Uniwersytet im. Adama Mickiewicza w Poznaniu

Wydział Matematyki i Informatyki

Adam Sosnowski

Nr albumu: 329559

Konwersja tekstu ortograficznego na tekst fonetyczny przy użyciu parsingu płytkiego

Praca magisterska na kierunku:

INFORMATYKA

Promotor:

prof. UAM dr hab. Krzysztof Jassem

Poznań 2012

Oświadczenie

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Spis treści

Wstęp

Konwersja tekstu to automatyczny proces przetwarzania tekstu w formie źródłowej na określoną formę docelową. W tej pracy zajmiemy się problemem normalizacji, czyli konwersji tekstu, gdzie formą źródłową jest tekst ortograficzny, a formą docelową tekst fonetyczny.

W szczególności będzie interesować nas normalizacja takich wyrażeń jak: skróty, skrótowce, liczby czy numery. Wyrażenia takiego typu mają jedną wspólną cechę: zapisujemy je w odmienny sposób niż je czytamy. Ich forma ortograficzna jest więc różna od ich formy fonetycznej.

Procesem, który wykorzystamy do konwersji tekstu będzie parsing płytki, czyli procedura przetwarzania zdań, podczas której nie dochodzi do ich kompletnej analizy lingwistycznej. Podejście to jest alternatywą dla stosowania parsingu głębokiego, gdzie przeprowadzana analiza jest kompletna, ale przez to też dłuższa od analizy częściowej. Narzędziem, które posłuży nam do płytkiego przetwarzania zdań, będzie parser Spejd.

Praca ta złożona jest z dwóch części. Trzy pierwsze rozdziały tworzą razem część teoretyczną, ostatnie dwa zaś składają się na część praktyczną. W rozdziale 1. omówimy zagadnienie konwersji tekstu oraz różne metody jej przeprowadzania. Opiszemy też problemy występujące przy konwersji tekstu. Rozdział 2. poświęcony będzie procesowi składniowego przetwarzania tekstu, a w szczególności parsingowi płytkiemu. Porównamy obie metody przetwarzania składniowego – płytką i głęboką, zwrócimy też uwagę na zalety i wady obu podejść. W rozdziale 3. opiszemy działanie parsera Spejd oraz składnię reguł, którymi się on posługuje. Pokażemy na przykładach, w jaki sposób parser przetwarza tekst. Rozdział 4. zawierać będzie nowe reguły dla parsera, które wykorzystamy podczas procesu normalizacji. Działanie tych reguł zobrazujemy na przykładach. W ostatnim, 5. rozdziale, omówimy implementację algorytmu konwersji tekstu, przeanalizujemy sposób jego działania dla różnych przykładów, po czym przejdziemy do ewaluacji algorytmu.

1 Konwersja tekstu

W rozdziale tym omówimy, czym jest konwersja tekstu: jaka jest definicja tego procesu oraz jej intuicyjne pojmowanie. Omówimy dwie najpowszechniejsze metody konwersji tekstu, po czym na kilku przykładach nakreślimy nasze oczekiwania co do wyników systemu konwertującego tekst ortograficzny na tekst fonetyczny. Na koniec skupimy się na zastosowaniach konwersji i problemach, które mogą przy niej wystąpić.

1.1 Definicje

Tematem tej pracy jest *konwersja tekstu ortograficznego na tekst fonetyczny,* zdefiniujmy więc pojęcia składające się na nazwę tego procesu.

Konwersja tekstu – proces automatycznego przetwarzania tekstu z formy źródłowej na formę docelową.

Tekst ortograficzny (pisany) – tekst w postaci pisanej, zawierający takie elementy jak nierozwinięte skróty i skrótowce, liczby i daty w zapisie liczbowym itp.

Tekst fonetyczny (mówiony) – tekst w formie przystosowanej do poprawnego odczytania. Przykładowo, wszystkie skróty i skrótowce są rozwinięte, a liczby i daty zapisane w postaci słownej.

Konwersja tekstu ortograficznego na tekst fonetyczny (zwana także normalizacją) jest zatem automatycznym procesem zamiany tekstu w formie pisanej na tekst w formie mówionej. Na przykład zdanie o formie ortograficznej *„Mam 2 tys. zł. długu”*, w formie fonetycznej przyjmie postać *„Mam dwa tysiące złotych długu”*.

Intuicyjnie, proces ten analogiczny jest do czynności, którą wykonuje człowiek, podczas czytania pisanego tekstu. Chcąc zautomatyzować ten proces, musimy ustalić, jaka wiedza konieczna jest do przeprowadzenia poprawnej konwersji. Wiedzę tą możemy podzielić na trzy kategorie:

- morfologiczna – informacje na temat fleksji przetwarzanych wyrazów,
- syntaktyczna – informacje na temat funkcji przetwarzanych wyrazów w zdaniu,

- semantyczna – informacje na temat znaczenia przetwarzanych wyrazów.

1.2 Omówienie metod konwersji

Proces konwersji tekstu może być realizowany na kilka sposobów. My zaprezentujemy dwa podejścia to tego problemu: pierwsze, opierające się na skończenie stanowych przetwornikach z wagami, oraz drugie, traktujące konwersję tekstu jako rodzaj tłumaczenia automatycznego. Oba rozwiązania stosowane są w komercyjnych implementacjach algorytmów konwertujących tekst, aczkolwiek nam, w dalszej części tej pracy, bliższe będzie podejście nawiązujące do tłumaczenia automatycznego.

1.2.1 Konwersja z wykorzystaniem przetworników skończenie stanowych z wagami

Automaty skończenie stanowe (ang. Finite-State Models, FSM), a w szczególności przetworniki skończenie stanowe z wagami (ang. Weighted Finite-State Transducers, WFST), są narzędziem wykorzystywanym w wielu zagadnieniach związanych z przetwarzaniem języka, w tym w konwersji tekstu. Są one podstawą systemów zajmujących się zarówno językiem pisanym jak i mówionym (Caseiro, 2004), takich jak:

- tłumaczenie maszynowe
- rozpoznawanie mowy ciągłej z zastosowaniem dużych leksykalnych baz danych (ang. Large Vocabulary Continuous Speech Recognition, LVCSR)
- synteza mowy

Przetworniki stosowane w tych systemach, przyporządkowują różne informacje o danym wyrazie do jednej konkretnej formy zapisu. Informacje te mogą dotyczyć różnych poziomów w opisie przetwarzanego wyrazu, takich jak:

- forma ortograficzna
- opis leksykalny
- opis fonetyczny

Przetworniki mogą być budowane automatycznie na podstawie danych „treningowych" korzystając z technik opartych na korpusach tekstów. W porównaniu do tradycyjnych modeli, gdzie buduje się systemy w oparciu o pewną wiedzę lingwistyczną, systemy oparte na WFST są korzystniejsze od opartych na WFST z uwagi na łatwość, z jaką można implementować lub przybliżać techniki oparte na wiedzy. Różne źródła tej wiedzy mogą być reprezentowane poprzez automaty skończenie stanowe, pozwalając tym samym na integrację wstępnej wiedzy z technikami indukcyjnymi.

Czynniki te przyczyniają się do popularności automatów skończenie stanowych (czy też przetworników skończenie stanowych z wagami) jako narzędzi realizacji procesu konwersji tekstu (Caseiro, 2004).

1.2.2 Konwersja jako przypadek tłumaczenia automatycznego

Podstawą konwersji tekstu w metodzie bazującej na technice tłumaczenia automatycznego jest rozbudowany słownik, którego językiem źródłowym jest forma ortograficzna, a słownikiem docelowym – forma fonetyczna danego języka. Słownik ten przypisuje słowom zapisanym w formie ortograficznej ich formę fonetyczną. Mogą to być na przykład pary:

ok. – około

ur. – urodzony

PKO – pe ka o

MPK – em pe ka

9 – dziewięć

9. - dziewiąty

200 – dwieście

Oprócz tego słownik może zawierać informacje o możliwej funkcji danego słowa w zdaniu, czy też jego prawdopodobnych sąsiadach.

Sam proces normalizacji odbywa się w następujących etapach:

segmentacja tekstu

↓

analiza morfosyntaktyczna

↓

dezambiguacja morfosyntaktyczna

↓

synteza wyjścia

Trzy początkowe etapy, od segmentacji tekstu do dezambiguacji morfosyntaktycznej, zostaną omówione w rozdziale 2. w punkcie 2.1.2.2. Etap syntezy wyjścia zostanie omówiony w rozdziale 5.

1.3 Przykłady konwersji

Przyjrzymy się teraz procesowi konwersji tekstu ortograficznego na tekst fonetyczny na konkretnych przykładach. Postaramy się przewidzieć możliwe błędne konwersje i uściślić, jakiego rezultatu oczekujemy dla różnych typów konwertowanych wyrażeń. Do wniosków tych wrócimy w rozdziale 5. w punkcie 5.3 – *Ewaluacja algorytmu.*

Przykład 1.

Rodzaj wyrażenia	nieodmienny skrót z kropką
Konwertowane wyrażenie	„ok.” (około)
Zdanie wejściowe	Wrócę ok. południa.
Spodziewany wynik konwersji	Wrócę około południa.
Możliwe błędne konwersje	Wrócę ok.
	Wrócę okej południa.
	Wrócę okej. południa.

Przykład 2.

Rodzaj wyrażenia	nieodmienny skrót bez kropki
Konwertowane wyrażenie	„wg” (według)
Zdanie wejściowe	Postępuj wg wskazań instrukcji.
Spodziewany wynik konwersji	Postępuj według wskazań instrukcji.
Możliwe błędne konwersje	Postępuj wg wskazań instrukcji.

Przykład 3.

Rodzaj wyrażenia	odmienny skrót z kropką
Konwertowane wyrażenie	„tys” (tysiąc)
Zdanie wejściowe	Wśród tych trzystu tys. monet nie ma żadnej o wartości równej dwóm tys. pesos.
Spodziewany wynik konwersji	Wśród tych trzystu tysięcy monet nie ma żadnej o wartości równej dwóm tysiącom pesos.
Możliwe błędne konwersje	Wśród tych trzystu tysiąc monet nie ma żadnej o wartości równej dwóm tysiąc pesos .

Przykład 4.

Rodzaj wyrażenia	odmienny skrót bez kropki
Konwertowane wyrażenie	„zł ” (złoty)
Zdanie wejściowe	Jesteś mi winien pięć zł.
Spodziewany wynik konwersji	Jesteś mi winien pięć złotych.
Możliwe błędne konwersje	Jesteś mi winien pięć zł. Jesteś mi winien pięć złoty.

Przykład 5.

Rodzaj wyrażenia	skrótowiec
Konwertowane wyrażenie	„PKO ” (pe ka o)
Zdanie wejściowe	Mam konto w PKO.
Spodziewany wynik konwersji	Mam konto w pe ka o.
Możliwe błędne konwersje	Mam konto w pko.
	Mam konto w pekao.

Komentarz.

Rozwinięcie skrótu „PKO” do formy „pekao” (zamiast „pe ka o”), uważamy za błędne z uwagi na brak odstępów pomiędzy kolejnymi głoskami. Odstępy te zapewnią przy poprawnej konwersji właściwe zaakcentowanie tej formy fonetycznej przez ewentualny syntezator. Więcej na temat zastosowań konwersji powiemy w punkcie 1.4 tego rozdziału.

Przykład 6.

Rodzaj wyrażenia	liczba prosta
Konwertowane wyrażenie	„3” (trzy)
Zdanie wejściowe	Idę do szkoły z 3 książkami.
Spodziewany wynik konwersji	Idę do szkoły z trzema książkami.
Możliwe błędne konwersje	Idę do szkoły z trzy książkami.

Przykład 7.

Rodzaj wyrażenia	liczba prosta
Konwertowane wyrażenie	„20” (dwadzieścia)
Zdanie wejściowe	Mam 20 książek.
Spodziewany wynik konwersji	Mam dwadzieścia książek.
Możliwe błędne konwersje	Mam dwa zero książek.
	Mam dwudziestu książek.

Przykład 8.

Rodzaj wyrażenia	liczba złożona
Konwertowane wyrażenie	„23 ” (dwadzieścia trzy)
Zdanie wejściowe	Brakuje mi 23 złotych.
Spodziewany wynik konwersji	Brakuje mi dwudziestu trzech złotych.
Możliwe błędne konwersje	Brakuje mi dwa trzy złotych.
	Brakuje mi dwadzieścia trzy złotych.
	Brakuje mi dwadzieścia trzech złotych.

Przykład 9.

Rodzaj wyrażenia	numer
Konwertowane wyrażenie	numer PESEL
Zdanie wejściowe	Mój numer PESEL to 88080377223.
Spodziewany wynik konwersji	Mój numer PESEL to osiemdziesiąt osiem zero osiem zero trzy siedemdziesiąt siedem dwieście dwadzieścia trzy.
Możliwe błędne konwersje	Mój numer PESEL to osiem osiem zero osiem zero trzy siedem siedem dwa dwa trzy.
	Mój numer PESEL to osiemdziesiąt osiem miliardów osiemdziesiąt milionów trzysta siedemdziesiąt siedem tysięcy dwieście dwadzieścia trzy.

Komentarz.

Pierwszą błędną konwersję („*Mój numer PESEL to osiem osiem zero osiem zero trzy siedem siedem dwa dwa trzy.*”) można by uznać za poprawną, jednak nie różniłaby ona się od prostej konwersji numeru każdej postaci, polegającej na przekształceniu numeru cyfra po cyfrze. Jako że nie jest to rezultat, którego oczekiwalibyśmy przy konwersji wyrażenia tego typu, uznajemy go za błędny.

1.4 Zastosowanie

Konwersja tekstu znajduje zastosowanie w wielu różnych dziedzinach życia, często jako podstawowy lub wstępny proces przetwarzania. Najpowszechniejsze zastosowania konwersji tekstu to:

- ujednoznacznianie tekstu
- synteza mowy
- tłumaczenie automatyczne
- porównywanie tekstów
- optymalizacja baz danych

1.5 Problemy występujące przy konwersji tekstu

Podczas konwersji tekstu możemy napotkać na kilka problemów. Część została omówiona już w punkcie 1.3 tego rozdziału, w przywołanych przez nas przykładach. Problemy te możemy podzielić na kilka, przedstawionych niżej, kategorii.

1.5.1 Homografy

W konwertowanym przez nas tekście mogą wystąpić homografy, czyli ciągi znaków o wielu interpretacjach. Przykładem homografu jest „p.” który rozwinąć można zarówno do „pan” jak i do „patrz”, „pani”, „piętro”, „punkt”, w zależności od kontekstu. Podobnie, pewne ciągi liczbowe mogą przedstawiać zarówno liczbę jak i datę (przykładowo „3.14” może przedstawiać liczbę lub datę czternastego kwietnia).

1.5.2 Tokenizacja

Część problemów wiąże się już z pierwszym etapem przetwarzania tekstu, czyli z tokenizacją. Przykładowo, liczba 9801 może zostać zapisana jako „9 801”. Przez dodaną spację system podczas tokenizacji może podzielić tę liczbę na dwa tokeny, co znacznie utrudni jej poprawną konwersję („dziewięć tysięcy osiemset jeden”).

1.5.3 Funkcja kropki w zdaniu

Konwertując tekst, musimy odpowiednio interpretować, jaką funkcję pełni kropka w przetwarzanym przez nas tekście. Najczęstszą funkcją kropki w zdaniu jest oczywiście jego zakończenie, jak w przykładzie: „*Mam konto w PKO.*”. Kropka może też być częścią skrótu, jak w zdaniu: „*Wrócę ok. południa.*”. Jednakże zdarzają się przypadki, kiedy obie te funkcje występują równocześnie, jak w zdaniu: *„Urodziłem się w 1988 r.”*. Dla niektórych skrótów taka sytuacja będzie normą („itp.”, „itd.”), dla innych jednak (jak w przytoczonym wyżej przykładzie) skrót może występować zarówno na końcu jak i w środku zdania („*W 1988 r. urodził się Adam Sosnowski.*”).

1.5.4 Liczebniki

Liczby zapisane w postaci cyfr, występujące w konwertowanym przez nas tekście, muszą zawsze zostać przetworzone. Jednakże sposób ich rozwinięcia zależy od kategorii, do której przypiszemy daną liczbę. Może ona być liczebnikiem głównym, jak w zdaniu „*Mam 3 książki*”, lub liczebnikiem porządkowym - „*Podaj mi 3. książkę”*. Daną liczbę możemy też chcieć rozwinąć mając na uwadze kontekst, w którym występuje. Wspomniane wcześniej „3.14” może być zarówno datą (trzeciego kwietnia) jak i zaokrągleniem liczby π do części setnych. Ponadto, bardziej złożone liczby – nazwijmy je numerami – możemy chcieć rozwijać w różny sposób. Pewne numery konwertować będziemy cyfra pod cyfrze, podczas gdy inne grupując cyfry w liczby wielocyfrowe. Numery kont bankowych grupuje się czwórkami, kody pocztowe odczytywane są zazwyczaj jako para i trójka, a PESEL możemy chcieć rozwinąć jako cztery pary i trójka.

1.5.5 Fleksja

Ostatni rodzaj błędów dotyczy fleksji rozwijanych wyrażeń, a konkretniej nieprecyzyjnych reguł dezambiguacji morfosyntaktycznej (więcej na ten temat w punkcie 2.1.2.2). Może się zdarzyć, że stworzona przez nas reguła konwersji działa dla wszystkich różnych przypadków, z wyjątkiem jednego. Ilustracją takiej sytuacji jest przykład 7. z punktu 1.3. Zdanie z tego przykładu: „*Mam 20 książek.*” zawiera liczbę „20” którą należy rozwinąć do „dwadzieścia”. Jednak w wyniku analizy morfosyntaktycznej, z uwagi na sąsiedztwo wyrazu „**książek**”, do tokenu „20” przypisane zostały dwa możliwe rozwinięcia - „dwadzieścia” oraz „dwudziestu”. Stało się tak, ponieważ poprawne są zdania: „*Mam dwadzieścia **książek***” oraz „*Nie mam dwudziestu **książek***”. Nasza reguła konwersji nie jest w stanie wybrać poprawnej formy dla tokenu „20”, wnioskując tylko na podstawie najbliższego sąsiada tego wyrażenia.

2 Parsing płytki

Przedstawione w poprzednim rozdziale problemy występujące przy konwersji tekstu istotnie utrudniają ten proces. Jednakże część z nich udaje się rozwiązać, korzystając z narzędzi przetwarzania języka naturalnego. Przetwarzanie to może odbywać się na poziomie morfologicznym, składniowym, semantycznym i pragmatycznym. W tym rozdziale skoncentrujemy się na przetwarzaniu składniowym – a konkretniej – parsingu płytkim.

2.1 Podstawowe pojęcia

Zanim przejdziemy jednak do istoty tego rozdziału, zdefiniujmy podstawowe pojęcia, którymi będziemy się posługiwać.

2.1.1 Definicje

Przetwarzanie języka naturalnego (ang. natural language processing) – *wszelkie prace zmierzające do automatycznego tworzenia lub przetwarzania wypowiedzeń, związane ze znaczeniem lub strukturą lingwistyczną tych wypowiedzeń* (Przepiórkowski 2008).

Parsowanie morfologiczne – przetwarzanie mające na celu przypisanie każdemu słowu w tekście wejściowym jego interpretacji gramatycznej. W zależności od złożoności tego procesu, na wyjściu możemy otrzymać mniej lub bardziej szczegółową interpretację gramatyczną.

Parsowanie składniowe – łączenie słów z tekstu wejściowego w związki strukturalne. Istnieje wiele podejść do parsingu składniowego, a samo to zagadnienie dzieli się na wiele różnych podkategorii. Nas interesować będzie przede wszystkim płytkie i – w mniejszym stopniu – głębokie przetwarzanie składniowe (patrz 2.1.2).

Parsowanie semantyczne – proces polegający na automatycznym określaniu znaczenia zdań w przetwarzanym tekście lub na budowie słowników semantycznych.

Parsowanie pragmatyczne – przetwarzanie identyfikujące wiele różnych odniesień do tego samego bytu lub analizujące struktury dialogów.

2.1.2 Przetwarzanie składniowe – dwa podejścia

2.1.2.1 Przetwarzanie głębokie

Głębokie przetwarzanie języka naturalnego (ang. Deep Natural Language Processing – DNLP), polega na analizowaniu wyrażeń językowych przy wykorzystaniu jak najszerszej wiedzy lingwistycznej (Schäfer, 2006). W informatyce przetwarzanie takie często określane jest terminami: „information-based", „knowledge representation-based", czy też „constraint-based", gdyż wiedza o języku naturalnym nie jest zapisana w postaci algorytmów lub prostych baz danych. W rzeczywistości, wiedza o języku jest oddzielona (z reguły) od prostych algorytmów w postaci gramatyki formalnej szerokiego podzbioru analizowanego języka.

Wynik analizy wyrażeń językowych zawiera wiele różnych możliwych hipotez struktury (znaczenia) każdego zdania, co odzwierciedla niepewność wyboru jednej z nich. Zdarza się też, że wynik analizy jest błędny, co ma miejsce w przypadku kiedy wiedza lingwistyczna systemu jest niewystarczająca do zanalizowania konkretnego wejścia.

Parsery głębokie (systemy analizy wykorzystujące głębokie przetwarzanie języka naturalnego) są zwykle oparte na regułach. Reguły te opisują ograniczenia poprawnych kompozycji jednostek lingwistycznych (zasady składni) opartych na gramatycznej teorii lingwistycznej. Abstrahują one jednak od samych słów, które są zakodowane w leksykonie. Używanie samej analizy syntaktycznej opartej o taką gramatykę (bez interpretacji semantycznych) może być podstawą zastosowań takich jak *sprawdzanie poprawności gramatycznej*, gdzie odpowiednie użycie składni języka naturalnego jest albo zweryfikowane pozytywnie, albo negatywnie.

Stosując wyniki analizy syntaktycznej, reguły mogą także opisywać reprezentację semantyczną zdania w języku naturalnym. Przez reprezentację semantyczną rozumiemy reprezentację znaczenia języka naturalnego, abstrakcyjny i uproszczony język opisujący (w postaci formuł logicznych) znaczenie reprezentowanego syntaktycznie wyrażenia języka naturalnego. Przykładowo język opisu reprezentacji semantycznych może być oparty na logice pierwszego rzędu, jak w poniższym przykładzie dla zdania „Jan dał Marii książkę":

przeszły(dać(Jan, Maria, książka))

Termin *struktura głęboka* wprowadzony został przez Chomsky'ego (1965), w znaczeniu abstrakcyjnej budowy zdania, wspólnej dla wielu *form powierzchniowych.* Przykładowo, takimi formami powierzchniowymi byłyby wyrażenia: „Jan dał Marii książkę" i „Książka została dana Marii przez Jana". Jako, że posiadają one to samo znaczenie, ich struktura głęboka jest taka sama. Chomsky zamienił później termin *struktura głęboka* na *forma logiczna,* a termin *forma powierzchniowa* na *forma fonetyczna*. Jednakże można zauważyć, że przetwarzanie głębokie to właśnie przetwarzanie, w wyniku którego powstaje *struktura głęboka.*

Skonstruowana reprezentacja semantyczna może być użyta w dalszych zastosowaniach, takich jak: *ekstrakcja relacji, podsumowanie tekstu, question answering, interpretacja poleceń,* itp. Co więcej, jako że idealna reprezentacja semantyczna byłaby niezależna od języka (interlingua, sztuczny neutralny język pomocniczy, z łac. *międzyjęzyk*), mogłaby zostać ona wykorzystana przy tłumaczeniu automatycznym.

W idealnym, wielojęzykowym środowisku przetwarzania głębokiego, składnia i leksykon są określone dla danego języka, a reprezentacja semantyczna abstrahuje od warstwy syntaktycznej. Prostym przykładem ilustrującym tę koncepcję jest określenie dla zdania „Rolnik czyta gazetę" następującej reprezentacji:

$$\text{sem\_rolnik}(y) \wedge \text{sem\_gazeta}(z) \wedge \text{sem\_przechodni\_czytać}(x,y,z) \wedge \text{sem\_czas\_teraźniejszy}(x)$$

W myśl tej koncepcji taką samą reprezentację semantyczną (w uproszczeniu) miałyby zdania w innych językach:

angielski: *The farmer reads the newspaper*

francuski: *Le paysan lit le journal*

niemiecki: *Der Bauer liest die Zeitung*

Założenie wspólnej struktury głębokiej dla zdań różnych języków nie sprawdza się w praktyce, gdyż nie bierze pod uwagę m. in. semantyki leksykalnej (przykładowo: wieloznaczność wyrazów), czy też bardziej złożonych konstrukcji.

Z uwagi na intensywne wykorzystanie wiedzy lingwistycznej w głębokim przetwarzaniu

języka naturalnego, proces ten wymaga znaczącej mocy obliczeniowej. Jednakże badania nad poprawą wydajności głębokiego przetwarzania są cały czas prowadzone (Callmeier, 2000; Uszkoreit, 2002) i dziś ten problem nie jest już tak poważny (Schäfer, 2006).

2.1.2.2 Przetwarzanie płytkie

W przeciwieństwie do przetwarzania głębokiego, przetwarzanie płytkie (ang. Shallow Natural Language Processing – SNLP) nie ma na celu przeprowadzenia kompletnej analizy lingwistycznej. Parsery płytkie (systemy wykorzystujące płytkie przetwarzanie języka naturalnego) są projektowane i konstruowane do konkretnych i sprecyzowanych zadań. Nie wykorzystują one wielu informacji z wyrażeń, jakie otrzymują na wejściu, nie korzystają też ze skomplikowanych gramatyk, czy obszernej wiedzy lingwistycznej (Schäfer, 2006).

Dzięki temu, że parsery płytkie opierają się na gramatykach regularnych lub rozwiązaniach statystycznych, są one w ogólności szybsze niż parsery głębokie. Ceną za tę zaletę jest jednak to, co otrzymujemy na wyjściu działania takiego parsera. Wynikiem analizy są proste, **częściowe**, niekompletne reprezentacje. Przetwarzanie płytkie (zwane niekiedy również powierzchniowym) odbywa się w kilku etapach:

segmentacja tekstu

↓

analiza morfosyntaktyczna

↓

dezambiguacja morfosyntaktyczna

Segmentacja tekstu – *liniowy podział tekstu na co najmniej dwóch poziomach. Po pierwsze, tekst jest często dzielony na jednostki, zwykle zdania, które mogą być przetwarzane syntaktycznie niezależnie od innych jednostek tego samego poziomu. Po drugie, tekst jest dzielony na jednostki, nazywane tokenami lub segmentami, którym można przypisać interpretacje morfosyntaktyczne* (Przepiórkowski, 2008).

Analiza morfosyntaktyczna – *polega na przypisaniu segmentom wszystkich możliwych interpretacji morfosyntaktycznych. Na przykład segmentowi „piec" mogą zostać przypisane 3 interpretacje morfosyntaktyczne: bezokolicznik, rzeczownik w mianowniku i rzeczownik w bierniku* (Przepiórkowski, 2008).

Dezambiguacja morfosyntaktyczna – *polega na wybraniu właściwych interpretacji spośród wszystkich możliwych interpretacji przysługujących danemu segmentowi. Zwykle celem tego etapu jest całkowite ujednoznacznienie morfosyntaktyczne tekstu, choć czasem dopuszcza się pozostawienie kilku możliwych interpretacji dla niektórych segmentów* (Przepiórkowski, 2008).

W związku z zaobserwowanym brakiem wydajności i solidności parserów głębokich, w ostatnich latach to parsery płytkie były częściej i chętniej rozwijane. Wiele z nich jest w stanie analizować megabajty tekstu w ciągu kilku sekund, jednakże obarczone są one wadami takimi jak niska precyzja i jakość analizy (Schäfer, 2006).

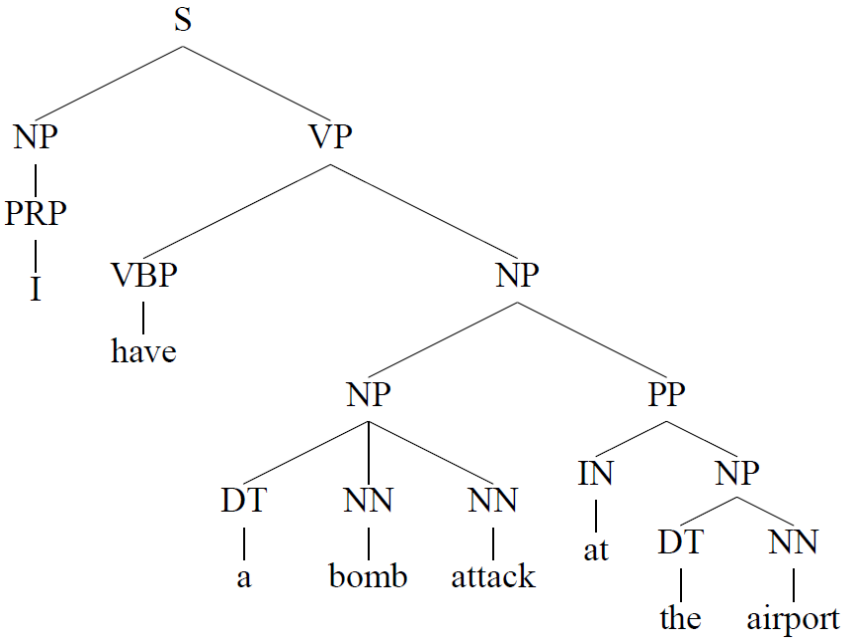
Pojęcie parsingu płytkiego ewoluowało na przestrzeni ostatnich dziesięcioleci. Początkowo oznaczało ono proces mający na celu dokonanie pełnej analizy zdań, jednak tylko w zakresie syntaktycznym. Aspektem, który odróżniał parsery głębokie od płytkich, był fakt analizy zarówno na przestrzeni syntaktycznej zdań, jak i ich semantyki.

Z czasem jednak pojęcie parsingu płytkiego zawężono jedynie do parsingu częściowego (ang. *chunk parsing*). Tym co odróżnia parsing płytki i głęboki w tym podejściu, nie jest przestrzeń analizy, a jej kompletność. W tej pracy, pojęcie parsingu płytkiego będziemy rozumieć właśnie w powyższy sposób, co odbiega od definicji Chomsky'ego (1965) czy Schäfera (2006), ale zgodne jest z podejściem przedstawionym przez Przepiórkowskiego (2008).

2.2 Przykłady działania

W celu zilustrowania różnic pomiędzy różnymi podejściami do tematu przetwarzania tekstu, omówimy teraz kilka przykładów działania parserów płytkich (dokonujących częściowej analizy składniowej), oraz parserów głębokich (dokonujących pełnej analizy składniowej).

Przykład 1. Zdanie „*I have a bomb attack at the airport*” (przykład za Swift, Allen, Gildea: *Skeletons in the parser: Using a shallow parser to improve deep parsing*). Parser: Collins Parser.



Rysunek (2.1): Drzewo analizy morfosyntaktycznej
zdania „*I have a bomb attack at the airport*”

Wynik parsera głębokiego został przedstawiony na rysunku (2.1) w postaci drzewa, by czytelniej zobrazować jego charakterystykę. Jako że jest to pierwszy przykład przetwarzania języka naturalnego, postaramy się szczegółowo go omówić. W wyniku przeprowadzonej analizy morfosyntaktycznej zostały utworzone następujące jednostki:

- zdanie (S) składające się z grupy rzeczownikowej (NP) i grupy czasownikowej (VP)
- grupa rzeczownikowa (NP) składająca się z zaimka rzeczownego (PRP)
- zaimek rzeczowny (PRP) składający się z wyrazu „I”
- grupa czasownikowa (VP) składająca się z czasownika w liczbie pojedynczej, czasu teraźniejszego (VBP) i grupy rzeczownikowej
- czasownik w liczbie pojedynczej, czasu teraźniejszego (VBP) składający się z wyrazu „have”
- grupa rzeczownikowa (NP) składająca się z grupy rzeczownikowej (NP) i grupy przyimkowej (PP)
- grupa rzeczownikowa (NP) składająca się z określnika (DT), rzeczownika (NN) i rzeczownika (NN)

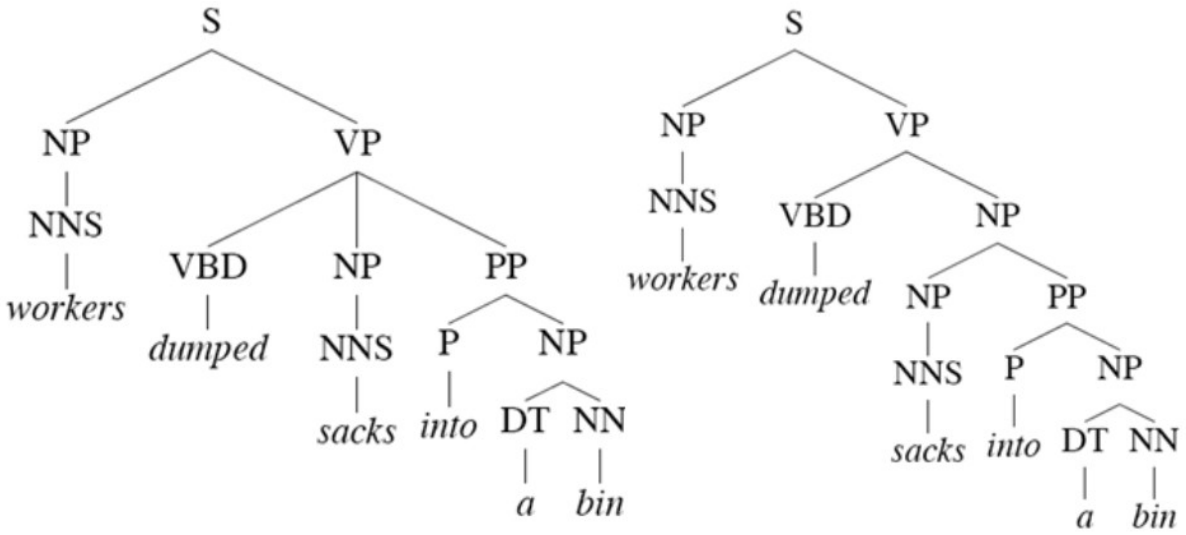
- określnik (DT) składający się z wyrazu „a”
- rzeczownik (NN) składający się z wyrazu „bomb”
- rzeczownik (NN) składający się z wyrazu „attack”
- grupa przyimkowa (PP) składająca się z przyimka (IN) i grupy rzeczownikowej (NP)
- przyimek (IN) składający się z wyrazu „at”
- grupa rzeczownikowa (NP) składająca się z określnika (DT) i rzeczownika (NN)
- określnik (DT) składający się z wyrazu „the”
- rzeczownik (NN) składający się z wyrazu „airport”

Jako że powyższy przykład ilustrował działanie parsera głębokiego, otrzymaliśmy pełną reprezentację analizowanego zdania.

Jak wspominaliśmy już w punkcie 2.1.2.2, parsery płytkie projektowane są do konkretnych i sprecyzowanych zadań. Przykładem takiego zadania może być wyszukiwanie wszystkich grup rzeczownikowych w analizowanych zdaniach. Wówczas, dla zdania „*I have a bomb attack at the airport*”, parser płytki zwróciłby następujące grupy:

- grupa rzeczownikowa (NP) składająca się z zaimka rzeczownego (PRP)
- grupa rzeczownikowa (NP) składająca się z grupy rzeczownikowej (NP) i grupy przyimkowej (PP)
- grupa rzeczownikowa (NP) składająca się z określnika (DT), rzeczownika (NN) i rzeczownika (NN)
- grupa rzeczownikowa (NP) składająca się z określnika (DT) i rzeczownika (NN)

Przykład 2. Zdanie „*Workers dumped sacks into a bin*” (przykład za Donaldson, Natural Language Processing: Statistical parsing). Parser: Collins Parser.



Rysunek (2.2): Drzewa analizy morfosyntaktycznej zdania „Workers dumped sacks into a bin”

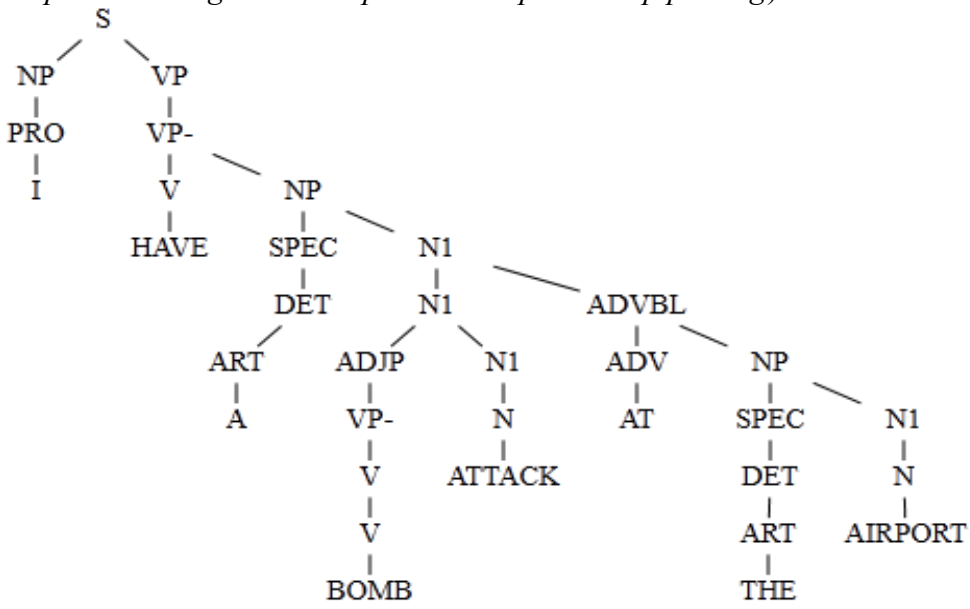
Powyższe wyniki głębokiego przetwarzania języka naturalnego są dowodem niejednoznaczności tego procesu. Oba drzewa na rysunku (2.2) są syntaktycznie poprawnymi reprezentacjami zdania z przykładu 2. Różnica polega na miejscu występowania grupy przyimkowej „into a bin”. W pierwszym drzewie grupa ta dołączona jest do grupy czasownikowej, razem z czasownikiem „dumped” i grupą rzeczownikową „sacks”. W drugim natomiast nasza grupa przyimkowa „into a bin” dołączona jest do grupy rzeczownikowej, razem z grupą rzeczownikową „sacks”.

Dla tego samego zdania, można by rozważyć sposób działania parsera płytkiego wyszukującego jedynie rzeczowniki w analizowanych zdaniach. Wówczas wynikiem analizy byłaby prosta lista odnalezionych rzeczowników (*workers*, *sacks*, *bin*), wraz z ich opisem gramatycznym. Należy zwrócić uwagę na fakt, że wynik tej analizy jest o wiele uboższy od wyniku analizy parsera głębokiego, jednakże jest on w pełni jednoznaczny.

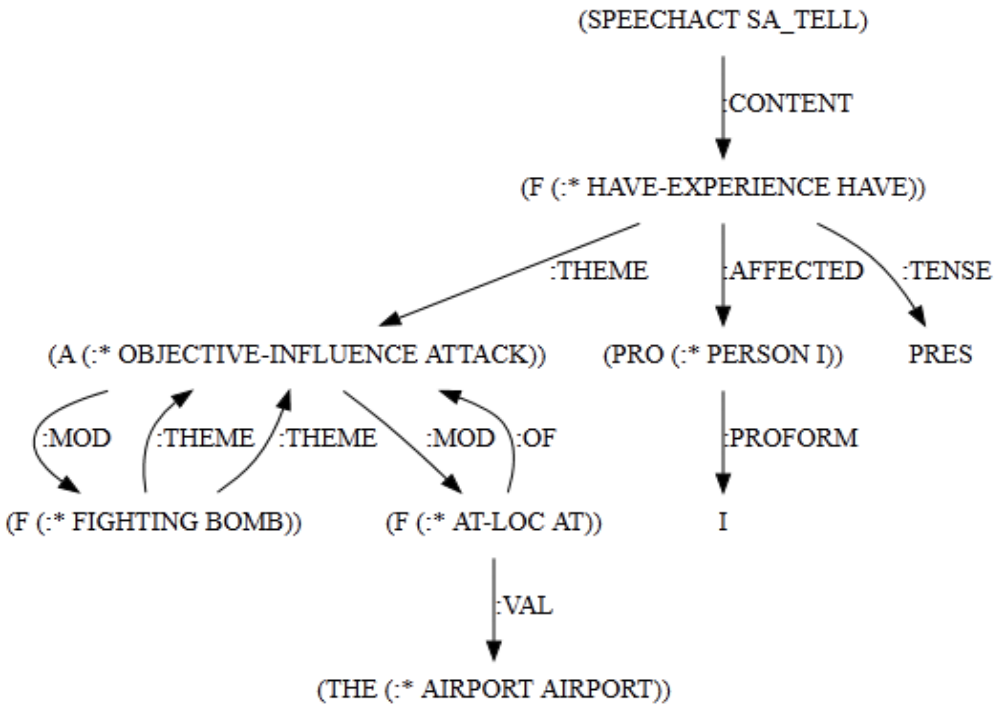
2.3 Analiza semantyczna

Tak jak wspominaliśmy w punkcie 2.1.2.1 tej pracy, niekiedy parsery głębokie, oprócz dokonywania pełnej analizy syntaktycznej, mogą także tworzyć reprezentację semantyczną zdania. Przyjrzyjmy się przykładom działania takich parserów na zdaniach z przykładów 1 i 2 punktu 2.2 tego rozdziału.

Przykład 1. Zdanie „*I have a bomb attack at the airport*” (przykład za Swift, Allen, Gildea: *Skeletons in the parser: Using a shallow parser to improve deep parsing*). Parser: TRIPS Parser.



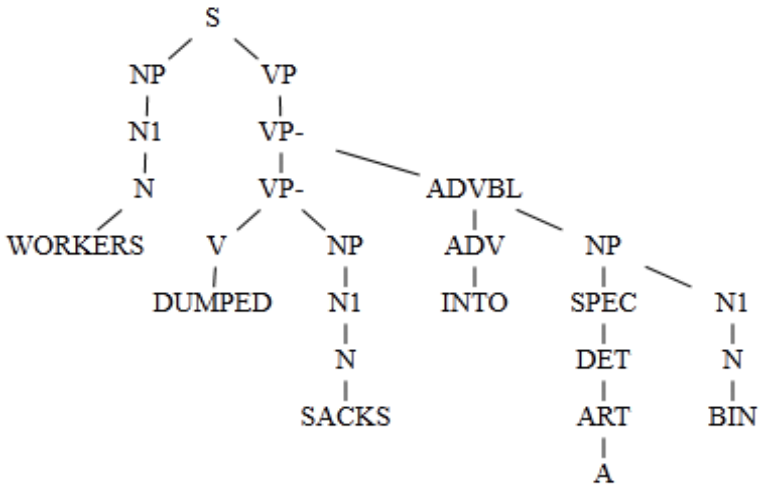
Rysunek (2.3): Drzewo analizy syntaktycznej i semantycznej zdania „I have a bomb attack at the airport”



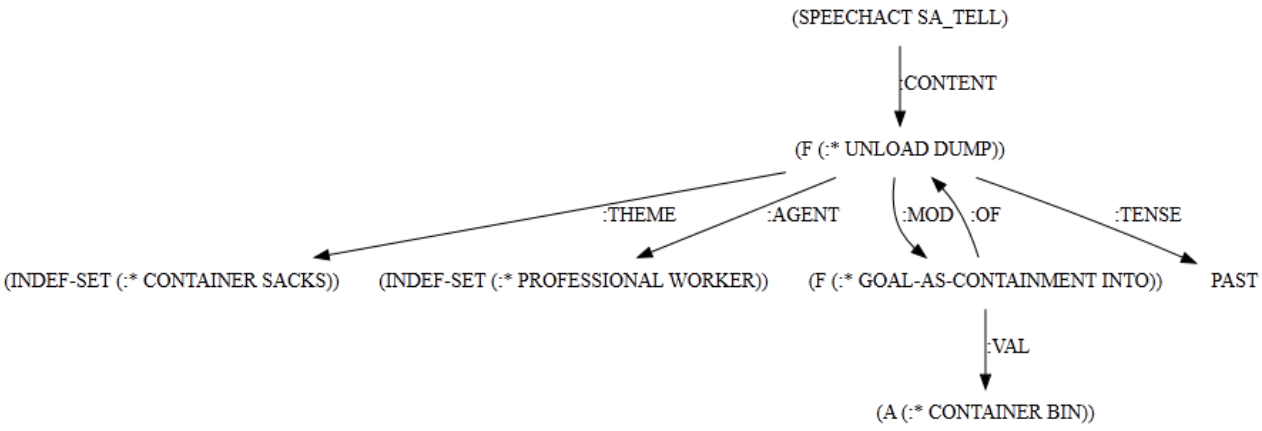
Rysunek (2.4): Zależności logiczne w zdaniu „I have a bomb attack at the airport”

Wynik został przedstawiony tutaj w dwojaki sposób: jako drzewo na rysunku (2.3) – analogicznie do przykładów z punktu 2.2, oraz w formie zależności logicznych na rysunku (2.4), co możliwe jest dzięki zastosowaniu analizy semantycznej. Widzimy, że struktura drzewa bardzo przypomina tę z przykładu 1. w punkcie 2.2, ale zawiera bardziej szczegółowe informacje na temat poszczególnych segmentów. Fakt ten lepiej obrazuje drugi sposób przedstawienia tego samego wyniku, gdzie widzimy relacje logiczne pomiędzy poszczególnymi segmentami. Relacje te są właśnie implementacją semantycznej reprezentacji zdania, o której wspominaliśmy w punkcie 2.1.2.1.

Przykład 2. Zdanie „*Workers dumped sacks into a bin*" (przykład za Donaldson, Natural Language Processing: Statistical parsing). Parser: TRIPS Parser.



Rysunek (2.5): Drzewo analizy syntaktycznej i semantycznej zdania „Workers dumped sacks into a bin"



Rysunek (2.6): Zależności logiczne w zdaniu „Workers dumped sacks into a bin"

Podobnie jak w poprzednim przykładzie, tutaj także otrzymujemy znacznie bogatsze struktury. Analiza semantyczna pozwoliła na ujednoznacznienie składniowe przetwarzanego tekstu.

2.4 Zalety i wady parsingu płytkiego

W punkcie 2.1.2.2 wspominaliśmy już zarówno o zaletach jak i wadach parsingu płytkiego. Dokładniej przedstawia je poniższa tabela (Schäfer, 2006), jednocześnie porównując różne własności pomiędzy parsingiem płytkim a głębokim.

Własność	**Parsing głeboki**	**Parsing płytki**
Koszt przetwarzania	wysoki	niski
Pokrycie	niskie	wysokie
Odporność na niepoprawne wejście	niska	wysoka
Wieloznaczność wyjścia	wysoka	niska
Analiza syntaktyczna	szczegółowa	ogólna
Precyzja	wysoka	niska
Analiza częściowa	sporadyczna	**częsta**

3 Parser Spejd

W niniejszym rozdziale omówimy płytki parser Spejd, przy pomocy którego dokonywać będziemy konwersji tekstu ortograficznego na tekst fonetyczny. Po wstępnym opisie systemu Spejd przeanalizujemy składnię reguł, którymi się on posługuje, a na koniec przedstawimy działanie parsera na kilku przykładach.

3.1 Opis parsera

Parser Spejd – Składniowy Parser (Ewidentnie Jednocześnie Dezambiguator) – to parser powierzchniowy opracowany w Instytucie Podstaw Informatyki PAN. Parser umożliwia identyfikowanie konstrukcji składniowych przy jednoczesnym ujednoznacznianiu wyników analizy morfologicznej.

Na wejściu parser Spejd przyjmuje jeden z dwóch formatów:

- format XML, zgodny z formatem plików morph.xml w Korpusie IPI PAN, z obowiązkowym atrybutem id przy elementach <tok>. Przykład (3.1) obrazuje ten format. Przedstawiony jest tam ciąg „*Przykładowym zdaniem*” w formacie XML, gdzie kilka interpretacji tokenu „*Przykładowym*” zostało usuniętych,
- czysty, nieoznakowany tekst.

Za pomocą reguł, o których więcej powiemy w punkcie 3.2, parser przetwarza otrzymane wejście, a uzyskany wynik ma taki sam format jak ten ukazany w przykładzie (3.1). Efekty działania reguł mogą być następujących typów:

- odrzucenie interpretacji tokenu (poprzez dodanie atrybutu „disamb_sh” równego 0 do elementu <lex>),
- dodanie interpretacji do tokenu (poprzez dodanie nowego elementu <lex>),
- pogrupowanie tokenów w grupy składniowe (<group>) i wyrazy składniowe (<syntok>).

```
<tok id="tA5">
        <orth>Przykładowym</orth>
        <lex>
                <base>przykładowy</base>
                <ctag>adj:sg:inst:m1:pos</ctag>
        </lex>
        <lex>
                <base>przykładowy</base>
                <ctag>adj:sg:inst:n1:pos</ctag>
        </lex>
        <lex>
                <base>przykładowy</base>
                <ctag>adj:sg:inst:n2:pos</ctag>
        </lex>
        <lex>
                <base>przykładowy</base>
                <ctag>adj:sg:loc:m1:pos</ctag>
        </lex>
        <lex>
                <base>przykładowy</base>
                <ctag>adj:pl:dat:f:pos</ctag>
        </lex>
</tok>
<tok id="tA6">
        <orth>zdaniem</orth>
        <lex>
                <base>zdanie</base>
                <ctag>subst:sg:inst:n2</ctag>
        </lex>
        <lex>
                <base>zdać</base>
                <ctag>ger:sg:inst:n2:perf:aff</ctag>
        </lex>
</tok>
```

Przykład (3.1): Format XML wejścia dla parsera Spejd – przykład opisu dla tekstu „Przykładowym zdaniem”

3.2 Reguły

Reguły parsera Spejd składają się z kilku części o ustalonym porządku. Te części to kolejno: *Rule*, *Left*, *Match*, *Right* i *Eval*. Spośród nich *Rule*, *Match* i *Eval* muszą zawsze wystąpić w regule, a *Left* i *Right* są nieobowiązkowe.

Pierwsza część reguły zaczyna się od słowa kluczowego „*Rule*”. Jest to część zawierająca nazwę (identyfikator) reguły. Może być to ciąg znaków ujęty w cudzysłowy lub identyfikator zaczynający się od litery i zawierający jedynie litery, cyfry, dywizy i znaki podkreślenia. Przykładem części *Rule* reguły parsera Spejd może być więc:

Rule „Reguła przykładowa”

Kolejną obowiązkową częścią reguły rozpoczyna się od słowa kluczowego „*Match*”. W tej części, po dwukropku, określamy, jaka jest specyfikacja szukanego ciągu, do którego zastosowana ma być nasza reguła. Część ta musi być zakończona dwukropkiem. Przykładem części *Match* reguły parsera Spejd może być więc:

Match: [pos~”adj”][pos~”subst”];

Taki fragment w naszej regule spowoduje dopasowanie jej do ciągu dwóch tokenów, gdzie pierwszy jest przymiotnikiem, a drugi rzeczownikiem. Szczegółowe zasady dopasowania omówimy w punkcie 3.2.1.

Ostatnim koniecznym elementem reguły jest część *Eval*. Rozpoczyna się ona od słowa kluczowego „*Eval*” i dwukropka, po którym następuje ciąg warunków i operacji, dotyczących odpowiednich fragmentów z tekstu dopasowanego przez naszą regułę. Przykładem części *Eval* reguły parsera Spejd może być więc:

Eval: unify(case number gender,1,2); group(NG,2,1);

Pierwsza operacja przykładowej części *Eval* spowoduje uzgodnienie przypadka, liczby i rodzaju pierwszego i drugiego tokenu. Oznacza to, że wszystkie interpretacje z obu tokenów, w których kombinacje wartości tych kategorii gramatycznych z jednego tokenu nie występują w żadnej interpretacji drugiego tokenu, oznaczane są jako błędne. Operacja taka jest przykładem ujednoznacznienia morfosyntaktycznego. Druga operacja to połączenie obu tokenów w grupę nominalną („NG") z drugim tokenem jako centrum składniowym i pierwszym tokenem jako centrum semantycznym. Szczegółowe zasady warunków i operacji występujących w części *Eval* omówimy w punkcie 3.2.2.

Pozostałe dwie części reguł – *Left* i *Right* – tak jak wspomnieliśmy wcześniej, są nieobowiązkowe. Mają analogiczną składnię do części *Match*, ale opisują odpowiednio lewe i prawe sąsiedztwo ciągu wskazanego w *Match*. Do tokenów wskazanych przez *Left* i *Right* można odwoływać się w części *Eval*.

Poza wyżej wymienionymi elementami, reguła parsera Spejd może zawierać komentarze. Komentarze to ciągi znaków, rozpoczęte symbolem „#" i kończące się wraz z końcem wiersza.

W całości, nasza przykładowa reguła, będzie wyglądać następująco:

```
Rule „Reguła przykładowa"
# przykładowy komentarz
Match: [pos~"adj"][pos~"subst"];
Eval: unify(case number gender,1,2); group(NG,2,1);
```

3.2.1 Dopasowanie

Dopasowanie ciągu do części *Match* (opcjonalnie *Left*, *Right*), jest warunkiem mającym na celu zapewnienie, że reguła zadziała na odpowiednim fragmencie przetwarzanego tekstu. Jak wspomnieliśmy wcześniej, po *Match*, *Left* i *Right* zawsze występuje dwukropek, po którym następuje specyfikacja ciągu, który ma zostać dopasowany. Specyfikacja ta może być następującego typu:

- specyfikacja tokenu, np. [pos~~"prep"] lub [base~"co|kto"] (forma leksemu „co" lub „kto"),
- specyfikacja grupy, np. [synh=[pos~~"num"]] (grupa, której centrum składniowe jest

jednoznacznie liczebnikiem),

- specyfikacja specjalna: ns (brak spacji), sb (początek zdania), se (koniec zdania),
- alternatywa postaci „(X | Y)”, gdzie X i Y to dowolna, wyżej wymieniona specyfikacja.

Każda z tych specyfikacji może być także uzupełniona jednym z trzech kwantyfikatorów wyrażeń regularnych:

- ? – dana specyfikacja może wystąpić raz lub nie wystąpić wcale,
- * – dana specyfikacja może wystąpić dowolną liczbę razy (także 0),
- + – dana specyfikacja może wystąpić dowolną liczbę razy (ale co najmniej raz).

Warto też zaznaczyć różnicę pomiędzy operatorami ~ i ~~ – np. [pos~”subst”] i [pos~~”subst”]. Ten pierwszy („~”) spowoduje dopasowanie takich tokenów, które posiadają co najmniej jedną interpretację rzeczownikową, podczas gdy operator „~~” dopasuje tylko te tokeny, których wszystkie interpretacje są rzeczownikowe.

3.2.2 Warunki i operacje

Ostatnim aspektem reguł parsera Spejd, jaki omówimy, są różne warunki i operacje składające się na część *Eval* reguł. W ogólności jest to zbiór wszystkich akcji, jakie mogą zostać wykonane na odpowiednich tokenach wskazanych w sekcji *Match* (opcjonalnie *Left* i *Right*). Akcje te mogą należeć do jednej z trzech kategorii:

- warunki morfosyntaktyczne (agree, orthnot),
- operacje morfosyntaktyczne (unify, persistent_unify, delete, leave, add, set)
- operacje składniowe (word, group, join, attach)

Omówmy więc krótko wszystkie możliwe warunki i operacje:

Warunki morfosyntaktyczne

- **agree**(lista kategorii gramatycznych, referencja_1, referencja_2, …, referencja_n) – sprawdza zgodność atrybutów z podanej listy kategorii gramatycznych, we wszystkich

jednostkach składniowych wskazanych przez referencje. Jako że jest to warunek morfosyntaktyczny (a nie operacja), samo uzgodnienie nie jest przeprowadzane (służy do tego operacja *unify*). W przypadku braku zgodności wynikiem ewaluacji warunku jest fałsz.

- **orthnot**(”wyrażenie_regularne”, referencja) – sprawdza, czy forma ortograficzna tokenu z referencji spełnia podane wyrażenie regularne. Jeśli tak jest, wynikiem ewaluacji warunku jest fałsz.

Operacje morfosyntaktyczne

- **unify**(lista kategorii gramatycznych, referencja_1, referencja_2, …, referencja_n) – działa analogicznie do *agree*, jednak dodatkowo wykonuje uzgodnienie, pozostawiając tylko te interpretacje, które udało się uzgodnić we wszystkich referencjach. Pozostałe interpretacje oznaczane są jako błędne.
- **persistent_unify**(lista_kategorii_gramatycznych, referencja_1, …, referencja_n) – działa analogicznie do *unify*, jednak uzgodnienie jest stałe. Oznacza to, że uzgodnienie będzie wykonywane ponownie, za każdym razem gdy dowolny z segmentów z referencji się zmieni.
- **delete**(warunek, referencja) – usuwa wszystkie interpretacje tokenu z referencji, które spełniają podany warunek.
- **leave**(warunek, referencja) – odwrotność *delete*. Pozostawiane są tylko te interpretacje tokenu z referencji, które spełniają podany warunek.
- **add**(specyfikacja_tagu, ”forma bazowa”, referencja) – dodaje interpretację (lub wiele interpretacji) określoną w podanej specyfikacji_tagu do tokenu z referencji.
- **set**(specyfikacja_tagu, ”forma bazowa”, referencja) – działa analogicznie do *add*, ale usuwa wszelkie inne interpretacje tokenu z referencji.

Operacje składniowe

- **word**(specyfikacja_tagu, specyfikacji_formy_bazowej) – tworzy nowy wyraz składniowy, o interpretacji wskazanej przez podaną specyfikację_tagu i formie bazowej (podstawowej) takiej jak podana specyfikacja_formy_bazowej.
- **group**(typ_grupy, centrum_składniowe, centrum_semantyczne) – tworzy grupę składniową o typie takim jak podany typ_grupy ze wskazanymi w odnośnikach centrum_składniowym i centrum_semantycznym.

- **join**(typ_grupy, centrum_składniowe, centrum_semantyczne) – działa analogicznie *do group*, ale usuwa grupy, które są objęte działaniem tej operacji.
- **attach**(referencja) – dodaje tokeny spoza referencji (ale te dopasowane w części *Match*) do grupy z podanej referencji.

3.3 Działanie parsera Spejd na przykładach

Przykład 1. Fragment jak w przykładzie (3.1)

Regułą, którą zastosujemy do tekstu wejściowego przytoczonego w przykładzie (3.1) będzie nieco zmodyfikowana „Reguła przykładowa" z punktu 3.2 tej pracy. Mianowicie, dla czytelności, uprościmy tę regułę tak by wykonywana była tylko operacja *unify*. Rezygnujemy więc z operacji *group*, na rzecz czytelności tego przykładu. Nasza nowa reguła ma postać:

Rule „Reguła przykładowa 2"

Match: [pos~"adj"][pos~"subst"];

Eval: unify(case number gender,1,2); #brak grupowania

Wynik działania tej reguły na tekście z przykładu (3.1) został ukazany w przykładzie (3.2).

Widzimy, że nastąpiło – zgodnie z operacją *unify*(case number gender,1,2) – uzgodnienie przypadka, liczby i rodzaju w pierwszym i drugim tokenie wskazanym w części *Match* reguły „Reguła przykładowa 2". Jedyną kombinacją wskazanych kategorii gramatycznych występującą w interpretacjach obu tokenów, była *sg:inst:n2* (liczba pojedyncza, narzędnik, rodzaj nijaki łączący się z liczebnikami głównymi), więc wszystkie inne interpretacje (elementy <lex>) zostały odrzucone poprzez dodanie atrybutu „disamb_sh" równego 0.

```
<tok id="tA5">
        <orth>Przykładowym</orth>
        <lex disamb_sh="0">
                <base>przykładowy</base>
                <ctag>adj:sg:inst:m1:pos</ctag>
        </lex>
        <lex disamb_sh="0">
                <base>przykładowy</base>
                <ctag>adj:sg:inst:n1:pos</ctag>
        </lex>
        <lex>
                <base>przykładowy</base>
                <ctag>adj:sg:inst:n2:pos</ctag>
        </lex>
        <lex disamb_sh="0">
                <base>przykładowy</base>
                <ctag>adj:sg:loc:m1:pos</ctag>
        </lex>
        <lex disamb_sh="0">
                <base>przykładowy</base>
                <ctag>adj:pl:dat:f:pos</ctag>
        </lex>
</tok>
<tok id="tA6">
        <orth>zdaniem</orth>
        <lex>
                <base>zdanie</base>
                <ctag>subst:sg:inst:n2</ctag>
        </lex>
        <lex disamb_sh="0">
                <base>zdać</base>
                <ctag>ger:sg:inst:n2:perf:aff</ctag>
        </lex>
</tok>
```

Przykład (3.2): Wynik działania reguły „Reguła przykładowa 2” na tekst z przykładu (3.1)

Przykład 2. Fragment jak w przykładzie (3.2)

Tym razem wejściem dla naszej reguły będzie tekst ukazany w przykładzie (3.2). Wykorzystana reguła uzupełni poprzednio otrzymany wynik o informację na temat grupy

nominalnej, którą tworzą tokeny „*Przykładowym*” i „*zdaniem*”. Jest to więc druga część operacji opisanych pierwotnie w pierwszej „Regule przykładowej”. Pozostawiając jedynie operację *group* otrzymujemy regułę postaci:

Rule „Reguła przykładowa 3”

Match: [pos~”adj”][pos~”subst”];

Eval: group(NG,2,1); #brak uzgodnienia

Wynik działania tej reguły na tekście z przykładu (3.2) został ukazany w przykładzie (3.3).

Nasz rezultat jest bardzo podobny do wejścia, jakie otrzymała Reguła przykładowa 3, jednakże zwróćmy uwagę na to, że teraz oba tokeny ujęte są – zgodnie z operacją *group*(NG,2,1) – w jedną grupę nominalną. Grupa ta ma typ zgodny z podanym przez operację, czyli “NG”, a jej centrum składniowe i centrum semantyczne są zgodne z referencjami podanymi przez „Regułę przykładową 3”:

- atrybut synh ma wartość “tA6”, co jest identyfikatorem drugiego tokenu,
- atrybut semh ma wartość “tA5”, co jest identyfikatorem pierwszego tokenu.

Oczywiście wynik przedstawiony w przykładzie (3.3) jest tożsamy z wynikiem, jaki otrzymalibyśmy stosując bezpośrednio „Regułę przykładową” (bez żadnych modyfikacji, w postaci takiej jak w punkcie 3.2 tej pracy) do tekstu z przykładu (3.1). Podzielenie tej reguły na dwa etapy zostało przeprowadzone tylko w celu lepszego zobrazowania operacji *unify* i *group*.

```
<group synh=”tA6” semh=”tA5” type=”NG”>
<tok id="tA5">
        <orth>Przykładowym</orth>
        <lex disamb_sh=”0”>
                <base>przykładowy</base>
                <ctag>adj:sg:inst:m1:pos</ctag>
        </lex>
        <lex disamb_sh=”0”>
                <base>przykładowy</base>
                <ctag>adj:sg:inst:n1:pos</ctag>
        </lex>
        <lex>
                <base>przykładowy</base>
                <ctag>adj:sg:inst:n2:pos</ctag>
        </lex>
        <lex disamb_sh=”0”>
                <base>przykładowy</base>
                <ctag>adj:sg:loc:m1:pos</ctag>
        </lex>
        <lex disamb_sh=”0”>
                <base>przykładowy</base>
                <ctag>adj:pl:dat:f:pos</ctag>
        </lex>
</tok>
<tok id="tA6">
        <orth>zdaniem</orth>
        <lex>
                <base>zdanie</base>
                <ctag>subst:sg:inst:n2</ctag>
        </lex>
        <lex disamb_sh=”0”>
                <base>zdać</base>
                <ctag>ger:sg:inst:n2:perf:aff</ctag>
        </lex>
</tok>
</group>
```

Przykład (3.3): Wynik działania reguły „Reguła przykładowa 3” na tekst z przykładu (3.2)

4 Autorskie reguły dla parsera Spejd

W poprzednim rozdziale omówiliśmy składnię reguł parsera Spejd oraz przeanalizowaliśmy przykłady jego działania. W tym rozdziale postaramy się rozbudować jego funkcjonalność poprzez dodanie nowych reguł, które umożliwią konwersję tekstu ortograficznego na tekst fonetyczny. Najpierw omówimy poszczególne reguły, po czym przejdziemy do prezentacji ich działania na przykładach.

4.1 Dodanie nowych reguł konwersji tekstu

Przed dodaniem nowych reguł musimy ustalić, jakiego rodzaju wyrażenia chcemy konwertować. Na potrzebę tej pracy ograniczymy się do wyrażeń następującego rodzaju:

- skróty („*ok. - około*”, „*ur. - urodzony*”, „*tys. - tysiąc*”),
- skrótowce („*PKO – pe ka o*”, „*MPK – em pe ka*”),
- liczby,
    - jednowyrazowe, typu: *1, 2, 10, 14, 60, 200*
    - dwuwyrazowe, typu: *21, 45, 99*
    - trójwyrazowe, typu: *121, 555, 971*
- numery (*numer PESEL*).

Należy zwrócić uwagę na fakt, że jest to jedynie część możliwych do konwersji wyrażeń. Przedstawiamy tu tylko kilka przykładowych reguł, konwertujących wyrażenia różnego typu. Dlatego też nasz system konwersji tekstu nie jest w żadnym stopniu systemem kompletnym. Zarówno liczba samych reguł jak i ich typów musiałaby być znacznie większa, by pokrycie algorytmu było na poziomie akceptowalnym np. w aplikacji komercyjnej. Problem ten zostanie szerzej omówiony w rozdziale 5. w punkcie 5.3 – *Ewaluacja algorytmu*, wrócimy też do niego we wnioskach z niniejszej pracy.

Dodane przez nas reguły są dwojakiego rodzaju. Pierwszy typ reguł, w sekcji *Match,* identyfikuje konwertowane wyrażenia w ich formie ortograficznej. W części *Eval* reguły te opisują nowy wyraz składniowy, o podanej specyfikacji gramatycznej, który powinien zostać wstawiony

przez algorytm. Ponieważ specyfikacja ta, w większości przypadków, będzie niejednoznaczna, potrzebować będziemy jeszcze jednego typu reguł. Zadaniem reguł drugiego typu będzie ujednoznacznianie specyfikacji gramatycznych nowo utworzonych wyrazów składniowych, na podstawie ich sąsiedztwa.

4.1.1 Skróty

Poniższe reguły umożliwiają rozwinięcie przykładowych skrótów:

```
Rule   "ok. @SKR@"
Match: [orth~ok/i] ns [orth~"\."];
Eval:  word(qub, "około");
```

Przykład (4.1): Reguła „około”

```
Rule   "ur. @SKR@"
Match: [orth~ur/i] ns [orth~"\."];
Eval:  word(adj:number*:case*:gender*:pos, "urodzony");
```

Przykład (4.2): Reguła „urodzony”

```
Rule   "tys. @SKR@"
Match: [orth~tys/i] (ns [orth~"\."])?;
Eval:  word(subst:number*:case*:m3, "tysiąc");
```

Przykład (4.3): Reguła „tysiąc”

Powyższe reguły odnajdują skróty „*ok.*” „*ur.*” i „*tys*.” w przetwarzanym tekście. Jeśli dopasowanie zawartości sekcji *Match* powiedzie się, to dodany zostanie nowy wyraz składniowy za pomocą polecenia *word*. Polecenia tego używamy z dwóch powodów. Po pierwsze, pozwala

ono w najprostszy sposób wprowadzić do analizowanego tekstu nowe słowo bazowe z dowolną specyfikacją – jest to dokładnie to, czego potrzebujemy przy przeprowadzanej przez nas konwersji. Po drugie, w wyjściu jakie otrzymamy z parsera Spejd, będzie widoczna informacja, jaka reguła stworzyła dane słowo składniowe. Fakt ten będzie niezwykle istotny dla dalszego przetwarzania wyników parsera, którym zajmiemy się w kolejnym rozdziale.

Reguła z przykładu (4.1) stara się odnaleźć dwa kolejne tokeny, pomiędzy którymi nie ma spacji (*ns – no space*). Pierwszy token ma mieć postać ortograficzną „*ok*”, drugi ma być pojedynczą kropką. Znacznik „*/i*” w opisie części „*ok*” oznacza, że wielkość liter w opisie tego tokenu nie ma znaczenia. Jeśli odnalezione zostaną takie tokeny, to zostanie dodane nowe słowo składniowe o formie bazowej „*około*”, oraz specyfikacji „qub”, która oznacza klasę gramatyczną posiadającą tylko jedną formę gramatyczną (*kublik*).

W przykładzie (4.2) dopasowanie jest analogiczne do dopasowania reguły z przykładu (4.1). Ciekawsza natomiast jest specyfikacja nowego słowa składniowego o formie bazowej „*urodzony*”. Specyfikacja ta określona jest jako „*adj:number*:case*:gender*:pos*”, czyli przymiotnik (*adj*) o dowolnej (*) liczbie (*number*), przypadku (*case*) i rodzaju (*gender*), ale określonym jednoznacznie czwartym atrybucie – stopniu – jako równy (*pos*).

W ostatnim przykładzie (4.3) – token zawierający samą kropkę jest dopuszczalny, ale niekonieczny (dzięki zastosowaniu kwantyfikatora „?”). Dodawany wyraz syntaktyczny o formie bazowej „*tysiąc*” jest rzeczownikiem (*subst*), o dowolnej liczbie i przypadku, z rodzajem ustalonym na *męski rzeczowny* (*m3*).

4.1.2 Skrótowce

Poniższe reguły umożliwiają rozwinięcie przykładowych skrótowców:

```
Rule "PKO @SKRTW@"

Match: [orth~PKO];

Eval:  word(qub, "pe ka o");
```

Przykład (4.4): Reguła „pe ka o”

Rule "MPK @SKRTW@"

Match: [orth~MPK];

Eval: word(qub, "em pe ka");

Przykład (4.5): Reguła „em pe ka”

Reguły z obu przykładów (4.4) i (4.5) dopasowują pojedyncze tokeny, odpowiednich skrótowców, tylko w zapisie wielkimi literami. Tworzony wyraz składniowy jest tak jak w przykładzie (4.1) nieodmienny, a jego forma bazowa to rozwinięty skrótowiec. Odstępy pomiędzy kolejnymi głoskami zapewniają poprawne przeczytanie i zaakcentowanie formy fonetycznej skrótowca przez ewentualny syntezator.

4.1.3 Liczby

Poniższe reguły umożliwiają rozwinięcie przykładowych liczb:

Rule "num @LICZ@"

Match: [orth~"[0-9][0]*"];

Eval: word(num:pl:case*:gender*,1.orth);

Przykład (4.6): Reguła dla 0-9 i 10, ... , 90, 100, ... , 900, 1000, ...

Rule "num1 @LICZ@"

Match: [orth~"[1][1-9]"];

Eval: word(num:pl:case*:gender*,1.orth);

Przykład (4.7): Reguła dla 11-19

```
Rule "num @LICZTWO@"
Match: [orth~"[2-9][1-9]"];
Eval:  word(num:pl:case*:gender*,1.orth);
```

Przykład (4.8): Reguła dla 21-29, 31-39, … , 91-99

```
Rule "num @LICZTHREE@"
Match: [orth~"[1-9][2-9][1-9]"];
Eval:  word(num:pl:case*:gender*,1.orth);
```

Przykład (4.9): Reguła dla 121-129, … , 191-199, 221-229, …

Powyższe reguły konwertują liczby zapisane cyframi arabskimi na ich fonetyczne odpowiedniki. Pierwsze dwa przykłady (4.6) i (4.7) konwertują liczby jednowyrazowe (takie, których zapis fonetyczny składa się z jednego wyrazu), kolejny przykład (4.8) dotyczy liczb dwuwyrazowych, ostatni zaś (4.9) opisuje konwersję liczb trójwyrazowych.

Ponieważ dana reguła zostanie zastosowana dla wielu różnych liczb, nie możemy jeszcze na tym etapie podać konkretnej formy bazowej w poleceniu *word.* Dlatego też używamy konstrukcji „*1.orth*”, która oznacza, że formą bazową ma być forma ortograficzna pierwszego (w naszym wypadku także jedynego) tokenu z reguły. Specyfikacja tego wyrazu składniowego to liczebnik (*num*), w liczbie mnogiej (*pl*), w dowolnym przypadku i rodzaju.

Oczywiście forma tego wyrazu zostanie w dalszym etapie przetworzona na poprawnie odmienioną formę fonetyczną, ale na tym etapie przetwarzania posiadamy wyłącznie informację, ile jest wyrazów opisujących konwertowaną liczbę oraz jakie są jej możliwe formy gramatyczne.

4.1.4 Numery

Poniższa reguła umożliwia rozwinięcie numeru PESEL:

```
Rule "pesel @PESEL@"
Match: [orth~"[0-9]{11}"];
Eval:  word(num:pl:nom:n,1.orth);
```

Przykład (4.10): Reguła PESEL

Ostatnim obsługiwanym przez nas wyrażeniem jest numer PESEL, którego konwersji dokonuje reguła z przykładu (4.10). Dopasowuje ona ciąg jedenastu cyfr i tworzy nowy wyraz składniowy o formie bazowej równej temu ciągowi (analogicznie do poprzednich przykładów). Wyraz ten jest określony jako liczebnik w liczbie mnogiej, w mianowniku (*nom*), rodzaju nijakiego (*n*).

4.1.5 Uzgadnianie form gramatycznych

Poniżej omówiono przykładowe reguły uzgadniania form gramatycznych:

```
Rule "sub+adj"
Match: A[pos~"subst"] B[pos~"adj"];
Eval: unify(case number gender,A,B);
```

Przykład (4.11): Reguła rzeczownik + przymiotnik

```
Rule "num+sub"
Match: A[pos~"num"] B[pos~"subst"];
Eval: unify(case number gender,A,B);
```

Przykład (4.12): Reguła liczebnik + rzeczownik

Rule "num+sub no number"

Match: A[pos~"num"] B[pos~"subst"];

Eval: unify(case gender,A,B);

Przykład (4.13): Reguła liczebnik + rzeczownik bez liczby

Przykłady (4.11), (4.12) i (4.13) ukazują reguły uzgadniające formy gramatyczne pomiędzy sąsiednimi tokenami. Pierwsza z nich dotyczy pary rzeczownik + przymiotnik, a dwie kolejne pary liczebnik + rzeczownik. Wielkie litery „*A*" i „*B*" przy dopasowywanych tokenach ułatwiają referencję w poleceniu *unify*, które uzgadnia kategorie gramatyczne wymienione w pierwszym argumencie pomiędzy tokenami z referencji. Interpretacje nie dające się uzgodnić pomiędzy tymi parami są oznaczane jako błędne. Pierwsze dwie reguły uzgadniają przypadek, liczbę i rodzaj. Ostatnia reguła uzgadnia tylko przypadek i rodzaj. Warto zaznaczyć, że reguła z przykładu (4.13) zadziała tylko dla takiej pary liczebnik + rzeczownik, która posiadać będzie interpretacje o takich samych przypadkach i rodzajach, ale różnych liczbach (przy takich samych liczbach zadziałałaby reguła 4.12).

4.2 Omówienie działania nowych reguł na przykładach

W celu zaprezentowania działania wszystkich reguł przedstawionych powyżej, porównamy fragmenty wyjścia z parsera Spejd bez stosowania opracowanej przez nas reguły (po lewej), oraz po jej zaimplementowaniu (po prawej). Różnice pomiędzy tymi dwoma wersjami będą oznaczane w następujący sposób:

- **informacje które zostały dodane przez regułę**,
- **informacje które zostały usunięte przez regułę**,
- **informacje pominięte w przykładzie**.

W obu przypadkach wejściem dla parsera był czysty tekst zawierający zdania z wyrażeniami do konwersji. Dla zwiększenia czytelności przykładów, przeanalizujemy jedynie te fragmenty wyjścia z parsera Spejd, które dotyczą konwertowanych wyrażeń.

Przykład (4.2). *Reguła „urodzony”*

	<syntok id="ab" rule="ur. @SKR@"> **<orth>ur.</orth>** **<lex><base>urodzony</base>** **<ctag>adj:sg:nom:m1:pos</ctag></lex>** **... //adj:number*:case*:gender*:pos** **<lex><base>urodzony</base>** **<ctag>adj:pl:voc:p3:pos</ctag></lex>**
<tok id="a4" string-range="string-range(p-1,16,2)"> <orth>ur</orth> <lex><base>urodzony</base> <ctag>brev:pun</ctag></lex> </tok> <ns/> <tok id="a6" string-range="string-range(p-1,18,1)"> <orth>.</orth> <lex disamb="1"><base>.</base> <ctag>interp</ctag></lex> </tok>	<tok id="a4" string-range="string-range(p-1,16,2)"> <orth>ur</orth> <lex><base>urodzony</base> <ctag>brev:pun</ctag></lex> </tok> <ns/> <tok id="a6" string-range="string-range(p-1,18,1)"> <orth>.</orth> <lex disamb="1"><base>.</base> <ctag>interp</ctag></lex> </tok> **</syntok>**

Komentarz.

Reguła z przykładu (4.2) w sekcji *Match* dopasowuje dwa kolejne tokeny, o formie ortograficznej kolejno „ur” oraz pojedynczej kropki. W powyższym przykładzie występują takie tokeny, w związku z czym algorytm dodał nowy wyraz składniowy, zgodnie z opisem zawartym w części *Eval* reguły. Nowy token (oznaczony niebieskim kolorem), składa się z formy bazowej „urodzony” (*<lex><base>urodzony</lex></base>*), oraz ze wszystkich możliwych specyfikacji postaci „*adj:number*:case*:gender*:pos*”. Dla czytelności przykładu pozostawione są jedynie pierwsza (*adj:sg:nom:m1:pos*) i ostatnia (*adj:pl:voc:p3:pos*) specyfikacja. Pośrednie specyfikacje zostały pominięte (co oznaczono zielonym kolorem).

Przykład (4.10). *Reguła PESEL*

	<syntok id="a153" rule="pesel @PESEL@"> **<orth>87072300128</orth>** **<lex><base>87072300128</base>** **<ctag>num:pl:nom:n</ctag></lex>**

<tok id="a14f" string-range="string-range(p-1,1209,11)"> <orth>87072300128</orth> <lex><base>87072300128</base> <ctag>ign</ctag></lex> </tok>	<tok id="a14f" string-range="string-range(p-1,1209,11)"> <orth>87072300128</orth> <lex><base>87072300128</base> <ctag>ign</ctag></lex> </tok> **</syntok>**

Komentarz.

W podanym przykładzie występuje ciąg jedenastu cyfr, co odpowiada sekcji *Match* reguły z przykładu (4.10). Algorytm wstawił nowy wyraz składniowy (oznaczono na niebiesko), zgodnie z częścią *Eval* tej samej reguły. Formą bazową tokenu jest dopasowany ciąg (*<lex><base>87072300128</lex></base>*). Token posiada jedną specyfikację gramatyczną (*num:pl:nom:n*).

Przykład (4.11). *Reguła rzeczownik + przymiotnik*

<tok id="a3" string-range="string-range(p-1,6,9)"> <orth>Sosnowski</orth> <lex><base>Sosnowski</base> <ctag>subst:sg:nom:m1</ctag></lex> <lex><base>Sosnowski</base> <ctag>subst:sg:voc:m1</ctag></lex> </tok> <syntok id="ab" rule="ur. @SKR@"> <orth>ur.</orth> <lex><base>urodzony</base> <ctag>adj:sg:nom:m1:pos</ctag></lex> **<lex><base>urodzony</base> <ctag>adj:sg:nom:m2:pos</ctag></lex> <lex><base>urodzony</base> <ctag>adj:sg:nom:m3:pos</ctag></lex> <lex><base>urodzony</base> <ctag>adj:sg:nom:f:pos</ctag></lex> <lex><base>urodzony</base> <ctag>adj:sg:nom:n:pos</ctag></lex>** **... //adj:number*:case*:gender*:pos** **<lex><base>urodzony</base> <ctag>adj:pl:voc:p2:pos</ctag></lex> <lex><base>urodzony</base> <ctag>adj:pl:voc:p3:pos</ctag></lex>**	<tok id="a3" string-range="string-range(p-1,6,9)"> <orth>Sosnowski</orth> <lex><base>Sosnowski</base> <ctag>subst:sg:nom:m1</ctag></lex> <lex><base>Sosnowski</base> <ctag>subst:sg:voc:m1</ctag></lex> </tok> <syntok id="ab" rule="ur. @SKR@"> <orth>ur.</orth> <lex><base>urodzony</base> <ctag>adj:sg:nom:m1:pos</ctag></lex> <lex><base>urodzony</base> <ctag>adj:sg:voc:m1:pos</ctag></lex>

Komentarz.

W przykładzie występują dwa kolejne tokeny, z których pierwszy (*Sosnowski*) zawiera specyfikacje rzeczownikowe, a drugi (*ur.*) posiada specyfikacje przymiotnikowe. Zgodnie z regułą (4.11), kategorie gramatyczne (przypadek, liczba i rodzaj) tych dwóch tokenów zostają uzgodnione. W związku z tym wszystkie specyfikacje nie posiadające odpowiednika w drugim tokenie oznaczone zostają jako błędne. Część z nich została wypisana po lewej stronie (kolorem czerwonym). Reszta nieuzgodnionych specyfikacji została pominięta dla czytelności przykładu (oznaczone kolorem zielonym). Jedynie dwie specyfikacje drugiego tokenu (*adj:sg:nom:m1:pos* oraz *adj:sg:voc:m1:pos*) posiadają odpowiadające im specyfikacje w pierwszym tokenie, w związku z czym tylko te dwie pary zostaną zachowane po zastosowaniu reguły (4.11).

Przykłady działania pozostałych reguł z punktu 4.1 tego rozdziału można znaleźć w Dodatku A niniejszej pracy.

5 Implementacja algorytmu konwersji tekstu

W rozdziałach 1-3 omówiliśmy podstawy teoretyczne, niezbędne do zaprezentowania algorytmu konwersji tekstu. W poprzednim rozdziale, przeanalizowaliśmy reguły, którymi algorytm ten będzie się posługiwać. Teraz możemy przejść do kluczowej części tej pracy, czyli omówienia algorytmu konwersji tekstu ortograficznego na tekst fonetyczny.

5.1 Opis algorytmu

Omawiany algorytm konwersji tekstu można podzielić na dwa podstawowe kroki:

- Parsowanie tekstu
- Synteza wyjścia

Parsowaniem, tak jak wspomniane to było w rozdziale 3, zajmuje się parser Spejd, wzbogacony o reguły podane w punkcie 4.1. Wejściem do parsera może być plik XML zgodny formatem plików morph.xml z Korpusu IPI PAN albo czysty nieoznakowany tekst. Na potrzeby tego rozdziału będziemy korzystać jedynie z tego drugiego trybu, co zwiększy czytelność przykładów w punkcie 5.2, ułatwiając analizę działania algorytmu.

Wynikiem procesu parsowania jest plik XML zawierający tekst przetworzony przez reguły. Przykłady fragmentów takiego pliku analizowaliśmy w punkcie 4.2 tej pracy. Wracając do przykładu (4.2), dotyczącego reguły „urodzony”, zwróćmy uwagę na informacje dodane po zastosowaniu reguły. Utworzyła ona nowy wyraz składniowy o formie ortograficznej „ur.”. Opis tego tokenu, w wyjściowym pliku XML, rozpoczyna się od linii:

<syntok id="ab" rule="ur. @SKR@">

Linia ta oznacza początek wyrazu (tokenu) składniowego, o danym id (*ab*), który został stworzony przez regułę o podanej nazwie (ur. *@SKR@*). Nazwy dodanych reguł mają ustaloną konwencję, która ułatwi drugi krok algorytmu konwersji tekstu – syntezę wyjścia.

Synteza wyjścia ma na celu odpowiednie przetworzenie utworzonego przez parser Spejd pliku XML. Chcemy na wyjściu naszego programu otrzymać czysty tekst, w którym wyrażenia

określonego typu (skróty, skrótowce, liczby, numery), zostaną zamienione na ich fonetyczną wersję. Jako że wykonywane w tym celu czynności będą operacjami na tekście, do syntezy wyjścia posłuży nam skrypt napisany w języku Perl.

Zanim dokładnie wypiszemy kolejne etapy omawianego algorytmu konwersji tekstu, prześledźmy jego działanie na jednym przykładzie. Zdanie „*Zdjęcie chłopca ur. w zeszły czwartek.*” przetworzone przez parser Spejd, rozdzielone zostanie na następujące tokeny w pliku XML:

- „*Zdjęcie*”,
- „c*hłopca*”,
- „*ur*”,
- kropka będąca częścią skrótu „*ur.*”
- „*w*”,
- „*zeszły*”,
- „*czwartek*”,
- kropka kończąca zdanie.

Zwróćmy uwagę na fakt, że wszystkie tokeny poza trzecim i czwartym, nie wymagają konwersji, gdyż ich forma ortograficzna jest tożsama z ich formą fonetyczną. Tokeny tworzące skrót „*ur.*” obsłuży zaś reguła z przykładu (4.2), tworząc nowy wyraz składniowy o formie bazowej „*urodzony*”, o specyfikacji „*adj:number*:case*:gender*:pos*”.

Reguła z przykładu (4.11) dokona uzgodnienia przypadku, liczby i rodzaju pomiędzy tokenem „*chłopca*” i nowo utworzonym wyrazem składniowym. Token „*chłopca*” ma tylko dwie specyfikacje gramatyczne: „*subst:sg:gen:m1*” i „*subst:sg:acc:m1*”. Oznaczają one rzeczownik w liczbie pojedynczej, rodzaju męskiego osobowego, w jednym z dwóch przypadków: dopełniaczu (*gen*) lub bierniku (*acc*). Poprzez regułę z przykładu (4.11), do takich samych parametrów (liczby, przypadka i rodzaju) zostanie ograniczona specyfikacja „*adj:number*:case*:gender*:pos*”. Dla uproszczenia przykładu załóżmy, że jedyną specyfikacją jest ta w dopełniaczu. Wtedy nasz wyraz składniowy reprezentowany byłby przez parę („*urodzony*”, „*adj:sg:gen:m1*”).

Aby przekształcić parę („*urodzony*”, „*adj:sg:gen:m1*”) na poprawnie odmienioną formę,

czyli „*urodzonego*”, potrzebujemy odpowiedniego słownika form fleksyjnych. Ten wykorzystany przez opisywany tu system oparty jest o słownik morfologiczny Morfologik (Miłkowski, 2007). Wpisy w słowniku mają następującą formę:

```
urodzonego  urodzony    adj:sg:acc:m1
urodzonego  urodzony    adj:sg:acc:m2
urodzonego  urodzony    adj:sg:gen:m1
urodzonego  urodzony    adj:sg:gen:m2
urodzonego  urodzony    adj:sg:gen:m3
urodzonego  urodzony    adj:sg:gen:n
```

Przykładowe hasła w słowniku form fleksyjnych

Format każdego hasła jest następujący: odmieniona forma wyrazu, forma bazowa, charakterystyka gramatyczna. Dzięki takiemu słownikowi z łatwością możemy przekształcić parę („*urodzony*”, „*adj:sg:gen:m1*”) na poprawną formę fonetyczną – „*urodzonego*”. Jest to ostatni krok algorytmu i możemy podać na wyjściu zadane na początku zdanie „*Zdjęcie chłopca ur. W zeszły czwartek.*” w formie:

„*Zdjęcie chłopca urodzonego w zeszły czwartek.*”

Przykłady konwersji skrótowców, liczb i numerów zostaną zaprezentowane w punkcie 5.2 niniejszego rozdziału.

Całkowity proces konwersji zdania „*Zdjęcie chłopca ur. w zeszły czwartek.*” ma przebieg taki, jak przedstawia Tabela 5.1.

	Etap	**Przykład**
	Wczytanie zdania wejściowego	„*Zdjęcie chłopca ur. w zeszły czwartek.*”
Speid	Identyfikacja wyrażeń do konwersji	Rule "ur. @SKR@" Match: [orth~ur/i] ns [orth~"\."]; Eval: word(adj:number*:case*:gender*:pos, "urodzony");

Spejd	Uzgodnienie form gramatycznych	Rule "sub+adj" Match: A[pos~"subst"] B[pos~"adj"]; Eval: unify(case number gender,A,B);
Perl	Ekstrakcja tokenów z pliku XML	*Zdjęcie → Zdjęcie* *chłopca → chłopca* *ur. → („urodzony”, „adj:sg:gen:m1”)* ...
Perl	Wybór poprawnej formy konwertowanych wyrażeń	**urodzonego** urodzony adj:sg:gen:m1
	Wygenerowanie zdania wyjściowego	„*Zdjęcie chłopca urodzonego w zeszły czwartek.*”

Tabela (5.1): Etapy przetwarzania zdania „Zdjęcie chłopca ur. w zeszły czwartek.”

5.2 Przykłady działania algorytmu konwersji

Przyjrzymy się teraz kilku innym przykładom działania zaimplementowanego algorytmu konwersji tekstu ortograficznego na fonetyczny. Każdy przykład zobrazujemy najpierw za pomocą tabeli etapów przetwarzania, po czym krótko omówimy.

Przykład (5.1). *Zdanie „Mam konto w PKO.”*

	Etap	**Przykład**
	Wczytanie zdania wejściowego	„*Mam konto w PKO.*”
Spejd	Identyfikacja wyrażeń do konwersji	Rule "PKO @SKRTW@" Match: [orth~PKO]; Eval: word(qub, "pe ka o");
Spejd	Uzgodnienie form gramatycznych	brak

Perl	Ekstrakcja tokenów z pliku XML	*Mam → Mam* *konto → konto* *w → w* *PKO → pe ka o* *. → .*
Perl	Wybór poprawnej formy konwertowanych wyrażeń	brak
	Wygenerowanie zdania wyjściowego	„*Mam konto w pe ka o.*”

Komentarz.

Przykład konwersji skrótowca PKO jest o wiele prostszy od tego omówionego w punkcie 5.1. Skrótowiec PKO konwertowany jest zawsze do tej samej formy fonetycznej – „*pe ka o*”, Nie jest więc konieczne uzgadnianie formy gramatycznej nowo utworzonego wyrazu składniowego czy też wybór właściwej formy konwertowanego wyrażenia. Warto podkreślić, że kropka kończąca zdanie jest samodzielnym tokenem i z punktu widzenia skryptu perlowego jest ona taką samą strukturą jak chociażby token „*konto*”.

Przykład (5.2). *Zdanie „Brakuje mi 23 złotych.”*

	Etap	**Przykład**
	Wczytanie zdania wejściowego	„*Brakuje mi 23 złotych.*”
Spejd	Identyfikacja wyrażeń do konwersji	Rule "num @LICZTWO@" Match: [orth~"[2-9][1-9]"]; Eval: word(num:pl:case*:gender*,1.orth);
Spejd	Uzgodnienie form gramatycznych	Rule "num+sub" Match: A[pos~"num"] B[pos~"subst"]; Eval: unify(case number gender,A,B);
Perl	Ekstrakcja tokenów z pliku XML	*Brakuje → Brakuje* *mi → mi* *23 → („20”, „num:pl:gen:m2”), („3”, „num:pl:gen:m2”)* *złotych → złotych* *.→ .*

Perl	Wybór poprawnej formy konwertowanych wyrażeń	**dwudziestu** 20 num:pl:gen:m2 **trzech** 3 num:pl:gen:m2
	Wygenerowanie zdania wyjściowego	„*Brakuje mi dwudziestu trzech złotych.*”

Komentarz.

Pokazany tu przykład jest nieco bardziej skomplikowany od poprzednich. Formą bazową nowego wyrazu składniowego jest liczba “*23*”. Skrypt perlowy podczas ekstrakcji tokenów z pliku XML, przekształca liczbę 23 na dwie liczby – 20 oraz 3, po czym sprawdza w słowniku ich formy fonetyczne przy zadanej specyfikacji gramatycznej.

Podzielenie liczby 23 na dwie części możliwe jest dzięki wąskiej grupie liczb spełniających regułę *"num @LICZTWO@"*. Liczby takiego typu, czyli spełniające wyrażenie regularne [2-9][1-9], łatwo podzielić na dwie „liczby składowe”, co umożliwia ich bezproblemową konwersję.

Liczby typu *104*, *219*, *630* są także liczbami dwuwyrazowymi, jednak ich podział nie jest jest analogiczny do tego z przykładu (5.2), dlatego też nie są obsługiwane przez regułę *"num @LICZTWO@"*. Chcąc konwertować liczby tego typu musielibyśmy utworzyć nowe reguły dla parsera Spejd, oraz odpowiednio obsłużyć podział tych liczb na części w skrypcie.

Przykład (5.3). Zdanie *„Twój numer PESEL to 88080377223”*

	Etap	**Przykład**
	Wczytanie zdania wejściowego	„*Twój numer PESEL to 88080377223.*”
Spejd	Identyfikacja wyrażeń do konwersji	Rule "pesel @PESEL@" Match: [orth~"[0-9]{11}"]; Eval: word(num:pl:nom:n,1.orth);
Spejd	Uzgodnienie form gramatycznych	brak
Perl	Ekstrakcja tokenów z pliku XML	*Twój* → *Twój* *numer* → *numer* *PESEL* → *PESEL* *to* → *to*

		88080377223 → („80”, „num:pl:nom:n”), („8”, „num:pl:nom:n”), („0”, „num:pl:nom:n”), („8”, „num:pl:nom:n”), („0”, „num:pl:nom:n”), („3”, „num:pl:nom:n”), („70”, „num:pl:nom:n”), („7”, „num:pl:nom:n”), („200”, „num:pl:nom:n”), („20”, „num:pl:nom:n”), („3”, „num:pl:nom:n”) .→ .
P e r l	Wybór poprawnej formy konwertowanych wyrażeń	**osiemdziesiąt** 80 num:pl:nom:n … … ... **trzy** 3 num:pl:nom:n
	Wygenerowanie zdania wyjściowego	„Twój *numer PESEL to osiemdziesiąt osiem zero osiem zero trzy siedemdziesiąt siedem dwieście dwadzieścia trzy .* ”

Komentarz.

Ostatni podany przykład ukazuje konwersję numeru PESEL do formy fonetycznej według wzorca: liczba dwuwyrazowa, liczba dwuwyrazowa, liczba dwuwyrazowa, liczba dwuwyrazowa, liczba trójwyrazowa. Tak jak w przykładzie (5.2), nie wszystkie przypadki numerów PESEL można podzielić w ten sposób.

Przykładowo, numer PESEL rodzaju *80102060400,* który chcielibyśmy przekształcić w ten sam sposób, składałby się z samych liczb jednowyrazowych (*osiemdziesiąt, dziesięć, dwadzieścia, sześćdziesiąt, czterysta*), w związku z czym podział tego numeru na części do konwersji musiałby przebiegać według innej reguły i innych instrukcji skryptu.

5.3 Ewaluacja algorytmu

Podczas omawiania działania algorytmu czy też opisu całego systemu do konwersji tekstu ortograficznego na tekst fonetyczny, wielokrotnie zwracaliśmy uwagę na jego niekompletność. W tym punkcie dokonamy jego oceny ze względu na dwa kryteria – pokrycie i precyzję, po czym przyjrzymy się czasowi działania implementacji algorytmu dla różnych przypadków testowych.

Pokrycie algorytmu – miara określająca stosunek liczby przypadków, które dany algorytm jest w stanie obsłużyć, do wszystkich przypadków danego problemu.

Precyzja algorytmu – miara określająca stosunek liczby poprawnie obsłużonych przypadków, do wszystkich przypadków obsługiwanych przez algorytm.

Oceniając pokrycie opisanego w tej pracy algorytmu, w pierwszej kolejności należałoby zwrócić uwagę na obsługiwane kategorie wyrażeń do konwersji. Kategorie zaprezentowane w tej pracy, czyli skróty, skrótowce, liczby i numery stanowią zaledwie część możliwych do konwersji wyrażeń.

W podobnej pracy, gdzie problem konwersji rozwiązywany jest za pomocą parsingu głębokiego (Wieczorek, 2010), opisana jest także konwersja dla: liczb rzymskich, dat, godzin, znaków specjalnych, adresów elektronicznych oraz wielu przypadków różnych numerów (NIP, REGON, konta bankowe, dowody osobiste).

Konwersje te, można byłoby także przeprowadzić korzystając z przedstawionego w tej pracy systemu konwersji opartego o parsing płytki. Wymagałoby to stworzenia dodatkowych reguł dla parsera Spejd, wzbogacenia słownika o nowe hasła oraz rozbudowania funkcjonalności skryptu dokonującego syntezy wyjścia. Większość z nich przebiegałaby w analogiczny sposób do konwersji numeru PESEL (np. numer REGON, NIP), lub konwersji liczb (liczby rzymskie, daty, godziny), tak więc ograniczenia związane z wykorzystaniem parsingu płytkiego, nie powinny wpływać w znaczący sposób na mnogość obsługiwanych kategorii konwertowanych wyrażeń.

Kolejnym aspektem kompletności opisywanego systemu konwersji jest liczba obsłużonych wyrażeń w ramach danej kategorii, czyli liczba różnych skrótów, skrótowców, liczb i numerów, które system ten jest w stanie przekształcić do formy fonetycznej.

W pracy tej podaliśmy tylko kilka przykładowych reguł, by zobrazować sposób działania systemu. Jednakże w ten sam sposób moglibyśmy konwertować znacznie więcej wyrażeń spośród obsługiwanych przez system kategorii. Przykładowo, większość reguł dla skrótów czy skrótowców nie różni się znacznie od tych pokazanych w rozdziale 4. Liczby o innym formacie niż te z przykładów (4.6) – (4.8), wymagałyby tylko dodatkowych, analogicznych reguł i odpowiedniego podziału bazowej liczby na części w skrypcie perlowym.

W związku z powyższymi uwagami możemy stwierdzić, że rozbudowa opisywanego systemu konwersji tekstu, w celu zwiększenia pokrycia, jest jak najbardziej możliwa.

Omówmy teraz drugie powszechnie stosowane kryterium ewaluowanych systemów, czyli precyzję. Cechy parsingu płytkiego, w szczególności częściowy charakter jego analizy, sprawiają, że posiadana przez nas wiedza o przetwarzanym tekście jest niekompletna. Problem ten znacznie obniża precyzję systemów opartych na parsingu płytkim, co prezentuje poniższy przykład działania systemu opisanego w tej pracy:

Nie mam 20 książek. → Nie mam dwudziestu książek.

*Mam 20 książek. → Mam **dwudziestu** książek .*

Powyższy problem jest konsekwencją uzgadniania form gramatycznych konwertowanych wyrażeń z najbliższymi sąsiadami. Z tego powodu, liczba *20* w obu zdaniach przyjmuje przypadek od tokenu „*książek*" (dopełniacz), co jest błędną formą dla drugiego zdania. Poprawne rozwiązanie dla obu tych zdań wymagałoby reguł opartych na pełnej analizie przetwarzanego tekstu, a więc użycia parsingu głębokiego.

Przyjrzyjmy się teraz czasowi działania implementacji omawianego algorytmu. Opracowany system konwersji został przetestowany na czterech testowych zbiorach zdań. Zbiory te zawierały kolejno 2500, 5000, 10000 i 20000 zdań i zostały wygenerowane automatycznie z kombinacji 37 podstawowych zdań testujących każdą utworzoną regułę. Test przeprowadzony został w celu zbadania dwóch czynników:

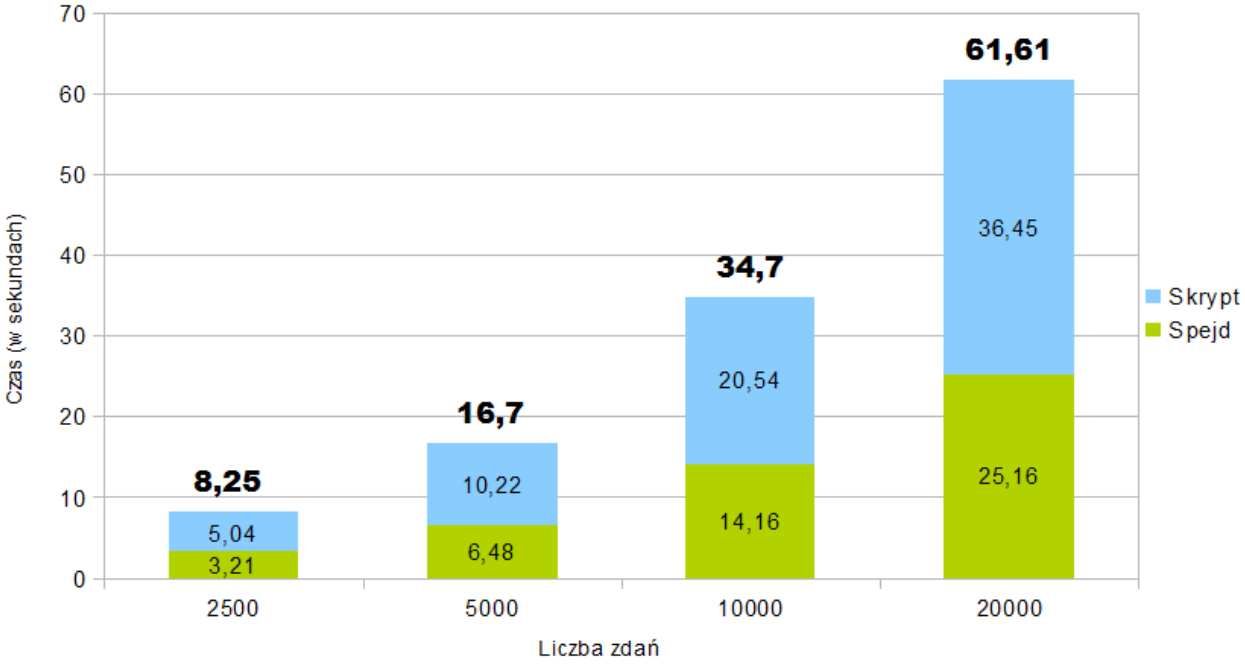
- zależności pomiędzy liczbą przetwarzanych zdań, a czasem działania systemu,
- czasowego udziału parsera Spejd i skryptu perlowego w procesie przetwarzania zdań.

Średnia liczba wyrażeń wymagających konwersji na jedno zdanie wyniosła 1,19.

Na wykresie (5.1) znajdują się wyniki przeprowadzanych testów. Kolorem zielonym oznaczony jest czas działania parsera Spejd, a kolorem niebieskim czas działania skryptu perlowego. Pogrubioną czcionką zaznaczony jest całkowity czas przetwarzania testowych zbiorów zdań.

Widać liniową zależność pomiędzy liczbą przetwarzanych zdań, a czasem działania systemu. W przybliżeniu, za każdym razem gdy liczba zdań wzrosła dwukrotnie, podwoił się też czas działania implementacji algorytmu.

Czas działania zarówno parsera Spejd jak i skryptu perlowego wzrastał równomiernie z przyrostem liczby przetwarzanych zdań. Dzięki przeprowadzonym testom, możemy określić czasowy udział parsera Spejd w całości procesu przetwarzania na poziomie około 40% (i odpowiednio 60% dla skryptu perlowego).



Wykres (5.1): Czas przetwarzania zdań przez implementację algorytmu

Wnioski

Praca ta miała na celu opis zagadnienia jakim jest konwersja tekstu, zwrócenie uwagi na problemy z nim związane i próbę zaimplementowania algorytmu, który opierając się o parsing płytki, automatyzowałby ten proces.

Przyjęte podejście do konwersji tekstu, a w szczególności wykorzystanie parsingu płytkiego, okazało się być wystarczające w większości omawianych przypadków. Przy użyciu parsera SPEJD wzbogaconego o nowo utworzone, autorskie reguły, udało się obsłużyć takie wyrażenia jak skróty, skrótowce, liczby i numery. Zaprezentowane rozwiązanie może też być z łatwością poszerzone o obsługę wyrażeń innego typu, za pomocą kolejnych reguł.

Podczas analizy wyników, zwróciliśmy jednak uwagę na fakt, że parsing częściowy może w niektórych przypadkach być niewystarczający do przeprowadzenia poprawnej konwersji tekstu z formy ortograficznej na fonetyczną. Warto byłoby zbadać, czy uproszczenie i przyspieszenie procesu konwersji tekstu rekompensuje spadek jego jakości. Należy rozważyć rozwiązanie dokonujące warunkowo pełnej analizy składniowej tych zdań, dla których parsing płytki nie daje oczekiwanych wyników.

Bibliografia

1. Przepiórkowski, „*Powierzchniowe przetwarzanie języka polskiego*”, 2008
2. Graliński, Jassem, Wagner, Wypych, „*Text Normalization as a Special Case of Machine Translation*”, 2006
3. Masternak, Łaczynski, „*Parsery wykorzystywane w analizie jezyka naturalnego*”
4. Maziarz, Radziszewski, „*Opis morfo-syntaktyczny liczebników zbiorowych oraz zaimków osobowych na potrzeby konwersji danych słownika Morfologik do tagsetu KIPI*”, 2011
5. Schäfer, „*Integrating Deep and Shallow Natural Language Processing Components – Representations and Hybrid Architectures*”, 2006
6. Swift, Allen, Gildea, „*Skeletons in the parser: Using a shallow parser to improve deep parsing*”, 2004
7. Sutradhar, „*Parsing in Information Extraction and Retrieval*”
8. Rahman, Das, Sharma, „*Parsing of part-of-speech tagged Assamese Texts*”, 2009
9. Donaldson, „*Natural Language Processing: Statistical parsing*”, 2011
10. Kaplan, Riezler, King, Maxwell, Vasserman, Crouch, „*Speed and accuracy in shallow and deep stochastic parsing*”, 2004
11. Caseiro, „*Weighted Finite State Transducers Applied to Spoken Language Processing*”, 2004

Dodatek A. Przykłady działania utworzonych reguł

W punkcie 4.2 tej pracy znalazły się tylko trzy przykłady działania utworzonych reguł. W Dodatku A. zostały umieszczone pozostałe przykłady działania reguł z punku 4.1.

Przykład (4.1). *Reguła „około”*

```
                                        <syntok id="a72" rule="ok. @SKR@">
                                        <orth>ok.</orth>
                                        <lex><base>około</base>
                                        <ctag>qub</ctag></lex>

<tok id="a6f" string-range="string-     <tok id="a6f" string-range="string-
range(p-1,392,2)">                      range(p-1,392,2)">
<orth>ok</orth>                         <orth>ok</orth>
<lex><base>OK</base>                    <lex><base>OK</base>
<ctag>qub</ctag></lex>                  <ctag>qub</ctag></lex>
<lex><base>oko</base>                   <lex><base>oko</base>
<ctag>subst:pl:gen:n2</ctag></lex>      <ctag>subst:pl:gen:n2</ctag></lex>
<lex><base>około</base>                 <lex><base>około</base>
<ctag>brev:pun</ctag></lex>             <ctag>brev:pun</ctag></lex>
</tok>                                  </tok>
<ns/>                                   <ns/>
<tok id="a71" string-range="string-     <tok id="a71" string-range="string-
range(p-1,394,1)">                      range(p-1,394,1)">
<orth>.</orth>                          <orth>.</orth>
<lex disamb="1"><base>.</base>          <lex disamb="1"><base>.</base>
<ctag>interp</ctag></lex>               <ctag>interp</ctag></lex>
</tok>                                  </tok>

                                        </syntok>
```

Przykład (4.3). *Reguła „tysiąc”*

```
                                        <syntok id="a7d" rule="tys. @SKR@">
                                        <orth>tys.</orth>
                                        <lex><base>tysiąc</base>
                                        <ctag>subst:sg:nom:m3</ctag></lex>
                                        ... // subst:number*:case*:m3
                                        <lex><base>tysiąc</base>
                                        <ctag>subst:pl:voc:m3</ctag></lex>

<tok id="a7a" string-range="string-     <tok id="a7a" string-range="string-
range(p-1,424,3)">                      range(p-1,424,3)">
<orth>tys</orth>                        <orth>tys</orth>
<lex><base>tys</base>                   <lex><base>tys</base>
<ctag>ign</ctag></lex>                  <ctag>ign</ctag></lex>
</tok>                                  </tok>
```

```
<ns/>
<tok id="a7c" string-range="string-
range(p-1,427,1)">
<orth>.</orth>
<lex disamb="1"><base>.</base>
<ctag>interp</ctag></lex>
</tok>
```

```
<ns/>
<tok id="a7c" string-range="string-
range(p-1,427,1)">
<orth>.</orth>
<lex disamb="1"><base>.</base>
<ctag>interp</ctag></lex>
</tok>


</syntok>
```

Przykład (4.4). *Reguła „pe ka o”*

```
<tok id="ab1" string-range="string-
range(p-1,602,3)">
<orth>PKO</orth>
<lex><base>PKO</base>
<ctag>subst:sg:nom:f</ctag></lex>
... // subst:number*:case*:f.n2
<lex><base>PKO</base>
<ctag>subst:pl:voc:n2</ctag></lex>
</tok>
```

```
<syntok id="ab4" rule="PKO @SKRTW@">
<orth>PKO</orth>
<lex><base>pe ka o</base>
<ctag>qub</ctag></lex>

<tok id="ab1" string-range="string-
range(p-1,602,3)">
<orth>PKO</orth>
<lex><base>PKO</base>
<ctag>subst:sg:nom:f</ctag></lex>
... // subst:number*:case*:f.n2
<lex><base>PKO</base>
<ctag>subst:pl:voc:n2</ctag></lex>
</tok>


</syntok>
```

Przykład (4.5). *Reguła „em pe ka”*

```
<tok id="ac1" string-range="string-
range(p-1,674,3)">
<orth>MPK</orth>
<lex><base>MPK</base>
<ctag>subst:sg:nom:n2</ctag></lex>
... // subst:number*:case*:n2
<lex><base>MPK</base>
<ctag>subst:pl:voc:n2</ctag></lex>
</tok>
```

```
<syntok id="ac8" rule="MPK @SKRTW@">
<orth>MPK</orth>
<lex><base>em pe ka</base>
<ctag>qub</ctag></lex>

<tok id="ac1" string-range="string-
range(p-1,674,3)">
<orth>MPK</orth>
<lex><base>MPK</base>
<ctag>subst:sg:nom:n2</ctag></lex>
... // subst:number*:case*:n2
<lex><base>MPK</base>
<ctag>subst:pl:voc:n2</ctag></lex>
</tok>


</syntok>
```

Przykład (4.6). *Reguła dla 0-9 i 10, … , 90, 100, … , 900, 1000, …*

	<syntok id="ad6" rule="num @LICZ@"> **<orth>3</orth>** **<lex><base>3</base>** **<ctag>num:pl:nom:m1</ctag></lex>** **... // num:pl:case*:gender*** **<lex><base>3</base>** **<ctag>num:pl:voc:p3</ctag></lex>**
<tok id="ad1" string-range="string-range(p-1,748,1)"> <orth>3</orth> <lex><base>3</base> <ctag>ign</ctag></lex> </tok>	<tok id="ad1" string-range="string-range(p-1,748,1)"> <orth>3</orth> <lex><base>3</base> <ctag>ign</ctag></lex> </tok> **</syntok>**

Przykład (4.7). *Reguła dla 11-19*

	<syntok id="afc" rule="num1 @LICZ@"> **<orth>11</orth>** **<lex><base>11</base>** **<ctag>num:pl:nom:m1</ctag></lex>** **... // num:pl:case*:gender*** **<lex><base>11</base>** **<ctag>num:pl:voc:p3</ctag></lex>**
<tok id="af8" string-range="string-range(p-1,881,2)"> <orth>11</orth> <lex><base>11</base> <ctag>ign</ctag></lex> </tok>	<tok id="af8" string-range="string-range(p-1,881,2)"> <orth>11</orth> <lex><base>11</base> <ctag>ign</ctag></lex> </tok> **</syntok>**

Przykład (4.8). *Reguła dla 21-29, 31-39, ... , 91-99*

	<syntok id="a10a" rule="num @LICZTWO@"> **<orth>23</orth>** **<lex><base>23</base>** **<ctag>num:pl:nom:m1</ctag></lex>** **... // num:pl:case*:gender*** **<lex><base>23</base>** **<ctag>num:pl:voc:p3</ctag></lex>**
<tok id="a106" string-range="string-range(p-1,938,2)"> <orth>23</orth> <lex><base>23</base> <ctag>ign</ctag></lex> </tok>	<tok id="a106" string-range="string-range(p-1,938,2)"> <orth>23</orth> <lex><base>23</base> <ctag>ign</ctag></lex> </tok> **</syntok>**

Przykład (4.9). *Reguła dla 121-129, ... , 191-199, 221-229, ...*

	<syntok id="a134" rule="num @LICZTHREE@"> **<orth>223</orth>** **<lex><base>223</base>** **<ctag>num:pl:nom:m1</ctag></lex>** **... // num:pl:case*:gender*** **<lex><base>223</base>** **<ctag>num:pl:voc:p3</ctag></lex>**
<tok id="a130" string-range="string-range(p-1,1086,3)"> <orth>223</orth> <lex><base>223</base> <ctag>ign</ctag></lex> </tok>	<tok id="a130" string-range="string-range(p-1,1086,3)"> <orth>223</orth> <lex><base>223</base> <ctag>ign</ctag></lex> </tok> **</syntok>**

Przykład (4.12). *Reguła liczebnik + rzeczownik*

```
<syntok id="a112" rule="num @LICZTWO@">
<orth>23</orth>
<lex><base>23</base>
<ctag>num:pl:nom:m1</ctag></lex>
<lex><base>23</base>
<ctag>num:pl:nom:m2</ctag></lex>
<lex><base>23</base>
<ctag>num:pl:nom:m3</ctag></lex>
<lex><base>23</base>
<ctag>num:pl:nom:f</ctag></lex>
... //num:pl:case*:gender*
<lex><base>23</base>
<ctag>num:pl:voc:n3</ctag></lex>
<lex><base>23</base>
<ctag>num:pl:voc:p1</ctag></lex>
<lex><base>23</base>
<ctag>num:pl:voc:p2</ctag></lex>
<lex><base>23</base>
<ctag>num:pl:voc:p3</ctag></lex>
```

```
<tok id="a10e" string-range="string-
range(p-1,967,2)">
<orth>23</orth>
<lex><base>23</base>
<ctag>ign</ctag></lex>
</tok>
</syntok>
<tok id="a10f" string-range="string-
range(p-1,970,11)">
<orth>składnikami</orth>
<lex><base>składnik</base>
<ctag>subst:pl:inst:m3</ctag></lex>
</tok>
```

```
<syntok id="a112" rule="num @LICZTWO@">
<orth>23</orth>
<lex><base>23</base>
<ctag>num:pl:inst:m3</ctag></lex>
```

```
<tok id="a10e" string-range="string-
range(p-1,967,2)">
<orth>23</orth>
<lex><base>23</base>
<ctag>ign</ctag></lex>
</tok>
</syntok>
<tok id="a10f" string-range="string-
range(p-1,970,11)">
<orth>składnikami</orth>
<lex><base>składnik</base>
<ctag>subst:pl:inst:m3</ctag></lex>
</tok>
```

Dodatek B. Dokumentacja projektu magisterskiego

Dodatek B. przedstawia dokumentację projektu magisterskiego będącego podstawą niniejszej pracy.

Dokumentacja projektu magisterskiego

Tytuł projektu

Program konwertujący tekst ortograficzny na tekst fonetyczny.

Początek

Zwięźle opisz wizję na swój projekt.

Istotą mojego projektu jest zaprojektowanie i implementacja algorytmu przetwarzającego tekst ortograficzny na fonetyczny. Algorytm ten otrzymywać będzie na wejściu tekst w języku polskim. Na wyjściu algorytm będzie miał zwracać zdania z tekstu źródłowego w ich zapisie fonetycznym.

Pośrednim narzędziem w implementacji tego procesu będzie parser Spejd. Dla tego parsera zostaną stworzone dodatkowe reguły, w oparciu o które będzie on dokonywał płytkiej analizy morfosyntaktycznej.
Reguły te będą dwojakiego rodzaju:

- reguły identyfikujące „specjalne” tokeny w zdaniach, do późniejszego przekształcenia,
- reguły umożliwiające unifikowanie form gramatycznych w tekście.

Program wykorzysta analizę przeprowadzoną przez parser Spejd do odpowiedniego przekształcenia podanego mu tekstu. Przykładowe operacje programu:

- rozwijanie skrótów / skrótowców,
- zapisywanie liczb / dat w formie fonetycznej,
- przetwarzanie numerów PESEL / NIP.

Opisz zwięźle uzasadnienie istnienia projektu. Jakie problemy zostaną rozwiązane?

Głównym powodem powstania projektu jest zbadanie możliwości konwersji tekstu przy wykorzystaniu parsingu płytkiego. Podobne zadania zostały już zrealizowane wcześniej, ale używały one parsingu głębokiego. Takie podejście zapewniało lepszą analizę przetwarzanego tekstu, jednak za cenę wydłużenia tego procesu w czasie.

Konsekwencją użycia parsingu płytkiego będzie mniej dogłębna analiza zdań otrzymanych na wejściu, przez co niektóre przekształcenia mogą być błędne, co z kolei zmniejszy precyzję implementowanego algorytmu.
Projekt ten jest więc próbą takiej optymalizacji konwersji tekstu, by przebiegała ona jak najszybciej, przy możliwie jak najmniejszym obniżeniu precyzji.

Zdefiniuj główne cele projektu.

- zaprojektowanie algorytmu konwersji tekstu
- zaimplementowanie algorytmu konwersji tekstu
- stworzenie reguł dla parsera Spejd
- przetestowanie napisanego programu – ze względu na czas działania
- przetestowanie napisanego programu – ze względu na precyzję działania

Jakie rezultaty są kluczowe dla projektu?

- możliwie jak najszybsze działanie programu implementującego algorytm konwersji tekstu
- możliwie jak najwyższa precyzja przekształceń dokonywanych na przetwarzanym tekście
- poprawne działanie programu dla najpowszechniejszych skrótów / skrótowców
- poprawne działanie programu dla prostych liczb / numerów

Przedstaw członków podstawowego zespołu oraz głównych zainteresowanych (interesariuszy).

Zespół: Adam Sosnowski
Główni zainteresowani: prof. UAM dr hab. Krzysztof Jassem

Jakich zasobów wymaga projekt?

- Parser Spejd
- słownik morfologiczny (np. Morfologik)
- słownik skrótów i skrótowców

Planowanie

Opisz wymagania funkcjonalne projektu wraz z opisem zakresu projektu.

Wymaganie	**Analiza morfosyntaktyczna zdań**
Opis	Tekst znajdujący się w pliku źródłowym jest dzielony na zdania i przetwarzany z wykorzystaniem parsera Spejd. Sparsowany tekst zapisywany jest w pliku .xml
Dane wejściowe	Tekst w pliku źródłowym
Dane wyjściowe	Plik .xml z opisem gramatycznym zdań z pliku wejściowego, oraz tagami wskazującymi na wyrażenia do konwersji (skróty / skrótowce / liczby / numery)
Warunek początkowy	Tekst w języku polskim, bez formatowania
Stan końcowy	Tworzony jest plik .xml z przeanalizowanym tekstem

Wymaganie	**Konwersja skrótów**
Opis	Plik .xml jest parsowany, a tokeny oznaczone jako skróty, są konwertowane do odpowiedniej formy fonetycznej za pomocą słownika Morfologik
Dane wejściowe	Plik .xml z opisem gramatycznym zdań, oraz tagami wskazującymi na skróty
Dane wyjściowe	Tekst z rozwiniętymi skrótami
Warunek początkowy	Plik .xml zawiera tagi wskazujące na skróty i ich opis w postaci par: skrót w formie podstawowej, opis gramatyczny docelowej formy), słownik Morfologik zawiera wpisy dotyczące odpowiednich skrótów
Stan końcowy	Tworzony jest plik tekstowy ze zdaniami po konwersji. Kolejne zdania odpowiadają tym sprzed analizy morfosyntaktycznej, ale skróty, które w nich występowały, rozwinięte są do odpowiedniej formy fonetycznej

Wymaganie	**Konwersja skrótowców**
Opis	Plik .xml jest parsowany, a tokeny oznaczone jako skrótowce, są konwertowane do odpowiedniej formy fonetycznej za pomocą słownika Morfologik
Dane wejściowe	Plik .xml z opisem gramatycznym zdań, oraz tagami wskazującymi na skrótowce
Dane wyjściowe	Tekst z rozwiniętymi skrótowcami
Warunek początkowy	Plik .xml zawiera tagi wskazujące na skrótowce i ich opis w postaci par: skrótowiec w formie podstawowej, opis gramatyczny docelowej formy), słownik Morfologik zawiera wpisy dotyczące wyrazów wchodzących w skład odpowiednich skrótowców
Stan końcowy	Tworzony jest plik tekstowy ze zdaniami po konwersji. Kolejne zdania odpowiadają tym sprzed analizy morfosyntaktycznej, ale skrótowce, które w nich występowały, rozwinięte są do odpowiedniej formy fonetycznej

Wymaganie	**Konwersja liczb**
Opis	Plik .xml jest parsowany, a tokeny oznaczone jako liczby, są konwertowane do odpowiedniej formy fonetycznej za pomocą słownika Morfologik
Dane wejściowe	Plik .xml z opisem gramatycznym zdań, oraz tagami wskazującymi na liczby
Dane wyjściowe	Tekst z liczbami zapisanymi w formie fonetycznej
Warunek początkowy	Plik .xml zawiera tagi wskazujące na liczby i ich opis w postaci par: liczba w formie podstawowej, opis gramatyczny docelowej formy), słownik Morfologik zawiera wpisy dotyczące wyrazów składających się na odpowiednią liczbę w jej fonetycznej formie
Stan końcowy	Tworzony jest plik tekstowy ze zdaniami po konwersji. Kolejne zdania odpowiadają tym sprzed analizy morfosyntaktycznej, ale liczby, które w nich występowały, rozwinięte są do odpowiedniej formy fonetycznej

Wymaganie	**Konwersja dat**
Opis	Plik .xml jest parsowany, a tokeny oznaczone jako daty, są konwertowane do odpowiedniej formy fonetycznej za pomocą słownika Morfologik
Dane wejściowe	Plik .xml z opisem gramatycznym zdań, oraz tagami wskazującymi na daty
Dane wyjściowe	Tekst z datami zapisanymi w formie fonetycznej
Warunek początkowy	Plik .xml zawiera tagi wskazujące na daty i ich opis w postaci par: data w formie podstawowej, opis gramatyczny docelowej formy), słownik Morfologik zawiera wpisy dotyczące wyrazów składających się na odpowiednią datę w jej fonetycznej formie

Stan końcowy	Tworzony jest plik tekstowy ze zdaniami po konwersji. Kolejne zdania odpowiadają tym sprzed analizy morfosyntaktycznej, ale daty, które w nich występowały, rozwinięte są do odpowiedniej formy fonetycznej

Wymaganie	**Konwersja numerów (PESEL, NIP)**
Opis	Plik .xml jest parsowany, a tokeny oznaczone jako numery, są konwertowane do odpowiedniej formy fonetycznej za pomocą słownika Morfologik
Dane wejściowe	Plik .xml z opisem gramatycznym zdań, oraz tagami wskazującymi na numery
Dane wyjściowe	Tekst z numerami zapisanymi w formie fonetycznej
Warunek początkowy	Plik .xml zawiera tagi wskazujące na numery typu PESEL czy NIP i ich opis w postaci par: numer w formie podstawowej, opis gramatyczny docelowej formy), słownik Morfologik zawiera wpisy dotyczące wyrazów składających się na odpowiedni numer w jego fonetycznej formie
Stan końcowy	Tworzony jest plik tekstowy ze zdaniami po konwersji. Kolejne zdania odpowiadają tym sprzed analizy morfosyntaktycznej, ale numery typu PESEL czy NIP, które w nich występowały, rozwinięte są do odpowiedniej formy fonetycznej

Zakres projektu:

- algorytm konwersji tekstu
- program implementujący algorytm konwersji tekstu
- reguły parsera Spejd

Wymień zagrożenia dla powodzenia projektu oraz sposoby ich zminimalizowania?

Zagrożenie dla powodzenia projektu	**Sposób zminimalizowania**	**Prawdopodobieństwo wystąpienia**	**Wpływ na powodzenie projektu**
nieukończenie projektu w terminie	dobra organizacja pracy nad projektem	wysokie	wysoki
trudności w tworzeniu reguł parsera Spejd	korzystanie z literatury, ograniczenie funkcjonalności projektu	średnie	wysoki
trudności w tworzeniu reguł zgodnych z gramatyką języka polskiego	ograniczenie funkcjonalności projektu	wysokie	niski

niska precyzja w działaniu programu	staranna analiza reguł	średnie	niski
niewystarczająca szybkość działania programu	ograniczenie funkcjonalności projektu i optymalizacja programu pod tym kątem	średnie	średni

Dołącz prosty plan prac wykonywanych na osi czasu

Poprawienie błędów w obecnej wersji		Połączenie skryptów w jedną całość		Dodanie obsługi skrótowców		Dodanie obsługi liczb		Dodanie obsługi numerów		Poprawki / testy precyzji i szybkości	
	22.03.12		24.03.12		27.04.12		03.04.12		10.04.12		17.04.12

Wykonanie

Napisz krótki opis z każdego przeglądu stanu projektu.

Przegląd 1: **22.03.2012**

Program realizuje swoje podstawowe zadanie. Tekst podawany na wejściu jest przetwarzany przez parser Spejd. Tworzony jest plik .xml z opisem gramatycznym zdań z tekstu źródłowego. Zaznaczone są tam też wyrażenia, które wymagają dalszej konwersji. W obecnej wersji są to tylko skróty, w kolejnych będą to także skrótowce, liczby, daty, numery typu PESEL. By umożliwić odpowiednią odmianę skrótów, do parsera Spejd dodane zostały proste reguły uzgadniające formę gramatyczną skrótów na podstawie sąsiednich tokenów.

Plik .xml jest podawany jako wejście do skryptu perlowego. Ten przy pomocy małego słownika (podzbioru słownika Morfologik), potrafi zwrócić na wyjściu odpowiednio odmienione rozwinięcie oznaczonego w pliku .xml skrótu.

Jak wspomniano wcześniej, program w obecnej wersji nie rozpoznaje i nie odmienia skrótowców, liczb, dat i numerów. Słownik jest ubogi, zawiera tylko kilka skrótów. Reguły uzgadniające formy gramatyczne konwertowanych wyrażeń nie są wystarczająco przetestowane, jest ich też znacząco za mało.

Przegląd 2: **15.04.2012**

Dodane zostało rozpoznawanie oraz konwersja skrótowców oraz prostych liczb. Stworzone zostały reguły do parsera Spejd rozpoznające bardziej złożone liczby, jednakże prace nad ich poprawną konwersją cały czas trwają. Ich ukończenie umożliwi implementację konwersji dat oraz numerów.
Słownik został wzbogacony o wpisy niezbędne do konwersji przykładowych skrótowców i liczb. Dodano również reguły uzgadniające formy gramatyczne konwertowanych wyrażeń, w szczególności reguły uzgadniające formy liczebnika (niezbędne przy konwersji liczb, dat, numerów).

Zamykanie

Czy użytkownicy i promotor są usatysfakcjonowani? Jeśli nie, wyjaśnij dlaczego.

Po prezentacji ostatecznej wersji projektu stwierdzam, że udało mi się zrealizować założenia mojego projektu. Program konwertuje różne skróty, skrótowce, liczby i numery. Przedstawiony przeze mnie program został pozytywnie oceniony podczas prezentacji. Jedynym aspektem projektu, który należałoby rozbudować, jest poszerzenie zakresu obsługiwanych wyrażeń. Jednak dzięki przyjętemu podejściu opartemu o reguły parsera Spejd i prosty skrypt w języku Perl, taka rozbudowa projektu nie powinna być trudna.

Wymień, czego się nauczyłeś podczas tworzenia projektu.

Podczas tworzenia projektu nauczyłem się:

- obsługi parsera Spejd
- tworzenia własnych reguł parsera Spejd
- podstawowych zasad składni języka polskiego
- podstawowych zasad fleksji języka polskiego
- wielu różnych podejść do przetwarzania tekstu

Oprócz tego poszerzyłem swoją wiedzę dotyczącą języka perl, w szczególności z zakresu przetwarzania tekstu. Poznałem wady i zalety przetwarzania płytkiego, a także różnice pomiędzy przetwarzaniem płytkim a głębokim.

Czy zostało już wszystko ukończone? Jeśli nie, wyjaśnij to.

Podstawowe funkcjonalności programu zostały ukończone i działają zgodnie z oczekiwaniami. Jak wspomniano powyżej, projekt można z łatwością rozbudować o nowe obsługiwane wyrażenia. Do obsługiwanych obecnie wyrażeń (skróty, skrótowce, liczby, numery), można byłoby dodać kolejne (liczby rzymskie, daty, godziny, znaki specjalne).