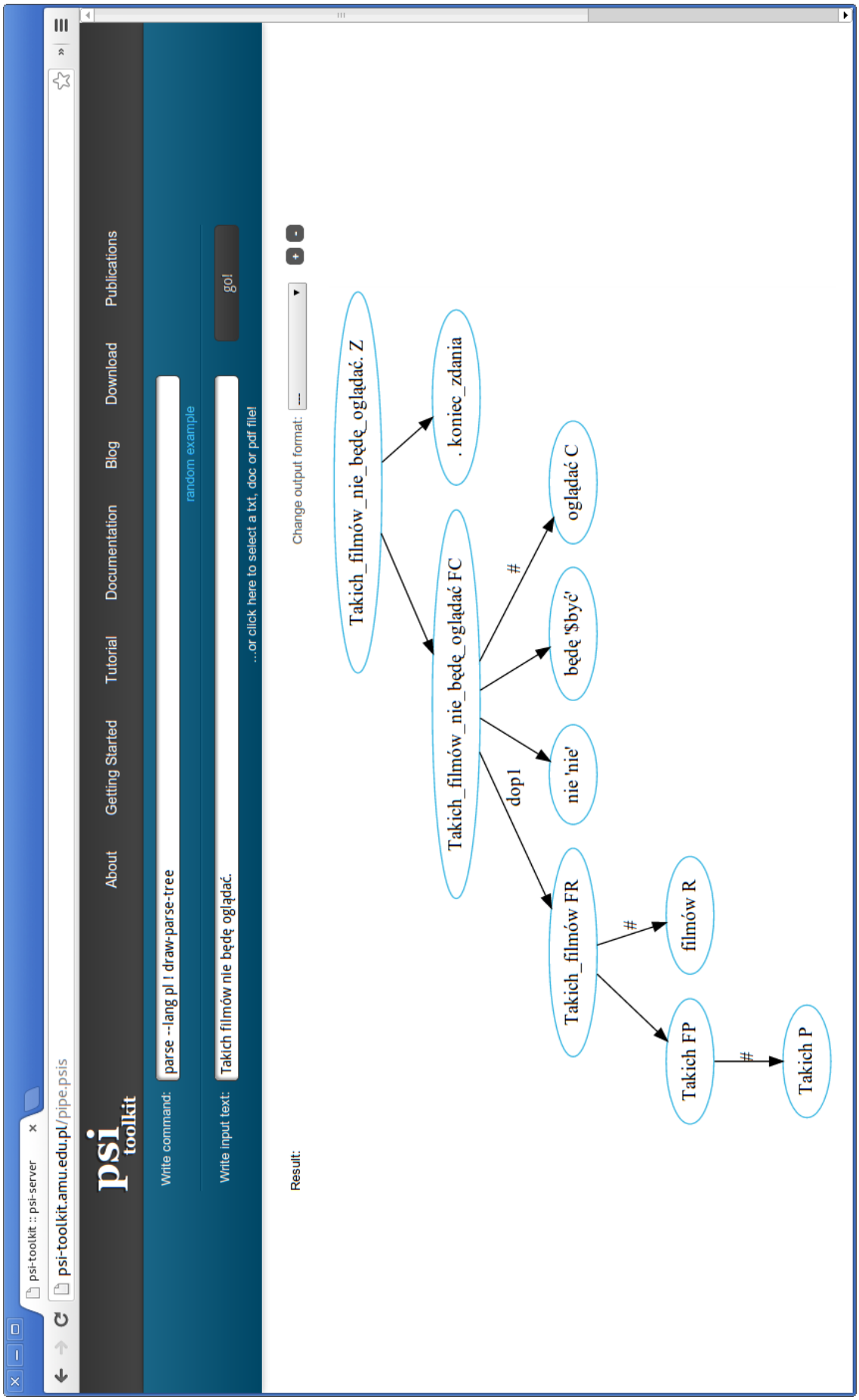
Idea podejścia modularnego do budowy systemów przetwarzania języka naturalnego ma swoje zalety, lecz przy konstruowaniu struktury takiego systemu, należy gruntownie przeanalizować potencjalne problemy wydajności. Być może zamiast nakładać ograniczenia na zawarte w systemie struktury danych i sposoby korzystania z nich należy raczej opracować dobre sposoby komunikacji między poszczególnymi narzędziami będącymi składnikami systemu.

Dodatek A

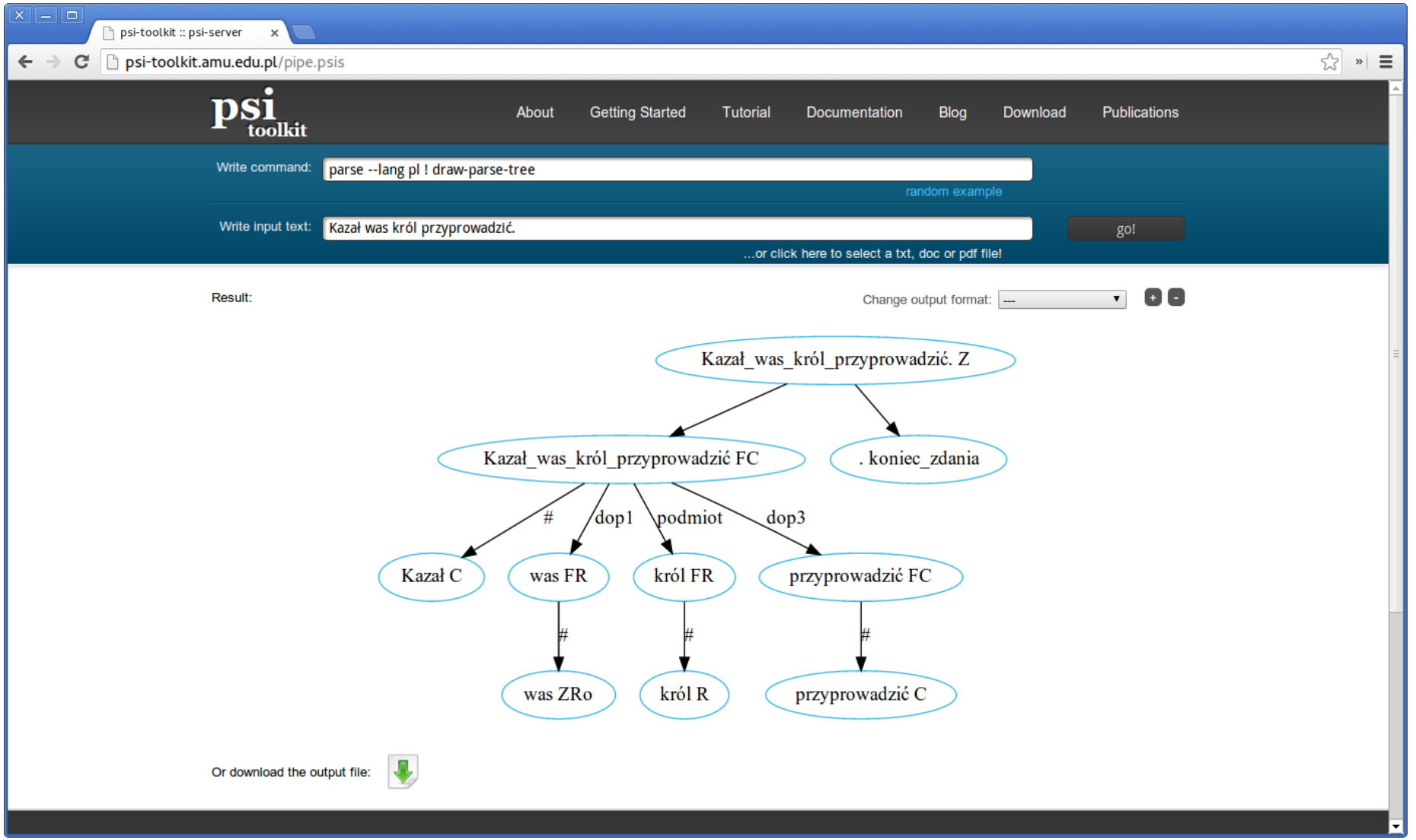
Przykłady użycia parsera Gobio w serwisie webowym PSI-Toolkit

Na kolejnych planszach widnieją zrzuty ekranu z wersji webowej systemu PSI-Toolkit, która dostępna jest pod adresem `http://psi-toolkit.amu.edu.pl`. Przedstawiają one wyniki parsingu za pomocą parsera Gobio przykładowych zdań zawierających nieciągłości:

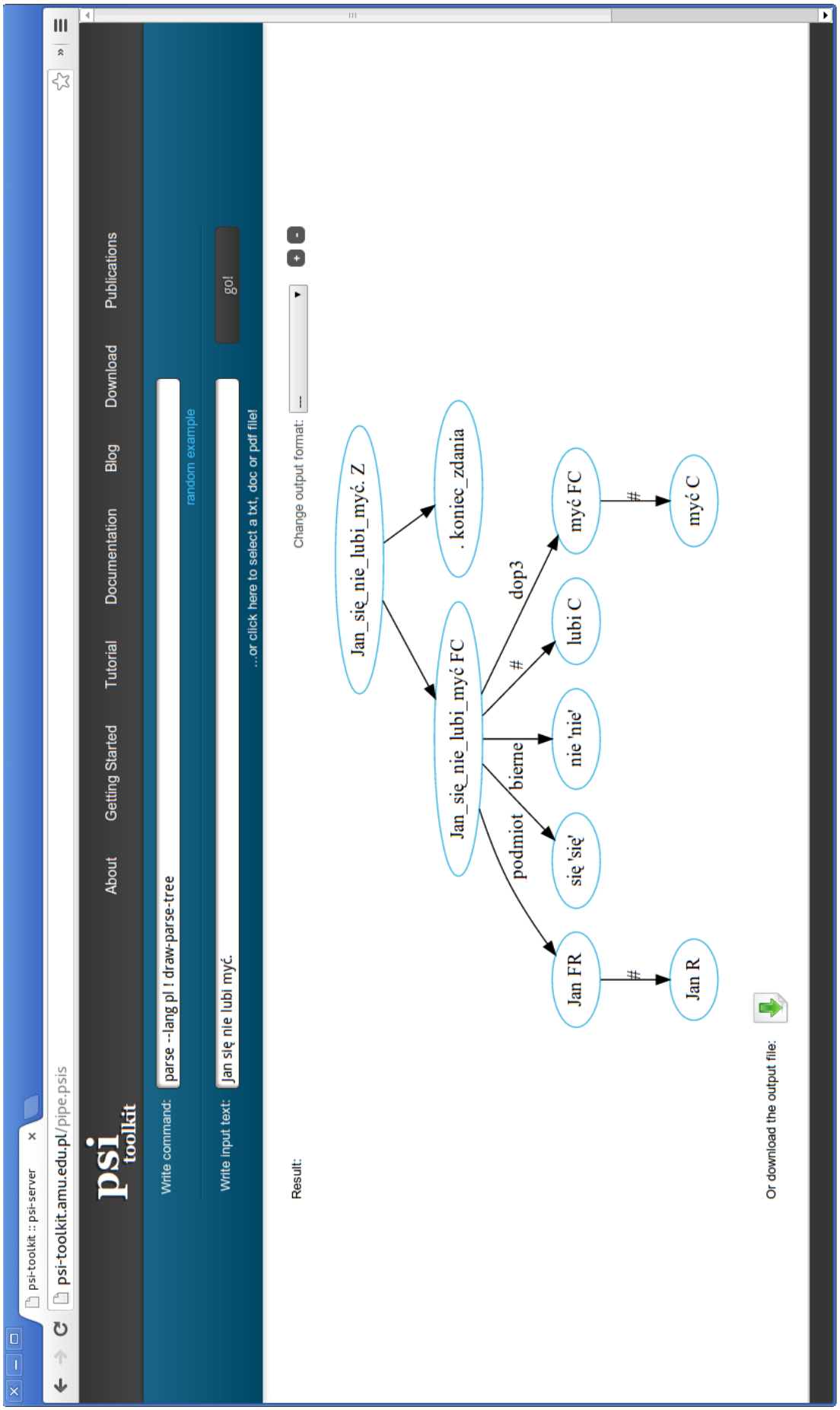
— zdanie I ze strony 22 — na rysunku A.1,
— zdanie J ze strony 22 — na rysunku A.2,
— zdanie K ze strony 22 — na rysunku A.3,



Rysunek A.1. Parsowanie zdania *Takich filmów nie będę oglądać.* za pomocą parsera Gobio w systemie PSI-Toolkit.



Rysunek A.2. Parsowanie zdania *Kazał Was król przyprowadzić.* za pomocą parsera Gobio w systemie PSI-Toolkit.



Rysunek A.3. Parsowanie zdania *Jan się nie lubi myć.* za pomocą parsera Gobio w systemie PSI-Toolkit.

Dodatek B

Obsługa parsera Gobio w konsolowej wersji systemu PSI-Toolkit

Zestaw narzędzi do przetwarzania języka naturalnego PSI-Toolkit można zainstalować na lokalnym komputerze z systemem Linux.

B.1. Instalacja systemu PSI-Toolkit

System PSI-Toolkit można zainstalować z pakietów bądź samodzielnie skompilować z kodu źródłowego.

B.1.1. Instalacja z pakietów

Ze strony `http://psi-toolkit.amu.edu.pl/download.html` można pobrać pakiety dla popularnych dystrybucji systemu Linux: Ubuntu, Linux Mint, Debian, Arch Linux. Aby zainstalować system, wystarczy odszukać paczkę dla posiadanego systemu, pobrać ją i uruchomić proces instalacji. Jeżeli podczas instalacji zostanie wyświetlony komunikat o błędzie, należy doinstalować odpowiednie wymagane pakiety zależne.

B.1.2. Instalacja z kodu źródłowego

Do instalacji z kodu źródłowego potrzebne są następujące biblioteki:
— system kontroli wersji Git,
— narzędzie CMake do automatycznego zarządzania procesem kompilacji,
— biblioteka Boost,
— biblioteka PCRE (*Perl Compatible Regular Expressions*) do obsługi wyrażeń regularnych.

Dodatkowo mogą być przydatne następujące biblioteki:
— biblioteka RE2 — dla szybszej obsługi wyrażeń regularnych,
— biblioteka SWIG — do integracji z językami skryptowymi (Perl, Python),
— biblioteka deweloperska języka Perl (pakiet `libperl-dev` dla Ubuntu) — do dowiązań do Perla,
— biblioteka GraphViz (`libgraphviz-dev`) — do wypisywania wyników w formie graficznej,

— biblioteki do obsługi plików PDF:
  — Poppler (`libpoppler-dev`),
  — Poppler-GLib (`libpoppler-glib-dev`),
  — GTK2 (`libgtk2.0-dev`),
— biblioteka LibMagic (`libmagic-dev`) — do rozpoznawania typów plików,
— Antiword (`antiword`) — do obsługi dokumentów programu MS Word,
— biblioteka DjVuLibre (`libdjvulibre-dev`) — do obsługi plików w formacie DjVu,
— biblioteka Link Grammar (`liblink-grammar4-dev`) — do obsługi parsera Link Grammar,
— narzędzia Bison i Flex (`bison`, `flex`) — konieczne do funkcjonowania parsera Gobio,
— biblioteka CMPH (`libcmph-dev`).

W przypadku systemu Ubuntu w wersji 11.10 lub nowszej, instalacji wymaganych pakietów dokonać można poleceniem:

```
sudo apt-get install g++ cmake make libpcre3-dev
    libboost-program-options-dev libboost-graph-dev
    libboost-filesystem-dev libboost-serialization-dev
    libboost-thread-dev libboost-system-dev libboost-test-dev
```

Instalacji pakietów opcjonalnych można natomiast dokonać poleceniem:

```
sudo apt-get install swig libperl-dev python python-all-dev
    openjdk-6-jdk libgraphviz-dev libpoppler-dev
    libpoppler-glib-dev libgtk2.0-dev libmagic-dev
    liblog4cpp4-dev libaspell-dev antiword libdjvulibre-dev
    liblink-grammar4-dev
```

Następnie należy pobrać kod źródłowy z repozytorium Git:

```
git clone ssh://git@mrt.wmi.amu.edu.pl:1978/psi-toolkit.git
```

Procedura instalacji została opisana w pliku *INSTALL.txt* i przebiega następująco:

```
cd psi-toolkit
mkdir -p build
cd build
cmake -D USE_JAVA=OFF ..
make
```

B.2. Korzystanie z systemu PSI-Toolkit w trybie konsolowym

PSI-Toolkit jest systemem, w którym poszczególne narzędzia mogą być ze sobą łączone w sposób przypominający przetwarzanie potokowe znane z systemów uniksowych. Potok procesorów zaczyna się od procesora odczytu (*reader*), następnie można użyć szeregu adnotatorów, a na końcu wypisywanie na ekran bądź zapisywanie do pliku realizowane jest przez procesor zapisu (*writer*). Poszczególne procesory oddzielone są od siebie znakiem `!`:

```
procesor_odczytu ! sekwencja_adnotatorów ! procesor_zapisu
```

Interfejs konsolowy systemu PSI-Toolkit uruchamiany jest poleceniem `psi-pipe`.

Dla przykładu, tokenizację zdania można uzyskać za pomocą potoku:

```
read-text ! tokenize ! write-simple
```

Pełne wywołanie tego potoku dla tokenizacji zdania *Koty jedzą myszy.* wygląda następująco:

```
echo "Koty jedzą myszy." | psi-pipe read-text ! tokenize !
    write-simple
```

Uzyskany wynik to:

```
Koty
jedzą
myszy
.
```

Wejście można również wczytać z pliku:

```
cat ../ścieżka/do/pliku.txt | psi-pipe read-text ! tokenize !
    write-simple
```

Aby dokonać analizy morfologicznej (lematyzacji) tego samego zdania, należy użyć potoku:

```
read-text ! tokenize ! lemmatize ! write-simple --tags lemma
```

Uzyskany wynik to:

```
Kot|kot|kota
jeść
mysz|myszy
```

(Opcja `--tags lemma` powoduje wypisanie informacji o lematach).

PSI-Toolkit wyposażony jest w funkcję automatycznego uzupełniania potoków, zatem zamiast powyższego potoku można użyć następującego:

```
lemmatize ! write-simple --tags lemma
```

Wówczas zadania konieczne do lematyzacji (takie jak wczytanie wejścia i tokenizacja) zostaną wykonane przy użyciu domyślnych procesorów. Procesor odczytu zostanie dopasowany do formatu wejścia, np. jeżeli jako wejście zostanie podany plik PDF, to do wczytania zostanie użyty procesor `pdf-reader`.

Wywołanie można nawet skrócić jeszcze bardziej, do pojedynczego napisu:

```
lemmatize
```

Wówczas do wypisania wyniku zostanie użyty domyślny format — format natywny PSI-Toolkitu (PSI-format).

Dzięki funckji autouzupełniania, w celu sparsowania zdania nie potrzeba wypisywać wszystkich etapów przygotowujących do parsingu (podział na zdania, tokenizacja, lematyzacja), lecz wystarczy napisać:

```
parse --lang pl
```

Opcja `--lang` służy wskazaniu języka — tu: polskiego (`pl`). Można ją pominąć, wówczas język może zostać określony automatycznie na podstawie tekstu źródłowego, pod warunkiem, że tekst jest dostatecznie długi, aby móc określić język, w jakim został napisany.

Funkcja autouzupełniania spowoduje, że zostaną użyte domyślne procesory podziału na zdania, tokenizacji i lematyzacji.

Ponieważ nie wskazano *explicite* procesora zapisu, zatem wyjście zostanie wypisane w domyślnym PSI-formacie. Aby uzyskać wyjście w innym formacie, należy użyć odpowiedniego procesora zapisu. Na przykład `bracketing-writer` pozwala na wypisanie wyjścia w wysoce konfigurowalnym formacie „zagnieżdżonych nawiasów"; za pomocą tego procesora można sformatować wyjście w taki sposób, by przypominało format XML. Wynik parsingu może również zostać zobrazowany graficznie za pomocą procesora `gv-writer`, który korzysta z biblioteki GraphViz.

Komenda `parse` nie wskazuje jednoznacznie, który procesor ma zostać użyty do parsingu. System sam decyduje, który spośród dostępnych parserów najlepiej nadaje się dla danego języka. Jeżeli potrzeba użyć konkretnie parsera Gobio, należy raczej użyć polecenia:

```
gobio --lang pl
```

Szczegółową dokumentację poszczególnych procesorów można znaleźć na stronie internetowej `http://psi-toolkit.amu.edu.pl`.

Bibliografia

[1] Apache UIMA. Strona internetowa: `http://uima.apache.org`, 2014.

[2] G. Edward Barton. On the complexity of ID/LP parsing. *Computational Linguistics*, 11:205–218, 1984.

[3] Adriane Boyd. Discontinuity revisited: an improved conversion to context-free representations. *Proceedings of the Linguistic Annotation Workshop*, strony 41–44, Prague, 2007. Association for Computational Linguistics.

[4] Harry Bunt. Parsing with discontinuous phrase structure grammar. *Current issues in parsing technology*, strony 49–63. Springer, 1991.

[5] Harry Bunt. Formal tools for describing and processing discontinuous constituency structure. *Natural Language Processing*, 6:63–84, 1996.

[6] Sharon A. Caraballo, Eugene Charniak. New figures of merit for best-first probabilistic chart parsing. *Computational Linguistics*, 24(2):275–298, 1998.

[7] Eugene Charniak. A maximum-entropy-inspired parser. *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, strony 132–139. Association for Computational Linguistics, 2000.

[8] Eugene Charniak, i in. Multilevel coarse-to-fine PCFG parsing. *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, strony 168–175. Association for Computational Linguistics, 2006.

[9] Eugene Charniak, Mark Johnson. Coarse-to-fine n-best parsing and maxent discriminative reranking. *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, strony 173–180. Association for Computational Linguistics, 2005.

[10] David Chiang. Hierarchical phrase-based translation. *computational linguistics*, 33(2):201–228, 2007.

[11] Christopher Dyer, Smaranda Muresan, Philip Resnik. Generalizing word lattice translation. Raport instytutowy, DTIC Document.

[12] Gerald Gazdar, Ewan Klein, Geoffrey K. Pullum, Ivan Sag. *Generalized Phrase Structure Grammar*. Basil Blackwell, Oxford, 1985.

[13] Susan L. Graham, Peter B. Kessler, Marshall K. Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, wolumen 17, strony 120–126. ACM, 1982.

[14] Filip Graliński. A simple CF formalism and free word order. Zygmunt Vetu-

lani, redaktor, *2nd Language & Technology Conference. Proceedings*, strony 172–176, Poznań, 2005. Wydawnictwo Poznańskie Sp. z o.o.

[15] Filip Graliński. *Formalizacja nieciągłości zdań przy zastosowaniu rozszerzonej gramatyki bezkontekstowej*. Praca doktorska, Adam Mickiewicz University, Faculty of Mathematics and Computer Science, Poznań, 2007. Promotor Zygmunt Vetulani.

[16] Filip Graliński, Krzysztof Jassem, Marcin Junczys-Dowmunt. PSI-Toolkit: Natural language processing pipeline. *Computational Linguistics — Applications*, 458:27–39, 2012.

[17] Robert Harper, Furio Honsell, Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[18] Johannes Heinecke, Jürgen Kunze, Wolfgang Menzel, Ingo Schröder. Eliminative parsing with graded constraints. *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics*, wolumen 1, strony 526–530. Association for Computational Linguistics, 1998.

[19] John Edward Hopcroft, Jeffrey David Ullman. *Wprowadzenie do teorii automatów, języków i obliczeń*. Wydawnictwo Naukowe PWN, Warszawa, 2003.

[20] Mark Hopkins, Greg Langmead. Cube pruning as heuristic search. *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, wolumen 1, strony 62–71. Association for Computational Linguistics, 2009.

[21] Yu-Yin Hsu. Comparing conversions of discontinuity in PCFG parsing. Markus Dickinson, Kaili Müürisep, Marco Passarotti, redaktorzy, *Proceedings of the 9th International Workshop on Treebanks and Linguistic Theories*, wolumen 9 serii *NEALT Proceedings Series*, strony 103–113, Tartu, 2010. Northern European Association for Language Technology.

[22] Liang Huang, David Chiang. Forest rescoring: Faster decoding with integrated language models. *Annual Meeting-Association For Computational Linguistics*, wolumen 45, strona 144, 2007.

[23] Krzysztof Jassem. *Przetwarzanie tekstów poslkich w systemie tłumaczenia automatycznego Translatica*. Wydawnictow Naukowe UAM, Poznań, 2006.

[24] Krzysztof Jassem. PSI-Toolkit — how to turn a linguist into a computational linguist. Petr Sojka, Ales Horák, Ivan Kopecek, Karel Pala, redaktorzy, *Proceedings of 15th International Conference on Text, Speech and Dialogue*, wolumen 7499 serii *Lecture Notes in Computer Science*, strony 215–222, 2012.

[25] Joanna Jędrzejowicz, Andrzej Szepietowski. *Języki, automaty, złożoność obliczeniowa*. Wydawnictwo Uniwersytetu Gdańskiego, Gdańsk, 2008.

[26] Mark Johnson. Parsing with discontinuous constituents. *Proceedings of the 23rd annual meeting on Association for Computational Linguistics*, ACL '85, strony 127–132, Stroudsburg, PA, USA, 1985. Association for Computational Linguistics.

[27] Daniel Jurafsky, James H. Martin. *Speech and Language Processing*. Prentice Hall, 1999.

[28] Ronald M. Kaplan, Joan Bresnan. Lexical-functional grammar: A formal system for grammatical representation. *Formal Issues in Lexical-Functional Grammar*, 1982.

[29] Natalia Kariaeva. *Radical discontinuity: syntax at the interface*. Praca doktorska, New Brunswick, NJ, USA, 2009.

[30] Dan Klein, Christopher D. Manning. A* parsing: Fast exact Viterbi parse selection. *In Proceedings of the Human Language Technology Conference and the North American Association for Computational Linguistics (HLT-NAACL)*, strony 119–126, 2003.

[31] Sandra Kübler. How do treebank annotation schemes influence parsing results? or how not to compare apples and oranges. *Proceedings of RANLP 2005*, Borovets, 2005.

[32] Roger Levy. *Probabilistic models of word order and syntactic discontinuity*. Praca doktorska, Stanford, CA, USA, 2005. AAI3186362.

[33] Christopher Manning, Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[34] Igor A. Mel'čuk, Alain Polguère. A formal lexicon in the meaning-text theory: (or how to do lexica with words). *Computational Linguistics*, 13:261–275, 1984.

[35] Marcin Miłkowski. Developing an open-source, rule-based proofreading tool. *Software: Practice and Experience*, 40(7):543–566, 2010.

[36] Vladimir Pericliev, Alexander Grigorov. Parsing a flexible word order language. *Proceedings of the 15th conference on Computational linguistics*, wolumen 1 serii *COLING '94*, strony 391–395, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics.

[37] Oliver Plaehn. Computing the most probable parse for a discontinuous phrase structure grammar, new developments in parsing technology, 2004.

[38] Adam Przepiórkowski. A comparison of two morphosyntactic tagsets of Polish. Violetta Koseska-Toszewa, Ludmila Dimitrova, Roman Roszko, redaktorzy, *Representing Semantics in Digital Lexicography: Proceedings of MONDILEX Fourth Open Workshop*, strony 138–144, Warszawa, 2009.

[39] Adam Przepiórkowski, Aleksander Buczyński. ♠: Shallow Parsing and Disambiguation Engine. Zygmunt Vetulani, redaktor, *Proceedings of the 3rd Language and Technology Conference*, strony 340–344, Poznań, 2007.

[40] Adam Przepiórkowski, Anna Kupść, Małgorzata Marciniak, Agnieszka Mykowiecka. *Formalny opis języka polskiego: Teoria i implementacja*. Akademicka Oficyna Wydawnicza EXIT, Warszawa, 2002.

[41] Michael Pust, Kevin Knight. Faster MT decoding through pervasive laziness. *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Lin-*

guistics, Companion Volume: Short Papers, strony 141–144. Association for Computational Linguistics, 2009.

[42] Chris Quirk, Robert Moore. Faster beam-search decoding for phrasal statistical machine translation. *Machine Translation Summit XI*, 2007.

[43] Aarne Ranta. Grammatical Framework: a type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189, 2004.

[44] Adwait Ratnaparkhi. Learning to parse natural language with maximum entropy models. *Machine Learning*, 34(1-3):151–175, 1999.

[45] Brian Roark. Probabilistic top-down parsing and language modeling. *Computational Linguistics*, 27(2):249–276, 2001.

[46] Brian Roark, Kristy Hollingshead. Classifying chart cells for quadratic complexity context-free inference. *Proceedings of the 22nd International Conference on Computational Linguistics*, wolumen 1, strony 745–751. Association for Computational Linguistics, 2008.

[47] Grzegorz Rosenberg, Arto Salomaa. *Handbook of Formal Languages, vol. 3: Beyond Words*. Springer, Berlin, 1997.

[48] Helmut Schmid. Efficient parsing of highly ambiguous context-free grammars with bit vectors. *Proc. of the 20th International Conference on Computational Linguistics (COLING)*, Geneva, 2004.

[49] Ingo Schröder. *Natural language parsing with graded constraints*. Praca doktorska, Hamburg University, Hamburg, 2002.

[50] Ingo Schröder, Horia F. Pop, Wolfgang Menzel, Kilian A. Foth. Learning grammar weights using genetic algorithms. *Proc. of Recent Advances in Natural Language Processing, RANLP-2001*, strony 235–239, 2001.

[51] Robert A. Sharman, Fred Jelinek, Robert Mercer. Generating a grammar for statistical training. *Proceedings of the workshop on Speech and Natural Language*, HLT '90, strony 267–274, Stroudsburg, PA, USA, 1990. Association for Computational Linguistics.

[52] Paweł Skórzewski. Efektywny parsing języka naturalnego przy użyciu gramatyk probabilistycznych. Praca magisterska, Wydział Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu, Poznań, 2010. Promotor: Krzysztof Jassem.

[53] Paweł Skórzewski. Effective natural language parsing with probabilistic grammars. Maria Ganzha, Marcin Paprzycki, redaktorzy, *Proceedings of the International Multiconference on Computer Scienceand Information Technology*, wolumen 5, strony 501–504, Wisła, 2010. Polskie Towarzystwo Informatyczne.

[54] Paweł Skórzewski. Gramatyki i automaty probabilistyczne. Praca magisterska, Wydział Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu, Poznań, 2010. Promotor: Wojciech Buszkowski.

[55] Paweł Skórzewski. Obtaining PCFG probabilities based on the corpus. Zygmunt Vetulani, redaktor, *Proceedings of the 5th Language and Technology Conference*, strony 411–414, Poznań, 2011. Fundacja UAM.

[56] Paweł Skórzewski. Gobio and PSI-Toolkit: Adapting a deep parser to an NLP toolkit. Zygmunt Vetulani, Hans Uszkoreit, redaktorzy, *Proceedings of the 6th Language and Technology Conference*, strony 523–526, Poznań, 2013. Fundacja UAM.

[57] Wojciech Skut, Brigitte Krenn, Thorsten Brants, Hans Uszkoreit. An annotation scheme for free word order languages. *Proceedings of the Fifth Conference on Applied Natural Language Processing ANLP-97*, Washington, DC, 1997.

[58] Paweł Skórzewski. An application of probabilistic grammars to efficient machine translation. *Investigationes Linguisticae*, 21:90–98, 2010.

[59] Adam Slaski. Opis fragmentu języka polskiego w formalizmie Grammatical Framework. Praca magisterska, Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, Warszawa, 2010. Promotor: Marcin Benke.

[60] Daniel D. Sleator, Davy Temperley. Parsing English with a link grammar. *arXiv preprint cmp-lg/9508004*, 1995.

[61] Andreas Stolcke. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201, 1995.

[62] Oliver Suhre. *Computational aspects of a grammar formalism for languages with freer word order*. Praca doktorska, Tübingen, 1999.

[63] Morten Heine B. Sørensen, Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, Amsterdam, 2006.

[64] Zbigniew Tenerowicz. Zastosowanie obliczeń ewolucyjnych w przetwarzaniu języka naturalnego. Praca magisterska, Wydział Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu, Poznań, 2010. Promotor: Krzysztof Jassem.

[65] Yoshimasa Tsuruoka, Jun'ichi Tsujii. Iterative CKY parsing for probabilistic context-free grammars. *Proceedings of the First International Joint Conference on Natural Language Processing*, IJCNLP '04, strony 52–60, Berlin, Heidelberg, 2005. Springer-Verlag.

[66] Zygmunt Vetulani. Free Order DCG (FROG) in application to fromal description of semi-free word order languages. *Sztuczna Inteligencja, Materiały V Konferencji Naukowej*, strony 59–72, Siedlce, Poland, 2002. Wyd. Akademii Podlaskiej.

[67] Robin James Wilson. *Wprowadzenie do teorii grafów*. Wydawnictwo Naukowe PWN, Warszawa, 1998.

Spis rysunków

Spis tabel