Uniwersytet im. A. Mickiewicza - Wydział Matematyki i Informatyki

Leszek Manicki
nr albumu: 287533

Płytki parsing języka francuskiego

Praca magisterska na kierunku:
INFORMATYKA
Promotor:
dr hab. Krzysztof Jassem

Poznań 2009

Oświadczenie

Poznań, dnia .....

Ja, niżej podpisany **Leszek Manicki** student Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt:

Płytki parsing języka francuskiego

napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem tej rozprawy lub jej części od innych osób.

Oświadczam również, że egzemplarz pracy dyplomowej w formie wydruku komputerowego jest zgodny z egzemplarzem pracy dyplomowej w formie elektronicznej.

Jednocześnie przyjmuję do wiadomości, że gdyby powy.zsze oświadczenie okazało się nieprawdziwe, decyzja o wydaniu mi dyplomu zostanie cofnięta.

.....

Streszczenie

W niniejszej pracy zajmiemy się płytkim parsingiem języka francuskiego. Przedstawimy płytki parser *puddle* wykorzystujący formalizm reguł oparty na wyrażeniach regularnych. Zaprezentujemy rozwiązanie, na którym oparty jest parser *puddle*, a także przedstawimy zmiany i różnice w stosunku do tego rozwiązania. Zaprezentujemy algorytm stosowany przez parser *puddle* oraz szczegóły związane z implementacją. Dokonamy ewaluacji wyników osiąganych przy użyciu parsera *puddle* poprzez porównanie wyników parsera z danymi z banku drzew języka francuskiego.

Słowa kluczowe

parsing, płytki parsing, analiza składniowa, język francuski

Spis treści

Wprowadzenie

Podstawą współczesnego językoznawstwa jest teoria strukturalizmu. Za jej twórcę uchodzi szwajcarski językoznawca Ferdinand de Saussure. Jego koncepcje sformułowane zostały m. in. w dziele „Kurs językoznawstwa ogólnego”[1], które zostało opracowane przez dwóch byłych studentów de Saussure'a, Charlesa Bally i Alberta Sechehaye, na podstawie wykładów prowadzonych przez niego na uniwersytecie w Genewie i wydane w 1916 roku, już po śmierci de Saussure'a.

Teorie de Saussure'a rozwijane były przez wielu badaczy. Szczególnie intensywny rozwój w pierwszej połowie XX wieku zaowocował powstaniem wielu nowych koncepcji (za najbardziej istotne uważane są koncepcje rozwijane w szkołach praskiej, kopenhaskiej, amerykańskiej oraz genewskiej). Także współczesne teorie lingwistyczne szeroko korzystają z dorobku de Saussure'a. We wszystkich koncepcjach wywodzących się ze strukturalizmu zauważalna jest wspólna cecha – język rozumiany jest w postaci struktury. Struktura (nazywana też systemem języka) to zbiór elementów językowych (nazywany uniwersum) oraz zespół relacji określonych na tych elementach. Rozważać można zatem np. struktury głoskowe (zbiór głosek z relacją homofonii – jednakowego brzmienia) czy struktury wyrazowe (zbiór wyrazów języka z relacją homonimii – jednakowego znaczenia). Takie spojrzenie na język wydaje się być dobrym podejściem z punktu widzenia komputerowego przetwarzania języka.

Ferdinand de Saussure jest również autorem ogólnej teorii znaku. Znak rozumie jako dowolną rzecz (przedmiot, czynność, zjawisko), która jest nośnikiem jakiegoś znaczenia, treści. Jakkolwiek teoria ta nie dotyczy tylko znaków językowych (dźwięków, wyrazów, etc), lecz wszelkich znaków, to nauka o znaku – semiotyka, która powstała z teorii znaku de Saussure'a, jest bardzo istotna również z punktu widzenia komputerowego przetwarzania języka. Amerykański semiotyk Charles Morris zaproponował w 1938 roku, funkcjonujący dziś powszechnie, podział nauki o znaku na 3 podstawowe elementy: składnię, semantykę oraz pragmatykę[2]. Składnia zajmuje się budową znaku, jego formą. Rozważa się zależności między złożonymi strukturami znaków a tworzącymi je jednostkami (znakami) prostszymi. W analizie składni języka naturalnego bada się np. związki między wyrazami tworzącymi zdanie. Semantyka bada związki między znakiem (jego formą) a znaczeniem znaku (jego treścią). W kontekście lingwistyki komputerowej semantyka obejmuje np. dziedzinę ekstrakcji informacji, tj. poszukiwania informacji odpowiadających jakimś kryteriom. Pragmatyka – w ujęciu Morrisa – obserwuje natomiast zależności między znakami a użytkownikami znaków. W komputero-

[1] tytuł oryginału *Cours de linguistique générale*

[2] Morris Ch. (1938) *Foundations of the Theory of Signs*. W tej publikacji Morris składnię, semantykę i pragmatykę określa za pomocą terminów odpowiednio: *sign vehicle*, *designatum* oraz *interpreter*.

wym przetwarzaniu języka za element pragmatyki można uznać dla przykładu zagadnienie formułowania odpowiedzi na pytania użytkowników w systemach typu *chatter-bot*.

Niniejsza praca traktuje o analizie składniowej języka naturalnego. Zatem, zgodnie z podziałem wprowadzonym przez Morrisa, elementem nauki o znaku, na którym będziemy się koncentrować w pracy, jest składnia. Zajmować się będziemy zagadnieniem przedstawiania zdania w języku naturalnym w postaci struktury przedstawiającej relacje między elementami języka. Opiszemy program komputerowy, który dokonuje analizy składni języka francuskiego.

W pierwszym rozdziale pracy przybliżone zostaną podstawowe pojęcia związane z komputerowym przetwarzaniem języka naturalnego z punktu widzenia analizy składniowej. Omówimy techniki i strategie analizy składniowej stosowane w komputerowym przetwarzaniu języka. Rozdział 2. stanowić będzie przedstawienie techniki płytkiego parsingu. Przedyskutujemy związek między analizą składniową a ujednoznacznianiem morfologiczno-składniowym. Omówimy również koncepcję płytkiego parsingu wykorzystywaną w niniejszej pracy. W rozdziale 3. opiszemy płytki parser *puddle* opracowany przez autora pracy. Wprowadzimy formalizm reguł stosowanych przez parser płytki z omówieniem różnic w stosunku do koncepcji przedstawionych we wcześniejszym rozdziale. Rozdział ten obejmie także przedstawienie głównego algorytmu parsera *puddle*. Szczegóły dotyczące implementacji parsera *puddle* znajdują się w rozdziale 5. Przedstawimy działania, na których opiera się działanie parsera, a także format danych wykorzystywanych i generowanych przez zaimplementowany płytki parser. W czwartym rozdziale omówione zostaną zasoby leksykalne wykorzystywane podczas opracowywania płytkiego parsera *puddle*. Przedstawimy też klasyfikację części mowy oraz kategorii gramatycznych przyjętą na potrzeby opisywanego rozwiązania. Rozdział 6. będzie stanowił ewaluację opisywanego w pracy parsera płytkiego. Opiszemy przyjętą procedurę oceny wyników generowanych przez parser puddle i stosowane miary. Przedstawimy otrzymane wyniki i dokonamy ich interpretacji. Pracę zamknie rozdział podsumowujący przedstawione w pracy zagadnienia i narzędzia oraz osiągnięte w ramach pracy wyniki.

ROZDZIAŁ 1

Definicje podstawowych pojęć

1.1. Gramatyki

W potocznym rozumieniu, gramatyka to zbiór reguł, które rządzą wypowiedziami w danym języku. Takie rozumienie nie odbiega znacząco od słownikowej definicji gramatyki: system środków formalnych i semantycznych organizujących tekst językowy; uporządkowany zbiór reguł opisujących działanie tych środków[1]. Gramatyka zatem określa, w jaki sposób łączy się elementy języka w większe całości (np. tworzenie wyrazów, budowanie zdań z wyrazów). Gramatyka nie dotyczy więc wyłącznie zagadnień związanych ze składnią. W niniejszym rozdziale jednak skupimy się jedynie na składniowych aspektach gramatyk.

1.1.1. Gramatyki formalne

W komputerowym przetwarzaniu języka, jak i w informatyce w ogólności, wykorzystywane jest bardzo często pojęcie języków formalnych i gramatyk formalnych. Język formalny to zbiór zdań (ciągów symboli) nad pewnym alfabetem (skończonym zbiorem symboli).

Gramatykę formalną definuje się jako czwórkę (N, T, R, S), taką że:

1. N, T są skończonymi zbiorami symboli,
2. $N \cap T = \emptyset$,
3. R jest skończonym zbiorem par (P, Q), takich że:
   - $P \in (N \cup T)+$ – P jest dowolnym niepustym ciągiem symboli terminalnych i nieterminalnych,
   - $Q \in (N \cup T)*$ – Q jest dowolnym, także pustym, ciągiem symboli terminalnych i nieterminalnych,
4. $S \in N$.

Zbiory N i T nazywa się odpowiednio zbiorami symboli nieterminalnych i terminalnych gramatyki. Elementy zbioru R nazywa się regułami produkcji gramatyki. Wyróżniony symbol nieterminalny S nazywa się symbolem początkowym gramatyki.

[1] Szymczak M. red. (1978) *Słownik języka polskiego*. Warszawa, PWN.

Zbiór wszystkich zdań takich, że dla każdego z nich można znaleźć ciąg produkcji danej gramatyki nazywa się językiem generowanym przez tę gramatykę. O tym języku mówi się także, że dana gramatyka definiuje ten język.

Przykład 1.1

Niech $N = (S, A, B)$ oraz $T = (a, b)$ będą odpowiednio zbiorami symboli nieterminalnych i terminalnych pewnego języka. Niech S będzie symbolem początkowym gramatyki tego języka. Niech zbiór reguł produkcji gramatyki P zawiera następujące reguły:

$$\begin{aligned} S &\rightarrow A\ B \\ A &\rightarrow A\ a \\ A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

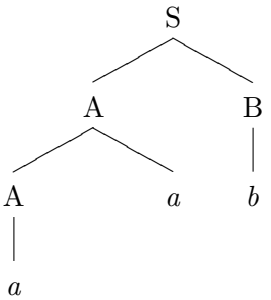
Gramatyka (N, T, P, S) definiuje język, składający się z ciągów dowolnej liczby (co najmniej jednego) symbolu a oraz dokładnie jednego symbolu b. Przykładowy ciąg symboli (zdanie) tego języka aab generowane jest przez podaną gramatykę w sposób następujący:

$S \rightarrow AB \rightarrow AaB \rightarrow aaB \rightarrow aab$

Ciąg reguł produkcji takich, że pierwszym elementem ciągu jest symbol początkowy gramatyki, a ostatnim zdanie języka opisywanego przez gramatykę nazywa się wypowiedzeniem lub wywodem tego zdania.

Wypowiedzenie zdania gramatyki można zobrazować także w formie drzewa. Korzeniem jest symbol początkowy gramatyki, węzłami symbole nieterminalne, zaś w liściach drzewa znajdują się symbole terminalne. Drzewa takie nazywa się drzewami wywodu lub drzewami składniowymi (syntaktycznymi).

Drzewi wywodu dla przykładowej produkcji przedstawionej powyżej ma postać:



1.1.2. Hierarchia gramatyk Chomsky'ego

W pracy (Chomsky, 1956) zapronowana została klasyfikacja gramatyk formalnych ze względu na postać reguł produkcji, która stanowi podstawową hierarchię gramatyk formalnych. Hierarchię tę nazywa się hierarchią Chomsky'ego lub hierarchią Chomsky'ego-Schützenbergera[2]

[2]Marcel-Paul Schützenberger - francuski matematyk, który odegrał znaczący wpływ w tworzeniu teorii języków formalnych

W omawianych produkcjach gramatyki przyjęta została konwencja, że *a, b* oznaczają symbole terminalne gramatyki, *A, B* – symbole nieterminalne. Natomiast α, β, γ są dowolnymi ciągami symboli terminalnych i nieterminalnych, przy czym γ jest ciągiem niepustym.

Hierarchia Chomsky'ego wyróżnia cztery typy gramatyk:

Gramatyki typu 0 obejmują wszystkie gramatyki formalne. Języki generowane przez gramatyki tego typu nazywa się językami rekurencyjnie przeliczalnymi.

Gramatyki typu 1 zawierają gramatyki, których produkcje mają postać: $\alpha A \beta \rightarrow \alpha \beta \gamma$. Gramatyki te noszą nazwę gramatyk kontekstowych (*context-sensitive grammars*), a języki przez nie generowane nazywane są językami kontekstowymi. Dla przykładu przyjrzyjmy się gramatyce kontekstowej opisującej język kontekstowy ($a^n b^n c^n$, $n > 1$) czyli składający się z napisów *abc*, *aabbcc*, *aaabbbccc* itd. Gramatyka składa się z symboli nieterminalnych S, A, B, C, X oraz symboli terminalnych a, b, c. Zbiór produkcji gramatyki obejmuje następujące produkcje:

$$
\begin{array}{lccc}
1 & S & \rightarrow & aSBC \\
2 & S & \rightarrow & aBC \\
3 & CB & \rightarrow & XB \\
4 & XB & \rightarrow & XC \\
5 & XC & \rightarrow & BC \\
6 & aB & \rightarrow & ab \\
7 & bB & \rightarrow & bb \\
8 & bC & \rightarrow & bc \\
9 & cC & \rightarrow & cc
\end{array}
$$

Przykładowe zdanie tego języka *aabbcc* jest generowane przez gramatykę za pomocą następującego wywodu (ciągu produkcji):
$S \xrightarrow{1} aSBC \xrightarrow{2} aaBCBC \xrightarrow{3} aaBXBC \xrightarrow{4} aaBXCC \xrightarrow{5} aaBBCC \xrightarrow{6} aabBCC \xrightarrow{7} aabbCC \xrightarrow{8} aabbcC \xrightarrow{9} aabbcc$
Liczby nad strzałkami oznaczają numer reguły ze zbioru produkcji gramatyki wykorzystanych w danym kroku produkcji zdania.

Gramatyki typu 2 to gramatyki, w których reguły produkcji są postaci $A \rightarrow \alpha$. Gramatyki tego typu noszą nazwę gramatyk bezkontekstowych (*context-free grammars*), a języki generowane przez nie języków bezkontekstowych.

Przykładowy zbiór produkcji prostej gramatyki bezkontekstowej opisującej fragment języka polskiego może mieć postać:

Gramatyka PL1

$$
\begin{array}{lcll}
S & \rightarrow & NP & VP \\
VP & \rightarrow & V & NP \\
NP & \rightarrow & N &
\end{array}
$$

Przykładowy zbiór produkcji prostej gramatyki bezkontekstowej opisującej fragment języka francuskiego może mieć postać:

Gramatyka FR1

$$\begin{array}{lcll} S & \rightarrow & NP & VP \\ VP & \rightarrow & V & NP \\ NP & \rightarrow & ART & N \\ NP & \rightarrow & N & \end{array}$$

W powyższych przykładach N, V, ART są symbolami nieterminalnymi, które odpowiadają poszczególnym częściom mowy. Są to symbole nieterminalne, z których produkuje się tylko symbole terminalne. Przykładowa produkcja polskiego rzeczownika *matka* ma postać: $N \rightarrow matka$

W gramatykach typu 3 reguły produkcji są ograniczone do postaci $A \rightarrow a$ lub $A \rightarrow aB$. Gramatyki typu 3 nazywa się gramatykami regularnymi. Języki generowane przez gramatyki regularne to języki regularne.

Użyteczność hierarchii zapronowanej przez Chomsky'ego polega na tym, że języki generowane przez gramatyki wyróżnionych typów dokładnie odpowiadają językom generowanym przez pewne abstrakcyjne modele maszyn. I tak, gramatyce typu 0 odpowiada maszyna Turinga, zaś gramatyce typu 1 automat liniowo ograniczony. Z kolei gramatyce bezkontekstowej odpowiada automat ze stosem, natomiast gramatyce regularnej automat skończony.

Niekiedy (np. Grune, Jacobs, 1998) wprowadza się także gramatyki typu 4 (*choice-free grammars*), w których prawe strony każdej produkcji mogą zawierać tylko symbole terminalne. Takie gramatyki nie posiadają w zasadzie mocy generatywnej, a produkcja sprowadza się do wyboru z możliwych alternatyw.

1.2. Analiza składniowa

Analiza składniowa zdania jest to czynność polegająca na stworzeniu takiej struktury reprezentującej analizowane zdanie, w której każdemu elementowi zdania przypisana jest informacja składniowa. Efektem analizy składniowej jest struktura składniowa zdania. Analiza składniowa odbywa się w oparciu o pewną ustaloną gramatykę.

Analiza składniowa nazywana jest również niekiedy parsingiem lub parsowaniem. Program komputerowy przeprowadzający analizę składniową nazywa sie analizatorem składniowym bądź parserem.

W tej pracy przyjmiemy, że wynik działania analizy składniowej czyli struktura reprezentująca strukturę składniową zdania ma postać drzewa. Strukturę tę nazywać będziemy drzewem struktury składniowej bądź drzewem parsingu analizowanego zdania.

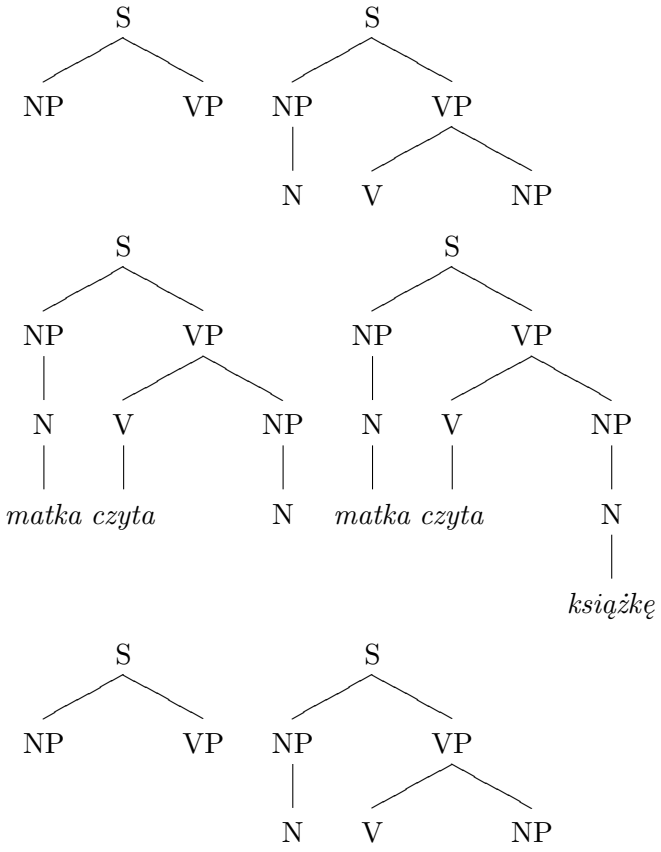
1.2.1. Strategie parsingu

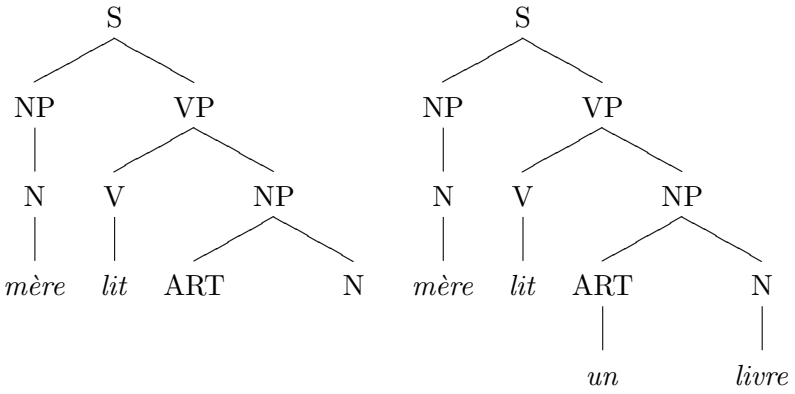
Na parsing zdania można spojrzeć, jak na określenie tego, czy i w jaki sposób, używając reguł danej gramatyki, możliwe jest wygenerowanie analizowanego zdania. Innymi słowy, parsing zdania można traktować jako poszukiwanie wypowiedzenia danego zdania w danej gramatyce. Można zastosować dwa podejścia do tego problemu: wychodząc od symbolu początkowego gramatyki poszukuje się kolejnych reguł produkcji prowadzących do analizowanego zdania, bądź też wychodząc od analizowanego zdania dąży się do symbolu początkowego gramatyki. W zależności od zastosowanego podejścia wyróżnia się dwie podstawowe strategie parsingu.

Strategia zstępująca (*top-down parsing*)

Parser działający zgodnie ze strategią zstępującą rozpoczyna działanie od symbolu początkowego gramatyki. Kolejne kroki postępowania parsera polegają na wykonywaniu produkcji zgodnie z regułami gramatyki. Parsing zostaje zakończony, gdy po zastąpieniu wszystkich symboli nieterminalnych symbolami terminalnymi otrzymane zostanie analizowane zdanie.

Poniżej przedstawione zostaną kolejne etapy działania parsera zstępującego dla przykładowych gramatyk bezkontekstowych dla języka polskiego i francuskiego przedstawionych we wcześniejszej części rozdziału (Gramatyka PL1 oraz Gramatyka FR1). Analizowane zdania to *matka czyta książkę* oraz *mère lit un livre*.

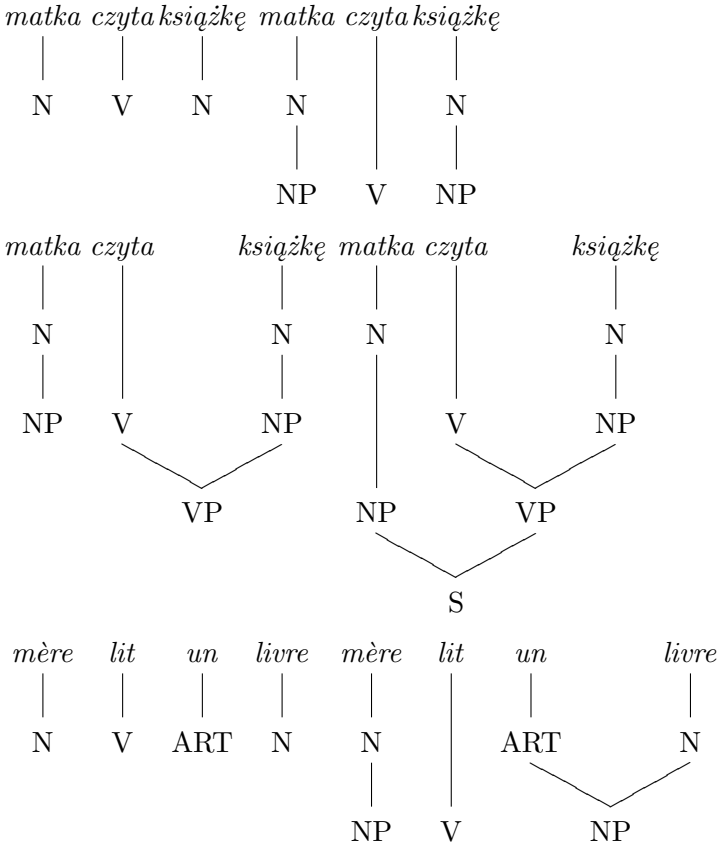


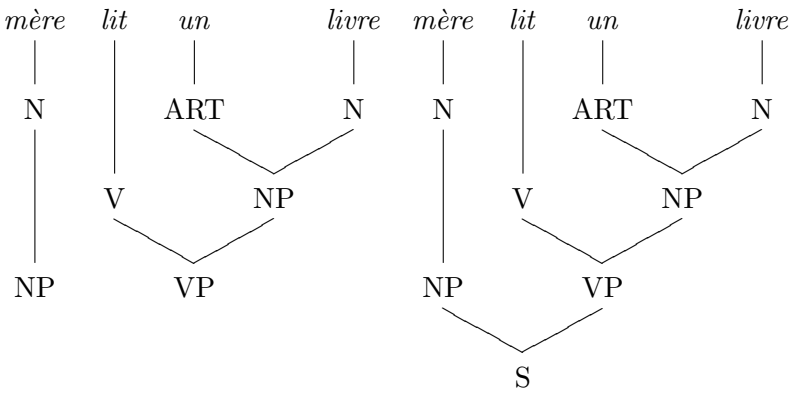


Strategia wstępująca (*bottom-up parsing*)

Punktem wyjścia parsingu działającego zgodnie ze strategią wstępującą jest analizowane zdanie. Następnie parser wykorzystuje reguły produkcji, w których prawymi stronami produkcji są elementy zdania. W ten sposób z podstawowych elementów tworzone są większe struktury, aż do momentu gdy parser napotka regułę produkcji, której lewą stroną jest symbol początkowy gramatyki.

Poniżej przedstawione zostaną kolejne etapy działania parsera wstępującego dla przykładowych gramatyk bezkontekstowych dla języka polskiego i francuskiego odpowiednio gramatyk PL1 oraz FR1. Analizowane zdania to *matka czyta książkę* oraz *mère lit un livre*.





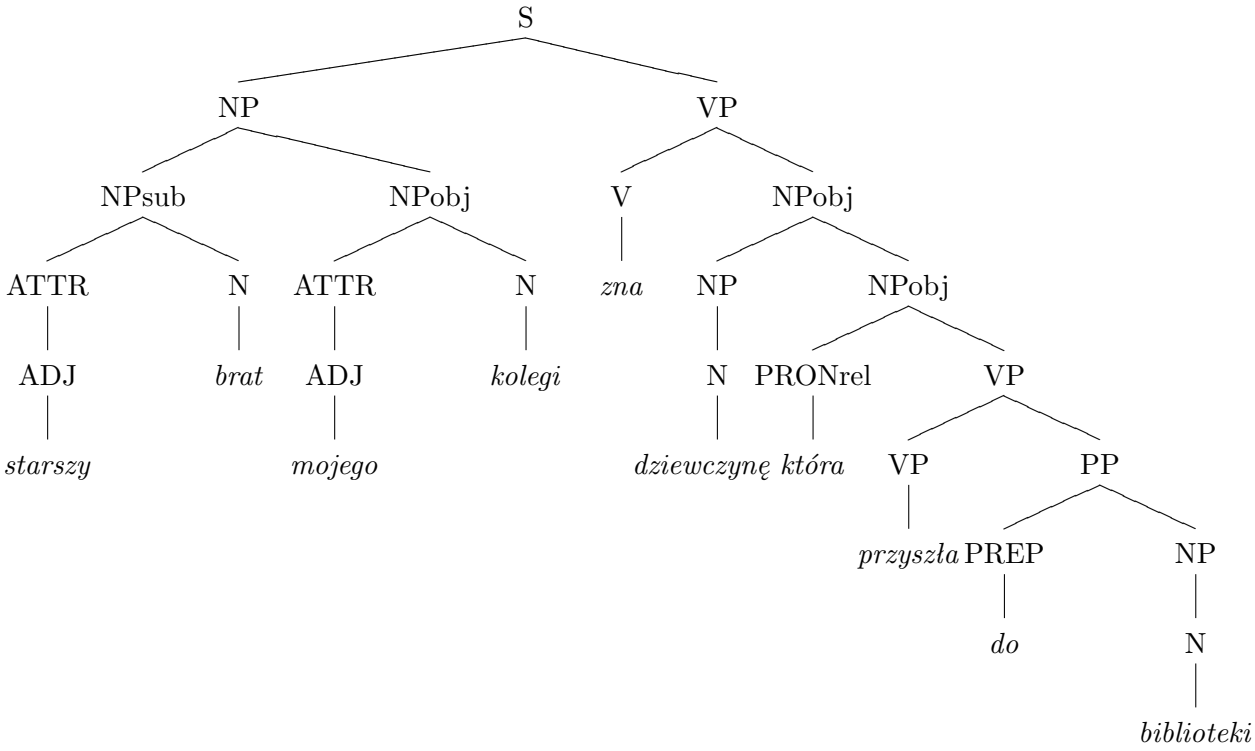
1.3. Płytki parsing

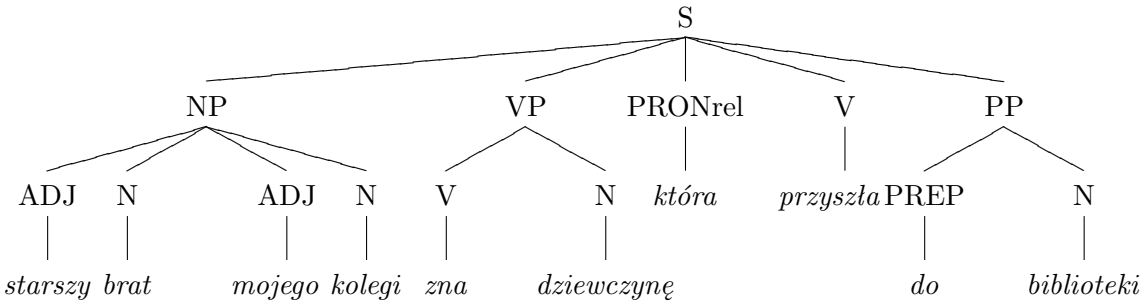
Jak wspomnieliśmy, celem parsingu jest uzyskanie pełnej struktury składniowej analizowanego zdania. W niektórych zastosowaniach stosowane jest podejście, w którym analiza polega na rozpoznawaniu elementów składowych zdania bez rozpoznawania całej jego struktury składniowej. Innymi słowy, w tej technice rozpoznawane w zdaniu wejściowym są pewne wybrane struktury np. grupy rzeczownikowe, grupy czasownikowe, jednak nie jest rozpoznawana pełna ich struktura wewnętrzna. Nie są też analizowane role, jakie grupy te pełnią w zdaniu.

Opisana technika nosi nazwę parsingu płytkiego lub powierzchniowego (*shallow parsing*) bądź parsingu częściowego (*partial parsing*). Dla odróżnienia, parsing, który analizuje pełną strukturę składniową, omówiony we wcześniejszej części rozdziału, nazywa się parsingiem głębokim bądź pełnym. W literaturze angielskojęzycznej (np. NLTK) funkcjonuje także dla płytkiego parsingu określenie *chunking* pochodzące od terminu *chunk* oznaczającego podstawową cześć budującą zdanie, np. grupa rzeczownikowa jest nazywana *noun chunk*.

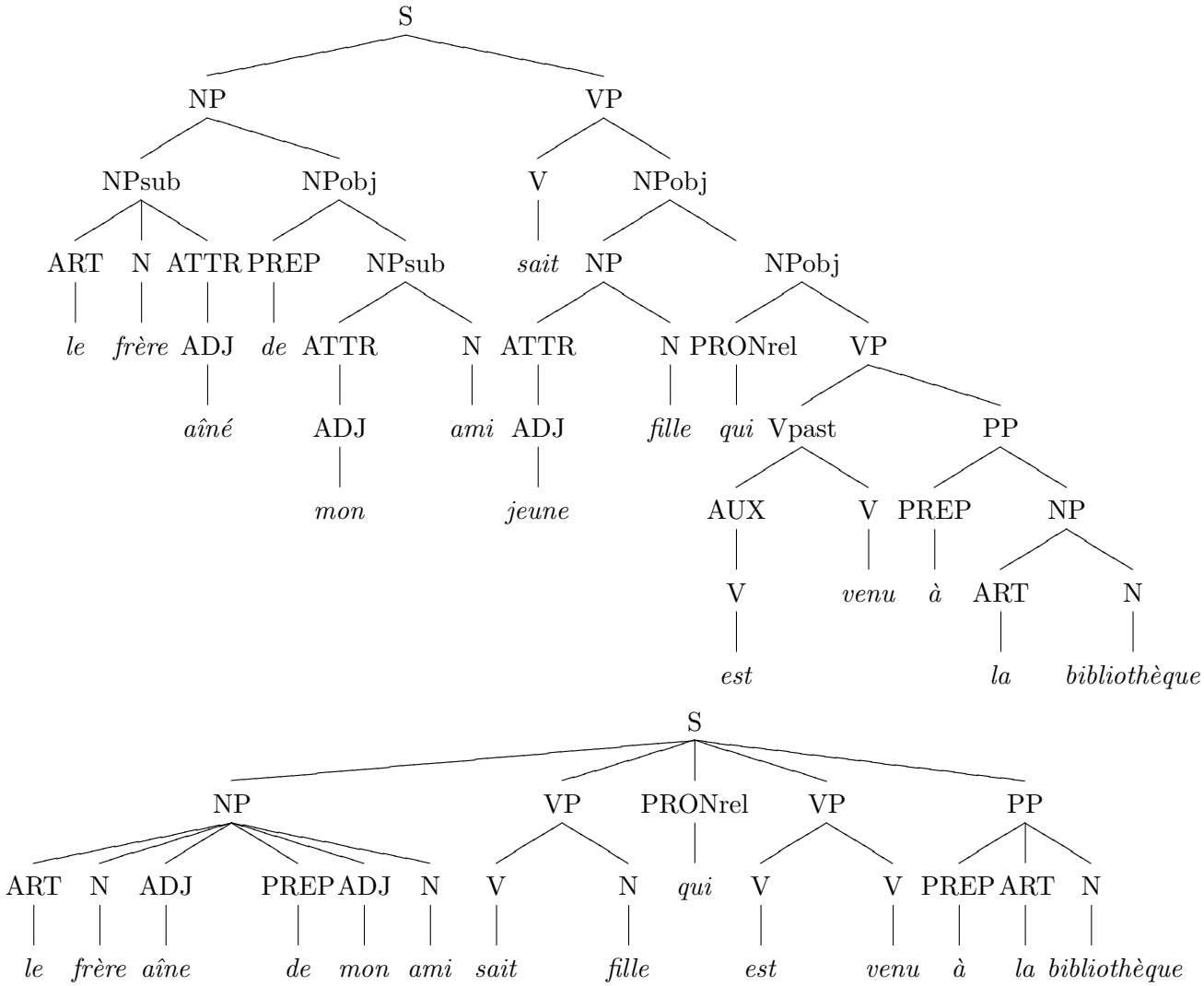
Strukturę rozpoznaną w wyniku płytkiego parsingu można również przedstawić w postaci drzewa. Zgodnie z intuicją, dla takich samych zdań drzewa struktury otrzymanej jako wynik parsingu płytkiego są mniej złożone od pełnych drzew struktury składniowej. Struktury generowane przez parsery głębokie są zagnieżdżone i złożone z wielu poziomów. Z kolei drzewa otrzymane w wyniku pracy parserów płytkich zawierają mniej poziomów zagnieżdżenia. Można zauważyć, że drzewo parsingu płytkiego to *de facto* drzewo zbudowane z kilku najbliższych liściom (najbardziej szczegółowych) poziomów drzewa pełnego parsingu.

Poniżej przedstawione są kolejno drzewo pełnej struktury składniowej oraz drzewa struktury płytkiej polskiego zdania: *starszy brat mojego kolegi zna dziewczynę, która przyszła do biblioteki.*





Poniżej przedstawione są kolejno drzewo pełnej struktury składniowej oraz drzewa struktury płytkiej francuskiego zdania: *le frère aîné de mon ami le sait la jeune fille, qui est venu à la bibliothèque.*



Płytki parsing nie stawia sobie za cel pełnej analizy struktury zdania, jest zatem problemem prostszym, także pod względem obliczeniowym. Parsery pełne wykorzystują często złożone metody statystyczne. Natomiast do celów parserów płytkich wykorzystuje się metody prostsze i oszczędniejsze, zarówno pod względem wykorzystywanej pamięci, jak i czasu wykonywania. W pracy przedstawione zostanie rozwiązanie wykorzystujące kaskadę automatów skończonych.

W literaturze znaleźć można opisy zastosowania płytkiego parsingu do ekstrakcji informacji (Jurafsky, Martin, 2000), rozpoznawania fraz (Appelt, Israel, 1999; Sha, Pereira, 2003), oznaczania jednostek nazwanych (McCallum, Li, 2003). Natomiast rozwiązanie omawiane w niniejszej pracy ma służyć zastosowaniu parsingu płytkiego w tłumaczeniu automatycznym z języka francuskiego.

1.4. Analiza morfologiczna

Tak, jak analiza składniowa zajmuje się określaniem, w jaki sposób wyrazy budują zdanie, tak dziedziną zajmującą się analizą wewnętrznej budowy wyrazów jest analiza morfologiczna. Analizę morfologiczną można określić jako działanie, którego celem jest zidentyfikowanie w wyrazie jego poszczególnych elementów składowych – zwanych morfemami – wraz z przypisaniem informacji morfologicznych do rozpoznanych morfemów. Wyróżnia się morfemy leksykalne tj. przenoszące informacje o charakterze słownikowym oraz morfemy gramatyczne tj. służące do wyrażania informacji gramatycznych. Przyjrzymy się dla przykładu polskiemu wyrazowi *domów*. Można w nim wyróżnić morfem leksykalny *dom-* oznaczający m. in. budynek przeznaczony na mieszkania lub pomieszczenie mieszkalne[3] oraz morfem gramatyczny *-ów*, który jest m. in. wyznacznikiem dopełniacza liczby mnogiej rzeczownika rodzaju męskiego[4].

Języki różnią się ze względu na strukturę morfologiczną. W językach fleksyjnych, których przykładem jest język polski, morfologia jest bardzo rozbudowana. Morfemy mają tendencje do *łączenia się* w jeden, stąd występuje wiele morfemów gramatycznych. Często jeden morfem gramatyczny może pełnić więcej niż jedną funkcję gramatyczną, w zależności od innych morfemów, z którymi występuje w obrębie wyrazu.
Za przykład niech posłuży końcówka *-u* rzeczowników w języku polskim. W zależności od użycia, morfem ten może nieść informację m. in. o narzędniku liczby pojedynczej (*zostajemy w domu*), dopełniaczu liczby pojedynczej (*nie obrabowałem sklepu*) czy też celowniku liczby pojedynczej (*wybieram prezent dziecku*).

Analiza morfologiczna w językach fleksyjnych jest ze względu na złożoną morfologię języka procesem całkiem skomplikowanym.

W językach aglutynacyjnych (np. język fiński) morfemy gramatyczne pełnią zazwyczaj tylko jedną funkcję gramatyczną. Morfemy gramatyczne są dołączane do wyrazu jeden za drugim, nie występuje zjawisko *łączenia* kilku morfemów gramatycznych w jeden.
W fińskim wyrazie *talossanikaanko*, który można przetłumaczyć jako *czy także nie w moim domu?*, wyróżnić możemy następujące morfemy:

- *talo-* – morfem leksykalny oznaczający dom, budynek,
- *-ssa-* – końcówka przypadka Inessivus, oznaczającego m. in. przebywanie we wnętrzu obiektu,
- *-ni-* – sufiks dzierżawczy pierwszej osoby liczby pojedynczej,
- *-kaan-* – morfem oznaczający zanegowane *też* (*też nie*, *również nie*),
- *-ko* – końcówka pytająca (można dostrzec pewne podobieństwo do staropolskiego *-li?*).

[3] definicje za Szymczak M. red. (1978) *Słownik języka polskiego*. Warszawa, PWN.
[4] dokładniej: rodzaju męskiego nieżywotnego

Cechą charakterystyczną języków aglutynacyjnych jest to, że wyraz składa się z jednego morfemu leksykalnego oraz szeregu morfemów gramatycznych.

W skład kolejnej grupy języków wchodzą języki izolujące (np. język chiński, wietnamski czy współczesny tybetański). W tej grupie zdania i wypowiedzi tworzone są poprzez zestawianie wyrazów nieodmiennych co do formy. Wewnętrzna struktura wyrazu nie zmienia się – w językach izolujących nie ma przyrostków, przedrostków, końcówek przypadków itd. Poniżej przedstawione są dwa zdania w języku wietnamskim[5]:
nhũ'ng cái bàn này rông – te szerokie stoły
nhũ'ng cái bàn rông này – te stoły są szerokie
Wyrazami występującymi w przykładzie są: *cái bàn* – stół, *rông* – szeroki, *nhũ'ng* – wykładnik liczby mnogiej, *này* – zaimek wskazujący.
Jak widać, w językach izolujących zmianę znaczenia uzyskuje się przez zmianę szyku wyrazów.

Jeszcze inną grupę stanowią języki analityczne. W tej grupie informacje gramatyczne wyrażane są przez odrębne morfemy występujące w postaci osobnych wyrazów (jak np. czasownik posiłkowy, przysłówek). Przykładem języka analitycznego jest język francuski. Elementy fleksji w języku francuskim można zauważyć w końcówkach liczby mnogiej czy końcówkach pewnych form czasownika

Ze względu na tematykę pracy oraz na właściwości języka francuskiego, w opisywanych w niniejszej pracy rozwiązaniach nie będziemy skupiać się na analizie morfologicznej. W omawianej implementacji informacje gramatyczne przypisywane będą przy wykorzystaniu słownika form fleksyjnych wyrazów francuskich. Prezentacja słownika form fleksyjnych znajduje się w rozdziale czwartym.

[5]przykład za: Bańczerowski J., Pogonowski J., Zgółka T. (1982) *Wstęp do językoznawstwa*. Poznań, Wydawnictwo UAM, str. 89.

ROZDZIAŁ 2

Płytki parsing

W niniejszym rozdziale przedstawiona zostanie koncepcja płytkiego parsingu, która była punktem wyjścia dla rozwiązania zaimplementowanego przez autora pracy – omawianego w kolejnych rozdziałach.

2.1. Wstęp

W pracy (Joshi, Hopely, 1997) autorzy dokonują przedstawienia i rekonstrukcji parsera rozwijanego w latach 1958-1959 na Univerisity of Pennsylvania w ramach projektu TDAP[1]. Parser ten był, prostą z dzisiejszego punktu widzenia, kaskadą przetworników skończonych i umożliwiał rozpoznawania m. in. prostych fraz, grup rzeczownikowych i dopełnień czasowników w tekstach w języku angielskim. Było to pierwsze zastosowanie przetworników skończonych w analizie składniowej. Ze względu na stosowane przez autorów parsera rozwijanego w ramach TDAP, można nazwać to rozwiązanie pierwszym parserem płytkim, mimo że sami autorzy tak go nie klasyfikowali.

W atykule (Kinyon, 2001) autorka podzieliła techniki używane przy tworzeniu parserów płytkich na dwie główne grupy:

1. techniki probabilistyczne,
2. techniki wykorzystujące automaty i przetworniki skończone.

Techniki probabilistyczne opierają się na wiedzy pozyskanej ze źródeł anotowanych składniowo. Na podstawie analizy źródeł (proces ten nazywa się trenowaniem bądź nauką parsera), parser płytki dokonuje analizy tekstu opierając się na prawdopobieństwie występowania i budowy określonych fraz, konstrukcji czy struktur. Autorka zauważa, że techniki zaliczone do pierwszej grupy wymagają dużych ilości danych uczących. Techniki te mogą być skutecznie wykorzystywane tylko w przypadku języków, dla których są dostępne odpowiednio duże zasoby leksykalne[2]. Dane, które zostały wykorzystane do trenowania parsera, determinują efekty jego działania. Parser wytrenowany na zbiorze tekstów z oznaczonymi jednostkami pewnego typu jest w stanie rozpoznawać jedynie jednostki podobnego rodzaju.

[1] *Transformations and Discourse Analysis Project* - projektem kierował Zellig S. Harris.

[2] Zdaniem autorki wystarczająco duże zasoby dostępne są wyłącznie dla języka angielskiego. Pozostałe języki cierpią na problemy związane z niewystarczającą ilością lub zupełnym brakiem takich zasobów leksykalnych.

Wspomnianej niedogodności pozbawione są techniki wykorzystujące automaty skończone. W rozwiązaniach z tej grupy wykorzystywane są reguły, które opisują budowę jednostek, które ma analizować i rozpoznawać płytki parser. Na podstawie reguł (często opartych na wyrażeniach regularnych) tworzone są automaty skończone, które są wykorzystywane przez parser. Autorka sugeruje, że problemem, który pojawia się w przypadku stosowania tego podejścia do płytkiego parsingu, jest potrzeba ręcznego stworzenia sporego zbioru reguł, które muszą uwzględniać wszystkie możliwości budowy i struktury jednostek analizowanych przez parser.

Rozwiązania prezentowane w niniejszej pracy wykorzystują techniki oparte na automatach skończonych.

2.2. Związek płytkiego parsingu z ujednoznacznaniem morfologiczno-składniowym

Ujednoznacznienie (dezambiguacja) morfologiczno-składniowe to działanie polegające na przypisaniu elementom tekstu (wyrazom, frazom) jednoznacznych informacji morfologicznych i składniowych. Niejednokrotnie jedna forma graficzna wyrazu odpowiada różnym formom tych samych lub innych wyrazów. Dla przykładu w zdaniu w języku polskim *Na ulicach trwały zamieszki* wyraz *trwały* jest formą 3. osoby liczby mnogiej czasownika *trwać*. Identyczną formę graficzną ma też forma liczby pojedynczej rodzaju męskiego przymiotnika *trwały*. Zadaniem ujednoznaczniania morfologiczno-składniowego w tym przykładowym zdaniu jest więc zatem wybór prawidłowej formy wyrazu i przypisanie mu wspomnianych cech czasownika.

Poza różnicami między parsingiem głębokim a parsingiem płytkim wynikającymi z samego podejścia do analizy składniowej, przedstawionymi w rozdziale pierwszym, często te rodzaje parsingu różnią się tym, czy tekst wejściowy, który analizuje parser, jest ujednoznaczniony pod względem morfologiczno-składniowym, czy nie. Zazwyczaj parser głęboki dopuszcza tekst niejednoznaczny, bowiem w procesie pełnej analizy struktury składniowej zdania bierze on pod uwagę różne możliwe interpretacje wyrazów, które wpływać mogą na powstanie różnych struktur zdania w wyniku parsingu. W konkretnej strukturze składniowej powstałej w wyniku parsingu głębokiego poszczególne wyrazy zostają zatem ujednoznacznione. Z kolei parsery płytkie w większości operują na tekście, który został uprzednio ujednoznaczniony.

W kilku opracowaniach, m. in. w (Neumann, Braun, Piskorski, 2000) oraz w (Marimon, Porta, 2000) zapronowane zostały rozwiązania, które pozwalają na przeplatanie ujednoznaczniania i parsingu płytkiego. W pracy (Neumann, Braun, Piskorski, 2000) autorzy opisują system, którego skuteczność opiera się na poprawnym rozpoznawaniu grup czasownikowych w tekstach w języku niemieckim. W tym celu wprowadzają pewne mechanizmy ujednoznaczniające, które służą do rozpoznawania, kiedy wyraz, który może być formą cza-

sownika, jest tym czasownikiem w istocie. Natomiast system, który został opisany w (Marimon, Porta, 2000) dokonuje ujednoznaczniania tekstów w języku hiszpańskim poprzez analizowanie otoczenia (kontekstu), w jakich pojawiają się wyrazy. Dzięki temu, jednocześnie z ujednoznacznianiem, rozpoznawane są także podstawowe jednostki w tekście, np. frazy rzeczownikowe.

Rozwiązanie, które zostanie opisane w paragrafie 2.3 jest koncepcją, która umożliwia jednoczesne ujednoznacznianie morfologiczno-składniowe i płytki parsing.

2.3. Płytki parser *Spejd*

2.3.1. Charakterystyka formalizmu

We wstępie pracy (Przepiórkowski, Buczyński, 2007) autorzy zauważają, że ujednoznacznianie morfologiczno-składniowe oraz płytki parsing są procesami, które wpływają na siebie wzajemnie, a jeden korzysta z wyników drugiego i *vice versa*. Dlatego, zdaniem autorów, powinny być one wykonywane jednocześnie, a nie, jak to ma zazwyczaj miejsce, jeden po drugim. Ponadto autorzy sugerują, że obie wspomnione operacje często wykorzystują te same intuicje językowe i dostrzegają potrzebę stworzenia formalizmu, w którym można by opisywać reguły opisujące ujednoznacznianie oraz tworzenie stuktur składniowych.

We wspomnianej pracy (Przepiórkowski, Buczyński, 2007) oraz w (Buczyński, Przepiórkowski, 2008) autorzy prezentują taki formalizm oraz narzędzie, które działa w oparciu o niego. Formalizm, a także płytki parser go wykorzystujący, noszą nazwę *Spejd*, co autorzy proponują traktować jako skrót od ang. *Shallow Parsing Engine Jointly with Disambiguation* lub pol. *Składniowy Parser (Ewidentnie Jednocześnie Dezambiguator)*[3]. Formalizm *Spejd* oparty jest w części na języku zapytań *CQP*[4] opisanym w (Evert, Kermes, 2003) oraz (Evert, 2005).

Jak zostało wspomniane, Spejd wykorzystuje podejście, w którym ujednoznacznianie morfologiczno-składniowe oraz płytki parsing wykonywane są jednocześnie i równolegle. Formalizm ten jest konstruktywny – służy łączeniu elementów w większe całości (np. wyrazów we frazy rzeczownikowe). Ze względu na postać reguł, które wykorzystywane są w formalizmie Spejd, parser Spejd ma formalnie moc kaskady gramatyk regularnych.

[3] Inne rozwinięcia skrótu przedstawione w (Buczyński, Przepiórkowski, 2008) to niem. *Syntaktisches Parsing Entwicklungsystem Jedoch mit Desambiguierung* oraz fr. *Super Parseur Et Jolie Désambiguïsation*.

[4] Język zapytań *CQP* oraz mechanizmy go wykorzystujące stanowią centralną część *CWB – The IMS Open Corpus Workbench* – zbioru narzędzi do zarządzania i przeszukiwania dużych korpusów tekstów. Jest to projekt rozpoczęty na Uniwersytecie w Stuttgarcie, a od 2007 roku rozwijany na zasadzie *open source*. Narzędzia dostępne są pod adresem internetowym `http://cwb.sourceforge.net/` [Dostęp 18 maja 2009].

2.3.2. Budowa reguł

Reguła składa się maksymalnie z pięciu części: `Rule`, `Left`, `Match`, `Right` oraz `Eval`, z których `Rule`, `Match` oraz `Eval` są obowiązkowe.

W części `Rule` określa się nazwę reguły.

Część `Match` zawiera specyfikację elementów tekstu, na których operacje opisane zostały w regule. Np.[5] określenie `Match: [pos~"prep"] [base~"co|kto"];` oznacza, że reguła opisuje działania, jakie należy wykonać, gdy w tekście pojawi się wyraz oznaczony jako przyimek (*prep* to oznaczenie przyimka przyjęte przez autorów) poprzedzający wyraz, którego formą bazową jest *co* lub *kto*.

W części `Eval` definiowany jest ciąg predykatów podobnych do predykatów języka Prolog[6], które mogą przyjmować wartość *prawda* lub *fałsz*. Z predykatami związane są akcje, tj. działania, które są wykonywane przez parser na dopasowanych (tzn. odpowiadających specyfikacji w część `Match` reguły) elementach. Np. ciąg predykatów `Eval: unify(case, 1, 2); group(PG, 1, 2);` oznacza, że na dopasowanych elementach mają zostać wykonane następujące działania:

1. uzgodnienie przypadku między elementami pierwszym i drugim w specyfikacji w części `Match`,

2. stworzenie z elementów dopasowanych do specyfikacji w części `Match` grupy przyimkowej (oznaczenie przyjęte przez autorów PG), których głową składniową będzie element pierwszy, a głową semantyczną element drugi.

W częściach `Left` i `Right` określać można kontekst lewy i prawy reguły, tj. specyfikować elementy, które mają pojawić się odpowiednio w bezpośrednim lewym bądź prawym otoczeniu elementów określonych w części `Match`.

W częściach `Match`, `Left` oraz `Right` określa się cechy związane z własnościami tokenów tekstu bądź grup (struktur wyższego poziomu stworzonych z tokenów bądź innych grup, jak np. grupa przyimkowa w przykładzie akcji w części `Eval` przedstawionym powyżej). Składnia tych specyfikacji jest zgodna z rozszerzeniem składni języka zapytań narzędzia *Poliqarp* zaproponowanej w (Przepiórkowski, 2007)[7]. Ponadto w formalizmie Spejd wprowadzone zostały dodatkowe symbole specjalne: `sb` (początek zdania - *sentence beginning*), `se` (koniec zdania - *sentence end*) oraz `ns` (brak odstępu - *no space*). Symbole *sb* oraz *se* umożliwiają uwzględnianie specyficznych struktur i zjawisk specyficznych dla początku i końca zdania.

[5] Wszystkie przykłady reguł oraz fragmentów reguł przedstawione w tym rozdziale pochodzą z pracy (Buczyński, Przepiórkowski, 2008).

[6] opis języka Prolog znaleźć można m. in. w pracy (Gazdar, Mellish, 1989).

[7] *Poliqarp* to narzędzie do przeszukiwania korpusu tekstów języka polskiego IPI PAN. Narzędzie to i oryginalną składnię języka zapytań tego programu opisuje (Przepiórkowski i inni, 2004). Korpus IPI PAN przedstawiony został m. in. w (Przepiórkowski, 2004).

Specyfikacja może także zawierać alternatywy wspomnianych elementów. Przykładowa specyfikacja `([pos~~"subst"] | [synh=[pos~~"subst"]] se)` oznacza rzeczownik[8] lub grupę, której głowa składniowa jest rzeczownikiem, po których następuje koniec zdania. Ponadto do powyższych określeń można stosować także kwalifikatory używane w wyrażeniach regularnych `?`, `*` oraz `+`.

Z predykatami, które można wykorzystywać w części `Eval` reguły, związane są akcje, które usuwają lub dodają interpretacje morfologiczne do poszczególnych elementów albo wykonują uzgodnienie wskazanych kategorii gramatycznych (w ten sposób dokonywane jest ujednoznacznienie tekstu) oraz akcje, które tworzą struktury wyższego rzędu (w ten sposób tworzona zostaje struktura składniowa tekstu).

Przeanalizujmy dwie przykładowe reguły w formalizmie Spejd:

```
Left: sb;

Match: [orth~"[Nn]ie"];

Eval: leave(pos~qub, 2);
```

Powyższa reguła jest dopasowywana do wyrazu, którego forma ortograficzna to *nie* lub *Nie*, znajdującego się na początku zdania. Po dopasowaniu, z tokenu drugiego (tzn. wyrazu *Nie* lub *nie*) usunięte zostaną wszystkie interpretacje morfologiczne poza przysłówkiem.

```
Match: [pos~"adj"]* [pos~"subst"] [synh=[pos~~"subst|num" && case~~"gen"]];

Right: [synh=[]];

Eval: unify(case number gender, 2, 3);

group(NG, 3, 3);
```

Powyższa reguła tworzy grupę rzeczownikową (predykat `group`) z rzeczownika i znajdującej się za nim grupy, której głową jest rzeczownik lub liczebnik w dopełniaczu. Dodatkowo, przed rzeczownikiem mogą znajdować się przymiotniki. Predykat `unify` zapewnia, że przymiotniki i rzeczownik będą pozostawać w związku zgody ze względu na przypadek, liczbę oraz rodzaj.

W rozwiązaniu opisanym w niniejszej pracy stosowany jest formalizm bazujący na formalizmie *Spejd*. Dokładny i formalny opis formalizmu znajdzie się w rozdziale 3. W rozdziale tym znajdzie się także opis zmian i rozszerzeń wprowadzonych do formalizmu *Spejd* przez autora pracy.

2.3.3. Działanie i zastosowanie

Parser Spejd działa na tekście oznaczonym morfologicznie w formacie XCES[9]. W takim samym formacie zapisywany jest wynik działania parsera. Formalizm *Spejd* jest formalizmem niezależnym od języka, w jakim jest napisany wejściowy tekst. Formalizm jest także niezależny

[8] oznaczenia części mowy przyjęte przez autorów zgodne są z oznaczeniami używanymi w korpusie IPI PAN

[9] *XML Corpus Encoding Standard* - jest to format zapisu tekstu oparty na standardzie XML. Specyfikacja formatu w (XCES).

od oznaczeń stosowanych dla części mowy i kategorii gramatycznych używanych do oznaczania tekstu wejściowego.

Parser *Spejd* zamienia specyfikacje w częściach `Left`, `Match` i `Right` reguł na wyrażenia regularne. Tekst wejściowy zamieniany jest na postać wewnętrzną, do której dopasowane są wyrażenia regularne, na które zamienione zostały reguły. Dopasowanie odbywa się przy wykorzystaniu niedeterministycznych automatów skończonych.

Formalizm *Spejd* wykorzystywany jest przy tworzeniu gramatyki dla korpusu języka polskiego IPI PAN. W wyniku tych działań powstała gramatyka obejmująca ponad 350 reguł w formalizmie *Spejd*. Szczegóły tych działań opisane zostały w publikacji (Przepiórkowski, 2008).

ROZDZIAŁ 3

Płytki parser *puddle*

W tym rozdziale przedstawiona zostanie koncepcja płytkiego parsera *puddle* (nazywanego w dalszej części rozdziału parserem *puddle* bądź po prostu parserem), który zaimplementowany został w ramach niniejszej pracy magisterskiej. Omówimy zasadę działania parsera *puddle* oraz przedstawimy formalizm reguł wykorzostywany przez ten analizator składniowy.

3.1. Reguły płytkiego parsera *puddle*

Działanie parsera *puddle* oparte jest na regułach parsingu. W niniejszej pracy jako regułę parsingu rozumiemy opis rozpoznania pewnej struktury języka naturalnego. Opis ten określa elementy składowe wspomnianych struktur, a także związki między tymi elementami (np. związek zgody pewnych kategorii gramatycznych, nadrzędność wskazanego elementy struktury w stosunku do pozostałych, etc.). Parser *puddle* na podstawie reguł parsingu generuje struktury, których opis stanowią reguły.

Parser udostępnia mechanizmy związane z przetwarzaniem tekstu wejściowego i generowaniem struktury składniowej tekstu, jednak sam parser nie posiada żadnej wiedzy na temat tego, w jaki sposób tworzyć strukturę składniową analizowanych zdań. Informacje o strukturze składniowej, jaką ma generować parser, są zawarte w regułach parsingu. Podejście takie służy oddzieleniu wiedzy lingwistycznej (na temat struktury języka, która jest odzwierciedlana w strukturze składniowej generowanej przez parser) od samego mechanizmu analizatora składniowego. Takie podejście czyni parser bardziej uniwersalnym, niż gdyby parser posiadał ściśle określony sposób analizowania i parsowania tekstu.

Reguły parsingu są zapisywane przy użyciu formalizmu opisywanego przez gramatykę bezkontekstową. Formalizm reguł wykorzystywany przez parser *puddle* jest oparty na formalizmie reguł parsera *Spejd* (patrz 2.2), który przedstawiony został w (Buczyński, Przepiórkowski, 2008).

3.1.1. Formalizm reguł

Formalizm reguł parsera *puddle* opisać można przy pomocy gramatyki bezkontekstowej. W niniejszym podrozdziale przedstawiony zostanie opis tego formalizmu przy użyciu rozszerzonej notacji Backusa-Naura[1].

[1]Specyfikacja formalizmu zgodnie ze standardem *Extended BNF* opisanym w (ISO/IEC 14977).

reguła = *nagłówek reguły* , *wzorzec reguły* , *akcje* ;

nagłówek reguły = ‘Rule‘ , ‘"‘ , *nazwa reguły* , ‘"‘ ;

nazwa reguły = {*znak*} − ;

wzorzec reguły = [*kontekst lewy*] , *dopasowanie* , [*kontekst prawy*] ;

kontekst lewy = ‘Left : ‘ , *specyfikacja* , “; “ ;

dopasowanie = ‘Match : ‘ , *specyfikacja* , “; “ ;

kontekst prawy = ‘Right : ‘ , *specyfikacja* , “; “ ;

akcje = ‘Eval : ‘, *akcja* − ;

specyfikacja = [“(“] , {*element*{[“|“] , *element*}}−, [“)“], [*kwalifikator*] ;

element = [“(“] , *token* | *grupa* | *symbol specjalny* , [“)“] , [*kwalifikator*] ;

token = “[“ , *warunek tokenu* , “]“ ;

grupa = “[“ , *warunek grupy* , “]“ ;

symbol specjalny = [“!“] , ‘sb‘ | ‘se‘ | ‘ns‘ ;

warunek tokenu = *atrybut* , *operator* , *wartość* , [‘/i‘] , {“&&“ , *atrybut* , *operator* , *wartość*[‘/i‘]} ;

warunek grupy = (‘type‘ , “ = “ | “! = “ , *identyfikator*)
| (‘head‘ , ‘ = ‘, *token*)
| (‘type‘, “ = ‘ | “! = “ , *identyfikator* , “&&“ , ‘head‘ , ‘ = ‘ , *token*)
| (‘head‘ , ‘ = ‘ , *token* , “&&“ , ‘type‘ , “ = “ | “! = “ , *identyfikator*) ;

atrybut = {*mała litera*} − ;

operator = “~“ | “~~“ | “!~“ ;

wartość = ‘"‘ , {*znak*} − , ‘"‘ ;

identyfikator = {*wielka litera*} − ;

kwalifikator = “ ∗ “ | “ + “ | “?“ ;

akcja = (‘group‘ , “(“ , *identyfikator grupy* , “, “ , *nr elementu* , “)“, “; “)
| (‘delete‘ , “(“ , *warunek tokenu* , “, “ , *nr elementu* , “)“ , “; “)
| (‘add‘ , “(“ , *interpretacja* , “, “ , *nr elementu* , “)“, “; “)
| (‘unify‘ , “(“ , *atrybut* , {*atrybut*} , “, “ , *nr elementu* , {“, “ , *nr elementu*} , “)“ , “; “)
| (‘syntok‘ , “(“ , *interpretacja* , “)“ , “; “) ;

interpretacja = {*znak*} − ;

nr elementu = *cyfra* − “0“ , {*cyfra*} ;

cyfra = “0“ | “1“ | “2“ | “3“ | “4“ | “5“ | “6“ | “7“ | “8“ | “9“ ;

znak = ? dowolny znak, który można wprowadzić z klawiatury ? ;

mała litera = ? każda z małych liter alfabetu, minuskuła ? ;

wielka litera = ? każda z wielkich liter alfabetu, majuskuła ? ;

3.1.2. Zmiany w stosunku do formalizmu *Spejd*

W stosunku do formalizmu *Spejd* w formalizmie reguł parsera *puddle* wprowadzono szereg zmian. Zmiany te polegają na rozszerzeniu formalizmu o dodatkowe elementy, zmianie istniejących w formalizmie *Spejd* rozwiązań, bądź rezygnacji z pewnych koncepcji z formalizmu *Spejd*.

W formalizmie reguł parsera *puddle* dodaliśmy możliwość wprowadzenia negacji wystąpienia symboli specjalnych takich, jak początek czy koniec zdania. W formalizmie opisywanym w tym rozdziale dodany został także operator `!=` w warunkach dotyczących grup.

Przyjęliśmy, że z punktu widzenia rozwiązania opisywanego w niniejszej pracy, nie jest konieczne wprowadzania rozróżnienia na głowę syntaktyczną oraz głowę semantyczną, jakie zostało przyjęte przez autorów parsera Spejd. W opisywanym tu podejściu przyjmujemy, że grupa posiada tylko jeden element wyróżniony – głowę. Jest to element, który jest nadrzędny w stosunku do innych elementów wchodzących w skład grupy. Własności morfologiczne głowy są wykorzystywane podczas ujednoznacznia wartości kategorii gramatycznych między grupą a innymi elementami parsowanego zdania oraz podczas usuwania interpretacji morfologicznych.

Operacja dodawania interpretacji morfologicznych (akcja *add*) została przez nas potraktowana nieznacznie inaczej niż w formalizmie *Spejd*. W formalizmie reguł parsera *puddle* podczas dodawania interpretacji morfologicznych określa się jedynie wartości kategorii gramatycznych bez modyfikowania formy bazowej.

W parserze *puddle* jako tokeny składniowe rozumiemy elementy struktury składniowej, które składają się z kilku wyrazów tworzących całość posiadającą określoną interpretację morfologiczną i rolę w strukturze składniowej. Zrezygnowaliśmy z operacji *word* dostępnej w formalizmie *Spejd*. Zamiast tego wprowadziliśmy akcję *syntok*, która ze wskazanych elementów zdania tworzy token składniowy z przypisaną określoną interpretacją morfologiczną.

3.1.3. Przykładowe reguły

Przedstawimy kilka reguł zapisanych w omawianym formalizmie wraz z omówieniem cech charakterystycznych formalizmu. Reguły przedstawiane w tej sekcji pochodzą ze zbioru reguł parsingu dla języka francuskiego stworzonych przez autora pracy. Reguły zostały uproszczone dla celów prezentacji w tej części pracy. Wspomniany zbiór reguł zostanie szerzej omówiony w rozdziale piątym.

```
Rule ''AP1.4''
Match: [pos~~''A''];
Eval: group(AP, 1);
```

Powyższa reguła o nazwie `AP1.4`, po dopasowaniu tokenu, którego częścią mowy (wyróżniony atrybut `pos`) jest `A` (przyjęte przez autora oznaczenie przymiotnika), tworzy z tego tokenu grupę o identyfikatorze `AP` (przyjęte oznaczenie frazy przymiotnikowej – ang. *adjective phrase*). Dopasowywany token posiada tylko interpretacje przymiotnika – zastosowany został operator `~~` oznaczający, że wszystkie interpretacje tokenu muszą spełniać warunek. Głową tej grupy jest element o numerze `1` czyli jedyny dopasowywany w przypadku tej reguły token.

```
Rule ''est''
Left: [base~''ne''];
Match: [orth~''est''/i];
Right: [base~''pas''];
Eval: delete(pos!~''V'', 2);
```

Reguła o nazwie `est` służy do ujednoznaczniania wyrazu *est*. Dopasowuje się ona do tokenu, którego formą ortograficzną jest `est` (nie jest istotna wielkość liter w zapisie tego wyrazu – modyfikator `/i`), bezpośrednio przed którym (kontekst lewy) znajduje się token, którego formą bazową jest `ne` (ściślej: token, który posiada taką interpretację morfologiczną, w której formą bazową jest `ne` – zastosowano operator `~`), oraz po którym znajduje się token, którego formą bazową jest `pas`. Po dopasowaniu tokenu, usunięte zostaną wszystkie interpretacje morfologiczne drugiego dopasowanego elementu (token `est`), które nie reprezentują czasownika (przyjęte oznaczenie `V`).

```
Rule ''compound conjunction''
Match [base~''à''] [base~''condition''] ([base~''que''] | [base~''de'']);
Eval: syntok(''csub'');
```

Reguła `compound conjunction` służy do wyróżniania spójników wielowyrazowych *à condition que* oraz *à condition de*. Wzorcem dopasowania reguły jest ciąg tokenów o formach bazowych `à` oraz `condition`, po których znajduje się token, którego formą podstawową jest `que` bądź `de`. Z dopasowanych elementów stworzony zostaje token składniowy (*syntoken*), którego interpretacją jest `csub` (przyjęte oznaczenie spójników wprowadzających zdanie podrzędne).

```
Rule ''NP1.1''
Match: [pos~''art'']? [type=AP] [pos~''N''] [type=AP];
Eval: unify(number gender, 1, 2, 3, 4);
group(NP, 3);
delete(pos!~''art'', 1);
delete(pos!~''N'', 3);
```

Reguła `NP1.1` tworzy podstawową grupę rzeczownikową. Dopasowane zostają dwie frazy przymiotnikowe (identyfikator `AP`), między którymi znajduje się rzeczownik (oznaczenie części

mowy `N`). Przed tymi elementami może znaleźć się także rodzajnik (oznaczenie `art`; modyfikator `?` oznacza, że znajdujący się przed nim element, może wystąpić w dopasowanym zdaniu lub nie). Zauważmy, że warunki dotyczące grup (w przykładzie jest to warunek dotyczący frazy przymiotnikowej) są opisywane w formalizmie reguł parsera *puddle* w inny sposób niż warunki dotyczące tokenów. Między dopasowanymi elementami zostają ujednolicone wartości liczby i rodzaju gramatycznego (atrybuty `number` i `gender`). Następnie z dopasowanych elementów stworzona zostaje grupa rzeczownikowa (identyfikator `NP`), której głową jest dopasowany rzeczownik (3. element w specyfikacji dopasowania). Kolejną czynnością jest usunięcie interpretacji pierwszego elementu, które nie są interpretacjami rodzajnika, oraz interpretacji trzeciego elementu, które nie odpowiadają rzeczownikowi.

3.2. Akcje parsera

Parser *puddle* jest wyposażony w zbiór akcji umożliwiających wykonywanie operacji na parsowanym tekście. Każdej akcji można przekazać wartości, które będziemy nazywać *parametrami* akcji. Parametry akcji są specyficzne dla rodzaju akcji – mogą one określać elementy, na których akcja będzie wykonywana, określać warunek akcji lub inne wartości charakterystyczne dla danej akcji.

Przed wykonaniem akcji zostaje wykonane testowanie warunków reguły. Polega ono na sprawdzeniu, czy możliwe jest wykonanie danej akcji na elementach dopasowanych do reguły, np. czy dany element posiada takie interpretacje morfologiczne, które może usunąć akcja reguły lub czy między danymi elementami można wykonać uzgodnienie wartości wskazanych atrybutów. Jeżeli warunki zostaną spełnione, następuje wykonanie akcji określonych w regule.

Parser umożliwia wykorzystanie pięciu rodzajów akcji. Są to:

- group,
- delete,
- add,
- unify,
- syntok.

3.2.1. Akcja `group`

Akcja `group` tworzy z elementów dopasowanych do wzorca dopasowania reguły jednostkę wyższego rzędu – grupę. Grupa posiada jeden element wyróżniony – głowę.

Parametrami akcji są identyfikator tworzonej grupy (przyjęto, że identyfikatory grup zapisywane są przy użyciu wielkich liter) oraz numer elementu, który ma być głową tej grupy.

3.2.2. Akcja `delete`

Akcja `delete` służy do usuwania interpretacji morfologicznych wskazanych elementów. Przy wykorzystaniu tej akcji można m. in. dokonywać ujednoznaczenia morfologicznego (dezambiguacji morfologicznej).

Parametrem akcji `delete` jest warunek, określający jakie interpretacje morfologiczne mają zostać usunięte, oraz numer elementu, do którego odnosi się akcja usuwania.

3.2.3. Akcja `add`

Przy użyciu akcji `add` można dodawać do wskazanych elementów nowe interpretacje morfologiczne.

Parametrami tej akcji są wzorzec interpretacji, jakie mają zostać dodane do elementu oraz numer wskazujący element, którego dotyczy działanie akcji.

3.2.4. Akcja `unify`

Akcja `unify` umożliwia ustalanie wspólnych wartości (*unifikację, ujednolicenie*) żądanych atrybutów dla wskazanych elementów dopasowanych do reguły. Akcja ta może mieć zastosowanie np. w przypadku kongruencji (związków zgody) zachodzących między rzeczownikami a określającymi je wyrazami (np. przymiotnikami).

Jako parametry tej akcji określa się zbiór atrybutów, których ma dotyczyć unifikacja, a także zbiór numerów elementów, dla których te atrybuty mają zostać zunifikowane.

3.2.5. Akcja `syntok`

Celem akcji `syntok` jest tworzenie z elementów dopasowanych do reguły większych całostek składniowych zwanych syntokenami[2] bądź tokenami składniowymi. Dla przykładu, francuskie wyrażenie *aujourd'hui* (*dzisiaj*) składa się z dwóch tokenów: *aujourd'* oraz *hui*. Ponieważ jednak wyrażenie to stanowi pewną całość znaczeniową, wydaje się być korzystne traktowanie całego tego wyrażenia jako jednej całości. Innym uzasadnieniem takiego podejścia może być np. występowanie w wielu językach spójników wielowyrazowych takich jak np. francuski *à condition de* i jego polski odpowiednik *pod warunkiem, że*.

Parametrem akcji `syntok` jest wzorzec interpretacji morfologicznej, jaka zostanie przypisana syntokenowi stworzonemu w wyniku działania tej akcji.

[2] od ang. *syntactic token*

3.3. Reprezentacja parsowanego tekstu

Tekst, na którym działa parser *puddle*, posiada dwie reprezentacje w wewnętrznej strukturze parsera.

3.3.1. Łańcuch tekstowy

Jedną z wewnętrznych reprezentacji zdania, jaką wykorzystuje parser *puddle*, jest reprezentacja w postaci łańcuchu tekstowego. Wszystkie elementy wchodzące w skład zdania (wyrazy, znaki interpunkcyjne czy np. tworzone w trakcie działania parsera frazy) zamienione zostają na łańcuch tekstowy (napis) zawierający informacje na temat form ortograficznych i informacji morfologicznych tych elementów.

Szczegółowe omówienie sposobu przekształcania zdania wejściowego na tę reprezentację znajdzie się w rozdziale 5.

Reprezentacja w postaciu łańcucha tekstowego jest wykorzystywana podczas dopasowywania reguł parsingu do analizowanego tekstu. Reguły są kompilowane do postaci wyrażeń regularnych, które są następnie dopasowywane do łańcucha tekstowego reprezentującego zdanie. Takie rozwiązanie ma na celu szybkie dopasowywanie reguł do tekstu.

3.3.2. Struktura obiektowa

W trakcie działania parsera, tworzona jest stuktura składniowa analizowanego zdania. Elementami struktury składniowej są *jednostki* (ang. *entities*), które tworzą drugą reprezentację wewnętrzną analizowanego zdania. Ta reprezentacja przedstawia właściwą strukturę składniową zdania i w oparciu o nią parser generuje swoje wyjście.

Parser wyróżnia cztery typy jednostek odpowiadające szczególnym elementom struktury składniowej. Każda z nich posiada specyficzne własności charakterystyczne dla danego rodzaju jednostek.

Wyróżnionymi rodzajami jednostek w parserze *puddle* są:

- *token*,
- grupa (*group*),
- syntoken (*syntok*),
- jednostka specjalna (*special entity*).

Jednostka token

Token jest jednostką odpowiadają każdemu tokenowi analizowanego tekstu wejściowego – może to być wyraz, znak interpunkcyjny, liczba, bądź jakikolwiek inny element wyróżniony

w trakcie tokenizacji (parser *puddle* zakłada, że podział tekstu na tokeny został wykonany przed rozpoczęciem parsingu i jego działanie jest niezależne od tokenizacji).

Każdy token posiada dwie główne właściwości: formę ortograficzną czyli informację o postacji graficznej tokenu, w jakiej pojawia się on w tekście, oraz zbiór interpretacji morfologicznych. Każda interpretacja morfologiczna zawiera informacje o formie bazowej oraz o właściwej interpretacji morfologicznej. Forma bazowa to forma podstawowa wyrazu, któremu odpowiada dana informacja morfologiczna. Na interpretację morfologiczną składają się także informacja o części mowy oraz o wartościach odpowiednich atrybutów, jeżeli dla danej części mowy takie występują.

Dla przykładu francuski wyraz o formie ortograficznej *fait* posiada następujące interpretacje morfologiczne:

- forma 3. osoby liczby pojedynczej trybu oznajmującego (*indicatif*) czasu teraźniejszego (*présent*) czasownika *faire*[3] (*robić*),
- forma imiesłowu przeszłego (*participe passé*) pochodzącego od czasownika *faire*,
- forma liczby pojedynczej rzeczownika *fait* (*czyn, uczynek, wydarzenie, fakt*) występującego w rodzaju męskim.

Jednostka grupa

Grupa jest jednostką reprezentującą jednostki wyższego rzędu tworzone podczas analizy składniowej zdania, a zatem np. frazy rzeczownikowe, grupy czasownikowe itd.

Do grupy jest przypisana informacja o typie struktury, jaką ona opisuje, oraz informacje o elemencie wyróżnionym w obrębie grupy (głowie grupy). Ponadto, grupa zawiera także informacje o pozostałych elementach niższego rzędu (a więc tokenach, ale także innych grupach) wchodzących w skład struktury opisywanej przez tę jednostkę.

Jednostka syntoken/słowo

Jednostka *syntoken* odpowiada tokenom składniowym, o których wspominaliśmy w sekcji 3.2.5.

Syntoken jest w pewnym sensie szczególną odmianą jednostki token. Właściwościami syntokenu są zatem również forma ortograficzna oraz zbiór interpretacji morfologicznych – analogicznie, jak to miało miejsce w przypadku jednostki token. Ponadto, jednostka syntoken zawiera także informacje o elementach, które wchodzą w skład opisywanego przez jednostkę tokenu składniowego.

[3]Forma *fait* jest jednocześnie także formą tworzącą czasy złożone czasownika *faire*, m. in. czasy *passé composé* oraz *plus-que-parfait*, jednak dla jasności przykładu, pominięto tę kwestię.

Jednostka specjalna

Jednostka specjalna jest to jednostka, która opisuje w strukturze składniowej zdania pewne szczególne elementy. Jednostka ta może określać początek zdania (jednostka typu *sb - sentence begin*) bądź koniec zdania (jednostka typu *se - sentence end*). W pewnych wyjątkowych sytuacjach, zależących od podziału tekstu na tokeny, jeżeli między dwoma tokenami nie ma pojawić się odstęp, to w strukturze zdania pojawi się jednostka typu *ns* (*no space*).

Poza informacją o typie, jednostka specjalna nie posiada innych właściwości.

3.4. Działanie parsera

Działanie parsera *puddle* odbywa się w następujących głównych krokach:

1. Załadowanie reguł parsingu,
2. Wczytanie tekstu wejściowego,
3. Dopasowywanie reguł – jeżeli reguła zostanie dopasowana, to następują dalsze kroki:
   (a) Testowanie warunków reguły,
   (b) Wykonanie akcji reguły,
4. Wygenerowanie wyjściowej struktury składniowej.

3.4.1. Załadowanie reguł parsingu

W pierwszym etapie działania, parser wczytuje reguły parsingu dostarczone przez użytkownika. Podczas wczytywania reguł, następuje sprawdzenie zgodności reguł z formalizmem. Następnie parser dokonuje kompilacji wzorców dopasowania reguł do postaci wyrażeń regularnych, a także przypisania akcjom reguł odpowiednich parametrów. Szczegóły dotyczące kompilacji wzorców dopasowania do wyrażeń regularnych znajdą się w rozdziale 5. dotyczącym implementacji parsera *puddle*.

3.4.2. Wczytanie tekstu wejściowego

Po załadowaniu reguł, parser wczytuje tekst wejściowy. Parser dokonuje analizy składniowej tekstu zdanie po zdaniu. Wejściowe zdanie zostaje zamienione na reprezentację w postaci łańcucha tekstowego. Zostaje również stworzona początkowa struktura składniowa analizowanego zdania, która będzie przetwarzana podczas dalszego działania parsera.

3.4.3. Dopasowywanie reguł

Po czynnościach wstępnych, wczytaniu reguł i tekstu wejściowego, następuje właściwe działanie parsera. Każda z reguł jest dopasowywana do analizowanego zdania wejściowego. Dopasowanie reguły polega na sprawdzeniu, czy analizowane zdanie w postaci łańcuchu tekstowego pasuje do wyrażenia regularnego będącego skompilowanym wzorcem dopasowania danej reguły. Jeśli dopasowanie reguły się powiedzie, następują dalsze kroki przetwarzania reguły.

Testowanie warunków reguły

Podczas dopasowania reguły do zdania, wyznaczone zostają elementy zdania odpowiadające dopasowaniu reguły (wraz z ewentualnymi kontekstami lewym i prawym). Na tych elementach wykonane zostaje testowanie warunków reguły. Testowanie to polega na sprawdzeniu, czy możliwe jest wykonanie wszystkich akcji związanych z regułą na elementach dopasowanych do wzorca reguły. Jedynie w sytuacji, gdy możliwe jest wykonanie wszystkich tych akcji, następuje ostatni krok: wykonanie akcji reguły.

Wykonanie akcji reguły

Po zakończonym sukcesem przetestowaniu warunków reguły, parser wykonuje kolejno akcje związane z daną regułą na elementach dopasowanych do reguły. Po zakończeniu działań określonych przez akcje reguły, zmodyfikowane zostają obie struktury reprezentujące analizowane przez parser zdanie. Następuje powrót do kroku 3 – parser poszukuje dalszych dopasowań danej reguły w strukturze parsowanego zdania zaktualizowanej po zmianach w wyniku dotychczasowych działań związanych z zastosowaniem reguły.

3.4.4. Wygenerowanie wyjściowej struktury składniowej

Po zastosowaniu wszystkich reguł, które można dopasować do analizowanego zdania, parser przystępuje do wygenerowania struktury składniowej danego zdania. Reprezentacja powstała w wyniku stosowania reguł parsingu zostaje przekształcona do formatu XML oraz zostaje przedstawiona w postaciu grafu. Szczegóły dotyczące przedstawienia wyników płytkiego parsingu zostaną omówione w rozdziale 5.

ROZDZIAŁ 4

Pozyskanie zasobów leksykalnych

W rozdziale omówione zostaną zasoby leksykalne wykorzystywane podczas tworzenia płytkiego parsera *puddle*, a także zasoby, które wykorzystuje on podczas swego działania. Przedstawiony zostanie słownik form fleksyjnych zebrany na potrzeby parsera *puddle*. Omówimy również zbiory tekstów (korpusy), które wykorzystywane były podczas rozwijania parsera oraz do ewaluacji wyników płytkiego parsingu.

4.1. Słownik form fleksyjnych

Jak zostało wspomniane w sekcji 1.4 niniejszej pracy, język francuski wykazuje stosunkowo ograniczoną morfologię. Z tego względu w parserze *puddle* zrezygnowaliśmy z przeprowadzania odrębnego procesu analizy morfologicznej. Informacje morfologiczne przyporządkowywane są do wyrazów na podstawie słownika form fleksyjnych. Słownik został stworzony poprzez kompilację dwóch istniejących i dostępnych publicznie słowników form fleksyjnych języka francuskiego: *Lefff* oraz *Lexique*.

4.1.1. Opracowanie słownika form fleksyjnych

Słownik *Lefff 2*[1] był generowany przy użyciu różnych metod automatycznych oraz korekty ręcznej. Szczegóły dotyczące tworzenia słownika znajdują się w artykule (Sagot i inni, 2006) oraz w pracy (Sagot, 2006, 131-153). Słownik *Lefff 2* zawiera 404483 formy fleksyjne (Sagot i inni, 2006, 2).

Słownik *Lexique 3* powstał w wyniku badań dotyczących częstości występowania niejednoznacznych i odmienionych form wyrazów francuskich w tekstach pisanych i relacjach mówionych. Szczegółowo badania, a także proces tworzenia słownika opisuje artykuł (New, 2006). Słownik *Lexique 3* zawiera 157000 form fleksyjnych (New, 2006, 4).

Dane z obu przedstawionych źródeł zostały połączone przez autora pracy. Punktem wyjścia był, ze względu na większy rozmiar, słownik *Lefff 2*. Do form ze słownika *Lefff 2* dodawano takie formy z *Lexique 3*, które w *Lefff 2* nie występowały lub takie leksemy z *Lexique 3*, które były pełniej opisane (posiadały więcej form fleksyjnych) w stosunku

[1] *Lefff* to skrót od franc. *Lexique des formes fléchies du français – leksykon francuskich form fleksyjnych.* Numer 2 oznacza nową wersję słownika. Wersja oznaczona 1 była wygenerowana w sposób automatyczny i została przedstawiona w (Clément i inni, 2004).

do *Lefff 2*. W wyniku tego scalenia źródeł otrzymano słownik form fleksyjnych dla parsera *puddle*, który zawiera 533747 form fleksyjnych pochodzących od 121247 form podstawowych.

4.1.2. Przyjęte oznaczenia

W tej części rozdziału przedstawimy przyjęte przez nas oznaczenia części mowy, kategorii gramatycznych oraz wartości, jakie mogą występować w obrębie tych kategorii. Takie same oznaczenia stosowane są w tagsecie dla parsera płytkiego oraz w zbiorze reguł parsingu stworzonych w ramach tej pracy magisterskiej, o których będzie mowa w sekcjach 5.2 i 5.3.

Oznaczenia kategorii gramatycznych

Należy nadmienić, że w niniejszej pracy rozumienie terminu kategorii gramatycznych jest trochę inne od rozumienia przedstawianego w podręcznikach gramatyki języka francuskiego. Z punktu widzenia niniejszej pracy, jako kategorie gramatyczne wyróżniamy jedynie te kategorie, dla których forma wyrazu zmienia się w zależności od wartości kategorii. Z tego względu w niniejszej pracy nie wyróżniamy kategorii stopnia przymiotnika i przysłówka, która występuje np. w gramatyce (Przestaszewski, 2006). Stopień wyższy (i odpowiednio niższy) w języku francuskim uzyskuje się przez poprzedzenie przymiotnika (bądź przysłówka) wyrazem *plus* (odpowiednio *moins*) zaś sam stopniowany wyraz nie ulega zmianie (w sensie ortograficznym). Analogicznie w stopniu najwyższym (najniższym) przed wyrazem należy dodać wyrażenie *la plus* (*la moins*). Ponieważ forma przymiotnika bądź przysłówka jest taka sama bez względu na stopień, nie traktujemy stopnia jako osobnej kategorii gramatycznej. Informacje dotyczące stopnia mają jednak znaczenie w parsingu płytkim, gdyż wspomnanie konstrukcje dotyczące stopniowania są uwzględniane w regułach parsingu.

W rozwiązaniu opisywanym w niniejszej pracy wyróżniliśmy sześć kategorii gramatycznych: rodzaj, liczbę, tryb, czas, osobę oraz czas imiesłowu. W poniższej tabeli prezentujemy oznaczenia przypisane do wyróżnionych kategorii wraz z wartościami, które mogą one przyjmować i oznaczeniami tych wartości.

Kategoria	Oznaczenie	Wartość	Oznaczenie
rodzaj	gender	męski	m
		żeński	f
liczba	number	pojedyncza	sg
		mnoga	pl

[2]Oznaczenia trybów i czasów czasowników zostały w dużej części zaczerpnięte z opisu w (Clément, 2001, 253-264).

Kategoria	Oznaczenie	Wartość	Oznaczenie
tryb[2]	mood	oznajmujący	I
		warunkowy	C
		rozkazujący	D
		subjonctif	S
		bezokolicznik	N
czas[2]	tense	teraźniejszy	P
		przyszły	F
		passé-simple	J
		imperfait	T
osoba	person	pierwsza	pri
		druga	sec
		trzecia	ter
czas imiesłowu	part-tense	przeszły	past
		teraźniejszy	present

Tabela 4.1.

Oznaczenia części mowy

Podczas tworzenia słownika form fleksyjnych na potrzeby parsera *puddle* wyróżniliśmy 17 części mowy. Zestawienie części mowy wraz z oznaczeniami oraz określeniem kategorii gramatycznych, jeżeli danej części mowy kategorie dotyczą, znajduje się w tabeli 4.2 znajdującej się poniżej.

Podobnie jak proponowany podziały kategorii gramatycznych, prezentowana tu klasyfikacja części mowy nie pretenduje do podziału części mowy języka francuskiego w rozumieniu lingwistycznym. Jest to podział wprowadzony przez autora na potrzeby konkretnego zastosowania przetwarzania języka naturalnego opisywanego w niniejszej pracy. Z tego względu np. traktujemy jako część mowy znaki interpunkcyjne, choć z punktu widzenia językoznawstwa częściami mowy one nie są.

Cześć mowy	Oznaczenie	Kategorie[3]
czasownik	V	mood
		tense
		number
		person

[4]Kategorie gramatyczne przedstawione są za pomocą oznaczeń określonych w tabeli 4.1.

Cześć mowy	Oznaczenie	Kategorie[4]
rzeczownik	N	gender number
przymiotnik	A	gender number
przysłówek	adv	
rodzajnik	det	gender number
rzeczownik odsłowny	grn	gender number
imiesłów	part	part-tense gender number
zaimek	pron	
zaimek względny	relpron	
zaimek pytający	intpron	
liczebnik	num	
przyimek	prep	
spójnik współrzędny	conj	
spójnik podrzędny	csub	
wykrzyknienie	excl	
znak interpunkcyjny	interp	
nieznana część mowy	ign	

Tabela 4.2.

4.2. Korpusy tekstów francuskojęzycznych

W tej części zaprezentujemy korpusy tekstów francuskojęzycznych, które były wykorzystywane podczas prac nad parserem *puddle*. Korpusy te były wykorzystywane podczas opracowywania reguł płytkiego parsingu. Na podstawie tych korpusów tworzyliśmy także listy fraz francuskojęzycznych.

Ponadto, zastosowanie dużych korpusów tekstów znajduje uzasadnienie w procesie tłumaczenia automatycznego, którego część stanowi płytki parser *puddle*. Opisywane w tej sekcji korpusy są korpusami równoległymi (dla języków francuskiego i polskiego). Na podstawie wyników płytkiego parsingu, system tłumaczący tworzy reguły tłumaczenia polsko-francuskiego oraz francusko-polskiego.

4.2.1. Korpus Unii Europejskiej

Korpusem Unii Europejskiej w tej pracy nazywamy korpus tekstów w języku polskim i francuskich pozyskanych z wielojęzycznego korpusu równoległego JRC–Aquis. Korpus JRC–Aquis zawiera teksty aktów prawnych Unii Europejskiej w 22 językach[5]. Korpus jest opracowany przez grupę Language Technology działającą w ramach Joint Research Centre funkcjonującego przy Komisji Europejskiej. Szczegóły działań związanych z tworzeniem korpusu jak i dotyczące samego korpusu znajdują się w publikacji (Steinberger i inni, 2006).

Z korpusu Unii Europejskiej uzyskaliśmy 4840479 zdań francuskich[6].

4.2.2. Korpus napisów filmowych

Jako korpus napisów filmowych w tej pracy rozumiemy korpus tekstów w języku francuskim i polskim pozyskanych z korpusu urównoleglonych napisów do filmów pochodzących z serwisu OpenSubtitles[7] udostępnionego w serwisie OPUS[8]. Korpus powstał w wyniku dopasowywania tekstów w 30 językach i jest dostępny w postaci zbiorów tekstów równoległych dla wskazanych par języków. Zatem podobnie jak w przypadku korpusu Unii Europejskiej, z opisywanego korpusu pozyskaliśmy zasób dwujęzyczny, polsko-francuski. Szczegółów dotyczących tworzenia korpusu dostarczają publikacje (Tiedemann, 2007a,b).

Z korpusu napisów filmowych uzyskaliśmy 1622105 zdań francuskich[9].

4.3. Bank drzew języka francuskiego

Bankiem drzew (ang. *treebank*) nazywamy korpus tekstów zawierający zdania, które zostały zanalizowane składniowo. Często (np. (Marcus i inni, 1993; Marciniak i inni, 2000)) zasoby te są zorganizowane w postaci drzew anotowanych składniowo, stąd nazwa *bank drzew*.

4.3.1. French Treebank

Dla celów niniejszej pracy wykorzystamy bank drzew języka francuskiego o nazwie French Treebank. Jest to korpus anotowany składniowo, który powstał na podstawie korpusu tekstów prasowych, które zostały oznaczone morfologicznie (wprowadzono oznaczenia części mowy, leksemów, etc.). Następnie przeprowadzono automatyczną analizę składniową tego korpusu.

[5] Są to teksty we wszystkich językach urzędowych Unii Europejskiej poza j. irlandzkim.

[6] Jak zostało wspomniane, korpus Unii Europejskiej traktujemy *de facto* jako korpus równoległy polsko-francuski i *vice versa*, dlatego można powiedzieć, że korpus Unii Europejskiej zawiera wspomnianą liczbę par równoległych zdań francuskich i polskich.

[7] `http://www.opensubtitles.org` [Dostęp 18 maja 2009].

[8] *OPUS - an open source parallel corpus*, `http://urd.let.rug.nl/tiedeman/OPUS/` [Dotęp 18 maja 2009].

[9] Uwaga jak w [5]

Wyniki parsingu były sprawdzane i poprawiane ręcznie. Szczegóły tworzenia banku drzew French Treebank omawia artykuł (Abeillé i inni, 2000).

W wyniku prac autorów banku drzew French Treebank powstał korpus anotowany morfologicznie-składniowo. Przyjęty został zbiór oznaczeń (tagset) morfologicznych oraz schemat oznaczenia funkcji składniowych, za pomocą którego dokonano anotacji składniowej w French Treebank (Abeillé i inni, 2000, 3, 5).

Bank drzew French Treebank zawiera 32000 anotowanych zdań, które zawierają 870000 słów (Abeillé i inni, 2003, 16).

4.3.2. Wykorzystanie banku drzew

W niniejszej pracy bank drzew języka francuskiego został wykorzystany do ewaluacji wyników płytkiego parsingu za pomocą rozwiązań tu opisywanych. W rozdziale szóstym zostanie pokazane, jak prezentuje się dokładność i poprawność struktury składniowej generowanej przez parser *puddle*, traktując jako punkt odniesienia (ang. *gold standard*) struktury składniowe pochodzące z French Treebank.

Przekształcenia danych pochodzących z banku drzew French Treebank, jakie zostaną wykonane na potrzeby ewaluacji zostaną omówione na początku rozdziału szóstego niniejszej pracy. Opisane zostaną tam również przyjęte miary oraz zastosowana procedura oceny skuteczności płytkiego parsera *puddle*.

ROZDZIAŁ 5

Omówienie implementacji

W tym rozdziale omówione zostaną szczegóły związane z implementacją płytkiego parsera *puddle*. Przedstawimy dane, na których opiera się działanie płytkiego parsera: tagset (zbiór oznaczeń morfologicznych) oraz zbiór reguł parsingu. Omówimy format danych wejściowych oraz wyjściowych parsera.

5.1. Szczegóły techniczne

Program *puddle* jest implementacją algorytmów parsera przedstawionych w rozdziale trzecim zrealizowaną w języku C++.

Działa on zarówno w systemach operacyjnych z rodziny Linux jak i w systemie operacyjnym Windows.

Pliki wykorzystywane przez parser oraz powstałe w wyniku jego działania wykorzystują system kodowania UTF-8[1].

5.2. Tagset

Działanie parsera *puddle* jest niezależne od oznaczeń morfologicznych przyjętych w parsowanym tekście. Przez oznaczenia morfologiczne rozumiemy oznaczenia części mowy oraz oznaczenia kategorii gramatycznych i przyjmowanych przez nie wartości (np. kategoria rodzaju gramatycznego przyjmuje w języku francuskim dwie wartości: rodzaj męski oraz rodzaj żeński). Parser *puddle*, dokonując konkretnej operaracji płytkiego parsingu danego tekstu, wykorzystuje zbiór oznaczeń morfologicznych dostarczony przez użytkownika.

5.2.1. Format tagsetu

Tagset jest przekazywany do parsera w postaci pliku, w którym zapisane są definicje oznaczeń morfologicznych, jakie są stosowane w tekście przeznaczonym do analizy składniowej. Jest to plik tekstowy o specyficznym formacie, przedstawionym poniżej.

Plik z opisem tagsetu jest podzielony na dwie części. W pierwszej części znajdują się definicje kategorii gramatycznych, a w drugiej określone są części mowy. Każda z części rozpoczyna się specjalnym znacznikiem. Są to odpowiednio znaczniki `[ATTR]` oraz `[POS]`.

[1] UTF-8 jest systemem kodowania standardu Unicode, który jest zgodny wstecz ze standardem ASCII. Specyfikację standardu UTF-8 można znaleźć w (UTF-8).

W pliku zawierającym opis tagsetu każda definicja znajduje się w jednej linii. W pliku tym można stosować komentarze. Komentarzem w pliku z opisem tagsetu jest dowolny ciąg znaków od symbolu komentarza # do końca linii.

Definicja kategorii gramatycznej zawiera oznaczenie kategorii oraz listę wartości, jakie dana kategoria może przyjmować.

Definicja części mowy składa się z oznaczenia części mowy oraz listy kategorii gramatycznych, które są przypisane do danej części mowy. Pewne kategorie mogą być nieobowiązkowe – takie kategorie umieszczone są w nawiasach kwadratowych. Jeżeli część mowy nie jest opisywana przez żadne kategorie gramatyczne, to lista kategorii pozostaje pusta.

Poniżej przedstawiamy fragment pliku z opisem tagsetu stworzonego na potrzeby parsera *puddle*. Jest to reprezentacja zbioru oznaczeń morfologicznych, który został omówiony w sekcji 4.1.2.

```
[ATTR]
number = sg pl
gender = m f
mood   = I C D S N
tense  = P F J T
person = pri sec ter

[POS]
N    = gender number
A    = gender number
V    = mood tense person number
num  =
prep =
conj =
csub =
```

W powyższym fragmecie pliku z tagsetem zdefiniowanych zostało pięć kategorii gramatycznych oraz siedem części mowy. Z trzema częściami mowy związane są kategorie, zaś cztery nie posiadają charakteryzujących je kategorii.

5.2.2. Załadowanie tagsetu

Załadowanie do pamięci opisu tagsetu z pliku wskazanego przez użytkownika jest pierwszą czynnością wykonywaną przez parser *puddle*. Dane dotyczące tagsetu są wykorzystywane w dalszych działaniach parsera.

Parser sprawdza, czy w regułach parsingu wykorzystywane są oznaczenia zgodne z tag-

setem. Dane z tagsetu wykorzystywane są także podczas procesu kompilowania reguł, który opisany został w kolejnej części tego rozdziału.

Ponadto, parser dokonuje także sprawdzenia, czy dane wejściowe są oznaczone zgodnie z oznaczeniami przyjętymi w załadowanym przez parser tagsecie.

5.3. Reguły

Podobnie, jak ma to miejsce w przypadku oznaczeń morfologicznych, parser *puddle* nie opiera swego działania na ściśle określonym zestawie reguł parsingu. Sam parser nie posiada żadnej wiedzy odnośnie struktury składniowej tekstu, który ma analizować – działanie parsera jest w pełni oparte na regułach dostarczonych przez użytkownika.

5.3.1. Format pliku reguł

Reguły parsingu parsera *puddle* zapisywane są w pliku tekstowym zgodnie z formalizmem reguł przedstawionym w sekcji 3.1.1.

Dodatkowo, w pliku reguł możliwe jest umieszczanie komentarzy. Podobnie jak w przypadku pliku z opisem tagsetu w pliku z regułami komentarzem jest dowolny ciąg znaków zaczynający się symbolem komentarza `#` do końca linii.

5.3.2. Kompilacja reguł

Załadowanie reguł jest działaniem wykonywanym po załadowaniu tagsetu.

Najbardziej istotną czynnością wykonywaną przez parser po wczytaniu reguł z pliku jest przetworzenie wzorca dopasowania reguły do wewnętrznej postaci wykorzystywanej przez parser *puddle*. Działanie to będziemy nazywać kompilowaniem wzorca reguły.

W parserze *puddle* wzorce reguł parsingu są wewnętrznie reprezentowane w postaci wyrażeń regularnych. Do tych wyrażeń regularnych dopasowywany jest tekst wejściowy w postaci łańcucha tekstowego. Jeżeli do łańcucha tekstowego reprezentującego analizowane zdanie zostanie dopasowane wyrażenie regularne, to parser dokonuje identyfikacji, które elementy zdania odpowiadają poszczególnym elementom wyszczególnionym we wzorcu dopasowania reguły. Na tak rozpoznanych elementach są następnie wykonywane akcje reguły.

Podczas wczytywania reguł i kompilowania wzorców reguł parser dokonuje także sprawdzenia, czy reguły dostarczone przez użytkownika są zgodne z przyjętym formalizmem reguł. Sprawdzana jest także zgodność oznaczeń używanych w regułach parsingu z oznaczeniami morfologicznymi określonymi w załadowanym wcześniej tagsecie.

Poniżej przedstawiamy kilka przykładów kompilowania wzorców dopasowania reguł do postaci wyrażeń regularnych wykorzystywanych przez parser. Dla uproszczenia prezentowane

zostaną tylko fragmenty reguł istotne z punktu widzenia przykładów. W przykładach, w każdej linii skompilowanego wyrażenia regularnego znajduje się fragment wyrażenia odpowiadający jednemu elementowi wzorca dopasowania reguły.

Część wyrażenia regularnego odpowiadającego każdemu elementowi wzorca zawiera się między znacznikami < oraz >. Wyrażenie to zawiera kilka części, które rozpoczynają się pojedynczym znacznikiem <. Analogicznie zbudowana jest reprezentacja tekstu w postaci łańcucha tekstowego, do której dopasowywane są wyrażenia regularne, które odpowiadają wzorcom reguł. Przyjęcie nawiasów ostrokątnych do oddzielania informacji w tej strukturze uzasadnia fakt, że w danych w plikach XML, z których korzysta parser *puddle*, te nawiasy pojawiają się tylko w formie zastąpionej przez oznaczenia `<` oraz `>`.

Przykład 5.1.

Wzorzec dopasowania reguły:
```
Match: [orth~''entre''];
Right: [pos~''prep''];
```

Wzorzec skompilowany do wyrażenia regularnego:
```
(<<t<[^<>]+<entre(<[^<>]+)+>)
(<<t<[^<>]+<[^<>]+(<[^<>]+)*(<L......[^<>]+)(<[^<>]+)*>)
```

Oznaczenie `<<t` określa, że dopasowany ma zostać token. Kolejną częścią reprezentacji tokenu w formie napisu jest identyfikator elementu, który w wyrażeniach regularnych, do których skompilowane zostają wzorce dopasowania reguł, może przyjąć wartość dowolną. Następna część reprezentacji tokenu to jego forma ortograficzna. W warunku dotyczącym pierwszego elementu wzorca reguły występuje `orth~''entre''`, dlatego w częściu wyrażenia regularnego dotyczącym pierwszego elementu także pojawia się ciąg *entre*. Ostatnią częścią reprezentacji tokenu jest ciąg reprezentacji intperpretacji morfologicznych oddzielonych separatorami <. W warunkach dotyczących pierwszego elementu wzorca reguły nie ma żadnych ograniczeń dotyczących interpretacji morfologicznych, dlatego wyrażenie regularne także nie zawiera żadnych ograniczeń dotyczących morfologii. Natomiast drugi element posiada warunek dotyczący części mowy `pos~''prep''`. W reprezentacji tokenu w postaci łańcucha tekstowego każda z informacji o jego części mowy oraz o wartościach kategorii gramatycznych jest przedstawiona w postaci jednego znaku. Przypisania wartości znaków odpowiadających danym częściom mowy oraz wartościom kategorii gramatycznych dokonuje parser podczas wczytywania tagsetu. W tym przypadku części mowy `prep` (przyimek) odpowiada oznaczenie `L`. Po stałej (określonej w tagsecie) liczbie wartości kategorii gramatycznych w wyrażeniu regularnym zawarte są informacje o formie bazowej. W przykładzie forma bazowa nie była określona w żadnym warunku, stąd w wyrażeniu regularnym także nie ma ograniczeń dotyczących formy bazowej. Co więcej, w warunku dotyczącym części mowy użyty został słabszy operator porównania

~, dlatego do części wyrażenia regularnego dotyczącej drugiego elementu wzorca reguły poza interpretacją morfologiczną, której częścią mowy jest przyimek, dopasowane mogą też zostać tokeny, które posiadają także inne interpretacje morfologiczne – dowolną interpretację morfologiczną w wyrażeniu regularnym oznacza ciąg `(<[^<>]+)*`.

Przykład 5.2.

Wzorzec dopasowania reguły:

```
Match: [pos~''adv'' && base!~''pas|tout'']+ [pos~''A''];
```

Wzorzec skompilowany do wyrażenia regularnego:

```
((<<t<[^<>]+<[^<>]+(<.......(?!pas|tout)[^<>]+)*(<G......(?!pas|tout)[^<>]+)
(<.......(?!pas|tout)[^<>]+)*>)+)
(<<t<[^<>]+<[^<>]+(<[^<>]+)*(<C......[^<>]+)(<[^<>]+)*>)
```

W warunku dotyczącym pierwszego elementu wzorca dopasowania występują ograniczenia dotyczące formy bazowej. Do części wyrażenia regularnego odpowiadającej pierwszemu elementowi dopasowane zostaną tylko takie tokeny, których forma bazowa jest różna od *pas* oraz *tout*, czyli zgodnie z warunkiem z wzorca reguły. Zauważmy, że poza warunkiem dotyczącym formy bazowej, we wzorcu reguły występuje też warunek określający, że jedna z interpretacji tokenu musi być przysłówkiem (oznaczenie *adv*). W wyrażeniu regularnym przysłówkowi odpowiada symbol `G`, jednak wystąpić on musi jedynie w jednej interpretacji. Mimo to, pozostałe interpretacje morfologiczne tokenu w dalszym ciągu nie mogą pochodzić od leksemu *pas* ani *tout*. W warunku dotyczącym drugiego elementu wzorca dopasowania określone jest ograniczenie dotyczące części mowy. Jedną z interpretacji elementu ma być przymiotnik (oznaczenie `A`) – w wyrażeniu regularnym przymiotnikowi odpowiada `C`.

Przykład 5.3.

Wzorzec dopasowania reguły:

```
Match: [pos~''conj''] [type=PP];
```

Wzorzec skompilowany do wyrażenia regularnego:

```
(<<t<[^<>]+<[^<>]+(<[^<>]+)*(<M......[^<>]+)(<[^<>]+)*>)
(<<g<[^<>]+<PP<[^<>]+<[^<>]+<[^>]+>)
```

Zauważmy, że drugim elementem wzorca dopasowania reguły jest grupa. Reprezentacja grupy w postaci łańcuchu tekstowego rózni się od reprezentacji tokenu. Oznaczeniem dopasowania grupy jest `<<g`. Kolejną częścią reprezentacji jest, tak jak w przypadku reprezentacji tokenu, identyfikator kolejnego elementu zdaniu, który w wyrażeniu regularnym może przyjąć wartość dowolną. Następną częścią reprezentacji grupy jest oznaczenie typu grupy, po których następuje ciąg reprezentacji interpretacji morfologicznych tokenu, który jest głową grupy. Interpre-

tacje reprezentowane są tak, jak ma to miejsce w przypadku reprezentacji tokenu. Zauważmy, że we wzorcu dopasowania reguły występuje warunek dotyczący typu grupy i do części wyrażenia regularnego dotyczącej drugiego elementu dopasowania dopasowane zostaną tylko grupy, które są typu *PP* (fraza przyimkowa).

5.3.3. Wykorzystywany zbiór reguł

Podczas prac związanych z niniejszą pracą magisterską, stworzony został zbiór reguł parsingu dla języka francuskiego zapisanych w formalizmie opisywanym w sekcji 3.1.1. Zbiór reguł obejmuje 250 reguł, wśród których znajduje się:

- 101 reguł tworzących frazy – od podstawowych fraz np. fraz rzeczownikowych, fraz przymiotnikowych do większych struktur reprezentujących całe zdanie,
- 17 reguł związanych ze spójnikami wielowyrazowymi,
- 132 reguły pomocnicze, związane m. in. z usuwaniem niejednoznaczności morfologiczno-składniowych.

Reguły parsingu stworzone przez autora są dołączone do pracy razem z parserem *puddle*. Zbiór reguł został przedstawiony w dodatku A.

5.4. Dane wejściowe

5.4.1. Format danych wejściowych

Dane wejściowe dla parsera *puddle* stanowi tekst w formacie XCES (XML Corpus Encoding Standard). XCES to format zapisu korpusów tekstu oparty na standardzie XML. Opis standardu XCES przedstawiony jest w (XCES).

Poniżej przedstawiamy zapis zdania *Je suis* (*Ja jestem*) w formacie XCES.

Przykład 5.4

```
...
<chunk type=''s''>
<tok>
<orth>Je</orth>
<lex><base>je</base><ctag>pron</ctag></lex>
<lex><base>je</base><ctag>N:f:sg</ctag></lex>
</tok>
<tok>
<orth>suis</orth>
<lex><base>être</base><ctag>V:D:P:sec:sg</ctag></lex>
<lex><base>être</base><ctag>V:I:P:sec:sg</ctag></lex>
<lex><base>être</base><ctag>V:I:P:pri:sg</ctag></lex>
<lex><base>suivre</base><ctag>V:D:P:sec:sg</ctag></lex>
<lex><base>suivre</base><ctag>V:I:P:pri:sg</ctag></lex>
<lex><base>suivre</base><ctag>V:I:P:sec:sg</ctag></lex>
</tok>
</chunk>
...
```

Opis w formacie XCES zawiera informacje o podziale tekstu na zdania (elementy `chunk`), a także informacje dotyczące poszczególnych tokenów (wyrazów, znaków interpunkcyjnych etc.) wyodrębnionych w zdaniu. Opis tokenu zawiera element tok. Opis każdego tokenu składa się z informacji o jego formie ortograficznej (tj. takiej, w jakiej występuje w zdaniu; element `orth`), a także z szeregu interpretacji morfologicznych (elementy `lex`). Interpretacja morfologiczna składa się z formy bazowej (element `base`) oraz z informacji morfologicznych związanych z daną interpretacją (element `ctag`).

Zdanie przedstawione w przykładzie 5.1 składa się zatem z dwóch tokenów. Formą ortograficzną pierwszego z nich jest *Je* i posiada on dwie interpretacje morfologicznie: zaimka *je* (*ja*) oraz rzeczownika *je* w formie liczby pojedynczej rodzaju żeńskiego. Drugi token o formie ortograficznej *suis* posiada interpretacje morfologiczne czasowników *être* (*być*) oraz *suivre* (*iść*, *naśladować*, *śledzić*) w formach pierwszej i drugiej osoby liczby pojedynczej trybu oznajmującego i rozkazującego czasu teraźniejszego. Szczegółowy opis oznaczeń morfologicznych stosowanych w rozwiązaniu omawianym w tej pracy znajduje się w sekcji 4.1.2.

5.4.2. Przetworzenie tekstu do postaci wewnętrznej

Przed rozpoczęciem parsingu, tekst wejściowy zapisany w formacie XCES zostaje przetworzony do postaci wewnętrznej wykorzystywanej przez parser. Do przetwarzania pliku XML zawierającego dane wejściowe parser *puddle* wykorzystuje parser języka XML Xerces[2].

Jak zostało opisane w sekcji 3.3, parser *puddle* wykorzystuje wewnętrznie dwie reprezentacje parsowanego tekstu. Są one tworzone podczas wczytywania danych wejściowych. Początkowa struktura obiektowa reprezentująca analizowane zdanie jest budowana na podstawie danych uzyskanych z wejścia parsera.

Jednocześnie, podczas wczytywania danych wejściowych, tworzona jest stuktura każdego analizowanego zdania w postaci łańcucha tekstowego. Struktura ta jest wykorzystywana do dopasowywania wyrażeń regularnych odpowiadających wzorcom reguł parsingu. Wykorzystanie takiej struktury służy szybkiemu dopasowywaniu wzorców reguł do analizowanego tekstu. Jest to pomysł podobny do koncepcji zastosowanej w płytkim parserze *Spejd* opisanej w (Przepiórkowski, Buczyński, 2007).

Początkowa struktura zdania w postaciu łańcucha tekstowego tworzona jest na podstawie informacji pobranych z wejściowego tekstu. Łańcuch reprezentujący zdanie zawiera informacje o wszystkich tokenach wchodzących w jego skład. Ponadto, dołączone są informacje o specjalnych symbolach *sb* i *se* reprezentujących odpowiednio początek oraz koniec zdania.

Dla przykładu, token z przykładu 5.1, którego formą ortograficzną jest *suis*, w postaci łańcucha tekstowego ma reprezentację:

```
<<t<2<suis<BA0CAB0être<BA0AAB0être<BA0AAA0être<BA0CAB0suivre
<BA0AAA0suivre<BA0AAB0suivre>
```

Reprezentacja tokenu w postaci łańcucha znakowego rozpoczyna się oznaczeniem `<<t` i jest zakończona znacznikiem `>`. Reprezentacja składa się z kilku części oddzielonych separatorami `<`. Pierwszą częścią jest identyfikator kolejnego elementu zdania – jest to numer elementu w zdaniu przedstawiony w postaci liczby szesnastkowej. Analizowany token jest drugim tokenem w zdaniu, stąd jego identyfikatorem jest `2`. Następnym elementem reprezentacji tokenu jest jego forma ortograficzna – tutaj jest to `suis`. Kolejne elementy reprezentacji tokenu w postaci łańcucha znakowego to reprezentacje jego interpretacji morfologicznych. Reprezentacja każdej interpretacji morfologicznej składa się z formy bazowej dla danej interpretacji poprzedzonej stałą liczbą znaków, z których każdy reprezentuje część mowy oraz wartości kategorii gramatycznych odpowiednich dla danej interpretacji. Reprezentacje części

[2]Xerces jest zestawem narzędzi do parsowania i przetwarzania języka XML. W implementacji parsera *puddle* wykorzystano mechanizm parsera XML opartego o mechanizm dostępu szeregowego SAX2. Mechanizm obsługi elementów dokumentu XML (*content handler*), który jest wykorzystywany przez parser XML został opracowany przez autora pracy. Opis pakietu Xerces znajduje się w (Xerces)

mowy (np. dla analizowanego tokenu reprezentacją czasownika jest `B`) oraz wartości kategorii gramatycznych są automatycznie generowane przez parser podczas ładowania tagsetu. Jeżeli w danej interpretacji morfologicznej nie są określone wartości pewnej kategorii gramatycznej, to wartość nieokreślona reprezentowana jest przez `0`.

Łańcuch tekstowy reprezentujący zdanie jest konkatenacją łańcuchów odpowiadających tokenom wchodzących w skład zdania. Zatem, zdanie z przykładu 5.1 w tej reprezentacji ma postać:

```
<<s<0<sb<>
<<t<1<Je<F000000je<AAB0000je>
<<t<2<suis<BA0CAB0être<BA0AAB0être<BA0AAA0être<BA0CAB0suivre
<BA0AAA0suivre<BA0AAB0suivre>
<<s<3<se<>
```

5.4.3. Wbudowany tagger

Na wejściu parsera *puddle* może pojawić się także czysty tekst, nieprzetworzony do formatu XCES. W takiej sytuacji parser przetwarza tekst wejściowy do formatu XCES przy wykorzystaniu modułu oznaczającego tekst, który będziemy nazywać taggerem.

Tagger wykorzystuje słownik form fleksyjnych języka francuskiego opisany w sekcji 4.1. Wyrazy, które znajdują się w słowniku form fleksyjnych, zostają oznaczone zgodnie z wpisami w słowniku. Ponadto tagger rozpoznaje znaki interpunkcyjne. Wyrazy i inne ciągi znaków, które nie figurują w słowniku form fleksyjnych są oznaczane przez tagger specjalnym oznaczeniem części mowy *ign*.

Po przetworzeniu wejściowego zdania do formatu XCES, zostaje one parsowane tak, jak w przypadku, gdy wejściem parsera jest plik w formacie XCES.

5.5. Wyjście parsera

Parser *puddle* generuje reprezentację struktury składniowej zdania w dwóch postaciach: reprezentacji w postaciu formatu XCES oraz reprezentacji w postaci grafu.

5.5.1. Wyjście w formacie XCES

Reprezentacja struktury składniowej wygenerowanej przez parser *puddle* jest zapisywana w formacie XCES. Struktura dokumentu XML zawierającego zdanie wraz z informacją o strukturze składniowej powstałej w wyniku płytkiego parsingu jest podobna do struktury danych wejściowych. Poza zmodyfikowanymi interpretacjami morfologicznymi, plik XML zawiera informacje o rozpoznanych w zdaniu grupach.

Zdanie z przykładu 5.1 po sparsowaniu ma w formacie XCES postać następującą:

```
...
<chunk type=''s''>
<group id=''6'' rule=''S1.1: Normal sentence'' head=''2''
type=''S''>
<group id=''4'' rule=''NP6.1: pronoun as noun phrase''
head=''1'' type=''NP''>
<tok id=''1''>
<orth>Je</orth>
<lex><base>je</base><ctag>pron</ctag></lex>
</tok>
</group>
<group id=''5'' rule=''VP2.1: general verbal phrase
(non-infinitive)'' head=''2'' type=''VP''>
<tok id=''2''>
<orth>suis</orth>
<lex><base>être</base><ctag>V:D:P:sec:sg</ctag></lex>
<lex><base>être</base><ctag>V:I:P:sec:sg</ctag></lex>
<lex><base>être</base><ctag>V:I:P:pri:sg</ctag></lex>
<lex><base>suivre</base><ctag>V:D:P:sec:sg</ctag></lex>
<lex><base>suivre</base><ctag>V:I:P:pri:sg</ctag></lex>
<lex><base>suivre</base><ctag>V:I:P:sec:sg</ctag></lex>
</tok>
</group>
</group>
</chunk>
...
```

Zauważmy, że w wyniku parsingu, w strukturze zdania przedstawionej w formace XCES każdemu elementowi zdania został nadany identyfikator (atrybut `id`). Informacje dotyczące grupy zawierają elementy `group`. Każdy z tych elementów posiada informacje o typie grupy (atrybut `type`), o głowie grupy (atrybut `head`) oraz o regule, która została zastosowana do rozpoznania danej grupy (atrybut `rule`).

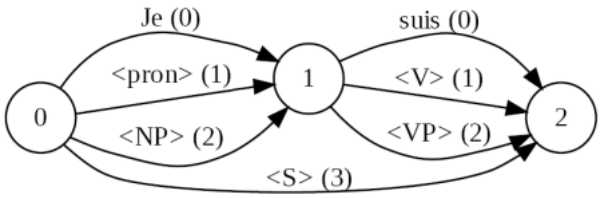
Warto zauważyć także, że na wyjściu parsera jedyną interpretacją morfologiczną tokenu *je* jest interpretacja jako zaimek. Inna interpretacja została usunięta w procesie parsingu – parser dokonał zatem także ujednoznaczenienia morfologicznego.

5.5.2. Wyjście w postaci grafu

Poza reprezentacją w pliku XML, parser generuje też reprezentację struktury składniowej analizowanego zdania w postaci grafu.

Elementy struktury składniowej zdania reprezentowane są przez krawędzie grafu. Do krawędzi grafu wychodzących z tego samego wierzchołka przypisane są liczby naturalne, które nazywamy *głębokością*. Najniższe (*najgłębsze*) są krawędzie zawierające informacje o formie ortograficznej tokenu. Drugimi pod względem głębokości krawędziami są krawędzie, do których przypisana jest informacja o części mowy. Kolejne krawędzie odpowiadają grupom rozpoznanym podczas analizy składniowej. Grupom będącym elementami struktury składniowej wyższego rzędu odpowiadają krawędzie grafu o odpowiednio wyższym parametrze głębokości.

Poniżej przedstawiamy graf wygenerowany dla zdania z przykładu 5.1.



Motywacją dla wykorzystania struktury grafowej do przedstawiania wyników analizy składniowej jest użycie parsera *puddle* w procesie tłumaczenia automatycznego. Program tłumaczący operuje na grafie w poszukiwaniu najlepszego tłumaczenia wejściowego zdania, dlatego parser *puddle* generuje reprezentację struktury składniowej także w postaci grafu.

Ponadto, parser *puddle* zapisuje graf struktury składniowej ostatniego analizowanego zdania w postaci pliku w formacie DOT[3].

5.6. Parametry i opcje parsera

Parser *puddle* może pobierać dane wejściowe z wskazanego pliku w formacie XML i generować wyjście do wskazanego pliku. Plik wejściowy określa się za pomocą parametru uruchomieniowego `--file`. Plik wyjściowy można określić za pomocą parametru `--xml-out`. Jeżeli plik wyjściowy nie zostanie określony, to wyjście parsera zostanie zapisane w pliku `morphSh.xml` w katalogu, w którym znajduje się plik wykonywalny parsera.

Inną możliwością uruchomienia parsera *puddle* jest wykorzystywanie standardowego wejścia i standardowego wyjścia. Parser pracuje w tym trybie, jeżeli zostanie uruchomiony bez określenia pliku wejściowego. W tym trybie, parser odczytuje zdanie ze standardowego wejścia, oznacza ja pomocą wbudowanego taggera (zostało to opisane w sekcji 5.4.3) i generuje wyjście w formacie XCES na standardowe wyjście.

Ponadto, parser wykorzystuje opcje konfiguracyjne zapisywane w pliku `config.ini` w ka-

[3]Język DOT językiem opisu grafów. Program *dot*, wchodzący w skład pakietu Graphviz, umożliwia drukowanie grafów zapisanych w języku DOT do dokumentów w formacie PostScript oraz PDF. Specyfikacja języka DOT oraz opis programu *dot* znajduje się w (DOT).

talogu, w którym znajduje się plik wykonywalny parsera. Podstawowymi opcjami są ścieżki dostępu do plików zawierających tagset oraz reguły parsingu. Za pomocą tych opcji można określić również pliki, w których zapisywane będą dane diagnostyczne (oznaczenia morfologiczne z tagsetu oraz wzorce dopasowania reguł w wewnętrznej postaci używanej przez parser *puddle*, które zostają wygenerowane podczas uruchomienia parsera) zapisywane przez parser podczas ładowania pliku zawierającego tagset oraz pliku z regułami parsingu. Standardowo, parser zapisuje dane diagnostyczne odpowiednio w plikach `tagset.log` oraz `rules.log` w katalogu, w którym znajduje się plik wykonywalny parsera. Za pomocą opcji konfiguracyjnych można także określić plik zawierający zserializowany słownik form fleksyjnych wykorzystywany przez wbudowany moduł taggera.

5.7. Wykorzystane biblioteki

W implementacji parsera *puddle* wykorzystano następujące biblioteki programistyczne:

- Boost – biblioteka wykorzystana do obsługi wyrażeń regularnych, grafów, serializacji danych słownika form fleksyjnych, dokładnego pomiaru czasu wykonania programu oraz obsługi parametrów programu; dokumentacja biblioteki znajduje się w (Boost),
- ICU – biblioteka obsługi standardu Unicode; dokumentacja biblioteki znajduje się w (ICU),
- Xerces – pakiet do obsługi i parsowania języka XML; dokumentacja pakietu znajduje się w (Xerces).

ROZDZIAŁ 6

Interpretacja wyników i wnioski

W niniejszym rozdziale przedstawimy ewaluację płytkiego parsera *puddle*. Opiszemy przyjętą przez nas procedurę oceny wyników płytkiego parsingu w oparciu o bank drzew języka francuskiego. Następnie zaprezentujemy wyniki otrzymane w wyniku procedury ewaluacji oraz skomentujemy je, także w kontekście ograniczeń i wad samej procedury.

6.1. Procedura ewaluacji

Do oceny wyników generowanych przez płytki parser *puddle* wykorzystaliśmy bank drzew języka francuskiego opisany w sekcji 4.3. Ocenę oparliśmy na porównaniu struktur składniowych zdań reprezentowanych w postaci drzew wchodzących w skład banku drzew ze strukturami składniowymi generowanymi dla tych samych zdań przez parser *puddle*. Po odrzuceniu zdań, które stanowiły nazwiska autorów artykułów bądź tytułów prasowych, z których pochodziły artykuły składające się na bank drzew, do ewaluacji uzyskaliśmy 21564 zdania francuskie.

6.1.1. Ograniczenia przyjętej metody

Ze względu na to, że w wykorzystywanym do ewaluacji banku drzew przechowywane są pełne struktury składniowe zdań, natomiast parser *puddle* generuje jedynie strukturz powierzchniowe, nieuzasadnione byłoby przeprowadzenie oceny działania parsera *puddle* przez proste porównanie struktury składniowej danego zdania stworzonej przez parser ze strukturą wzorcową zawartą w banku drzew.

Zdecydowaliśmy się, aby oceniać wyniki parsera puddle poprzez sprawdzenie, z jaką skutecznością w generowanej przez niego strukturze składniowej poprawnie (w odniesieniu do banku drzew) rozpoznawane są podstawowe typy fraz. Analizowaliśmy skuteczność oznaczania przez parser *puddle* fraz rzeczownikowych i przyimkowych, a także fraz przymiotnikowych i czasownikowych. Frazę rozpoznaną przez parser jako frazę danego typu uznawaliśmy za rozpoznaną poprawnie, jeżeli w strukturze składniowej zawartej w banku drzew została ona oznaczona jako fraza tego samego typu.

Opisana powyżej metoda (nazywać ją będziemy dalej *metodą pierwszą*) okazała się być stosunkowo mało skutecznym sposobem na ocenę skuteczności parsera *puddle*. Wadę tego podejścia możemy zaobserwować, jeżeli przyjrzymy się następującemu fragmentowi zdania: *sur des normes de qualité* (*odnośnie norm jakości*). Wzorcowa struktura składniowa tego

fragmentu wygląda następująco (dla czytelności będziemy przedstawiać struktury składniowe w postaci *spłaszczonej*):

```
<PP> sur <NP> des normes <PP> de <NP> qualité </NP> </PP> </NP> </PP>
```

Natomiast parser *puddle* wygenerował dla tego fragmentu poniższą strukturę:

```
<PP> sur <NP> <NP> des normes </NP> <PP> de <NP> qualité </NP> </PP> </NP> </PP>
```

Jak widać, w strukturze wygenerowanej przez parser *puddle* wyodrębnionych zostało więcej fraz rzeczownikowych niż w strukturze wzorcowej z banku drzew. Wynika to z faktu, że parser płytki nie analizuje pełnej struktury zdania, stąd m. in. nie uwzględnia tego, w jaki sposób w zdaniu połączone są ze sobą frazy przyimkowe. Przyjęliśmy, że jeżeli w zdaniu parser *puddle* natrafi na frazę rzeczownikową, po której następuje fraza przyimkowa, to ta fraza przyimkowa zostaje dołączona do grupy rzeczownikowej.

Widać zatem, że struktury wzorcowa oraz zbudowana przez parser dla powyższego fragmentu zdania francuskiego się różnią. Z drugiej strony należy zauważyć, że cały fragment w obu przypadkach został rozpoznany jako fraza przyimkowa. Mając na uwadze powyższy komentarz dotyczący traktowania fraz przyimkowych przez parser płytki, nie można uznać struktury składniowej wygenerowanej przez parser *puddle* dla analizowanego fragmentu za niepoprawną.

Ze względu na opisane powyżej spostrzeżenia, zdecydowaliśmy się wprowadzić inną metodę (nazywaną dalej *metodą drugą*) oceny wyników parser płytkiego. W tej metodzie nie dokonywaliśmy prostego porównania fraz danych typów w obu strukturach składniowych, jak to miało miejsce w metodzie pierwszej. Zamiast tego, jako kryterium oceny strutktury generowanej przez parser przyjęliśmy to, czy frazy przez niego oznaczone pokrywają się z frazami wyodrębnionymi we wzorcowej strukturze, nawet jeżeli z powodu omówionych powyżej ograniczeń, wewnętrzna budowa struktur może się różnić. Wydaje nam się, że metoda druga w pełniejszym stopniu pozwala ocenić skuteczność działania parsera *puddle* z uwzględnieniem różnic między pełną a powierzchniową strukturą składniową zdania.

6.1.2. Przyjęte miary

Do oceny wyników w opisywanej tu procedurze zastosowaliśmy dwie standardowe miary ewaluacji zaproponowane w (van Rijsbergen, 1979, 114-115): precyzję oraz pokrycie.

Precyzję (ang. *precision*) rozumiemy w tej pracy jako stosunek liczby fraz danego typu rozpoznanych prawidłowo przez parser puddle do liczby wszystkich fraz oznaczonych przez parser jako frazy danego typu.

Przez **pokrycie** (ang. *recall*) rozumiemy stosunek liczby fraz danego typu prawidłowo rozpoznanych przez parser puddle do liczby fraz danego typu występujących w danych wzorcowych (w banku drzew).

Jeżeli przez A oznaczymy zbiór fraz danego typu występujących w danych wzorcowych,

zaś przez B zbiór fraz oznaczonych przez parser puddle jako frazy danego typu, to powyższe miary określone są przez poniższe równości:

Precyzja $= \frac{|A \cap B|}{|B|}$

Pokrycie $= \frac{|A \cap B|}{|A|}$

Obie miary są liczbami rzeczywistymi z przedziału $\langle 0, 1 \rangle$.

Innymi słowy, precyzję można rozumieć jako ocenę tego, z jaką skutecznością frazy danego typu zostały rozpoznane przez parser *puddle*. Natomiast pokrycie określa jaką część fraz danego typu udało się rozpoznać w wyniku działania parsera płytkiego.

6.2. Wyniki testowania

W tej sekcji przedstawiamy wartości miar umówionych w sekcji 6.1.2 dla dwóch metod oceny skuteczności płytkiego parsingu przedstawionych w sekcji 6.1.1.

W tabelach przedstawiających wyniki dla poszczególnych typów fraz przyjęliśmy następujące oznaczenia:

- AP – fraza przymiotnikowa,
- VP – fraza czasownikowa,
- NP – fraza rzeczownikowa,
- PP – fraza przyimkowa.

Wartości miar dla metody pierwszej przedstawia poniższa tabela:

	Precyzja	*Pokrycie*
AP	0,48	0,79
VP	0,48	0,33
NP	0,46	0,61
PP	0,58	0,60

Tabela 6.1.

Wartości miar dla metody drugiej zawarte są w tabeli poniżej:

	Precyzja	*Pokrycie*
AP	0,50	0,83
VP	0,49	0,39

	Precyzja	*Pokrycie*
NP	0,62	0,83
PP	0,70	0,72

Tabela 6.2.

6.3. Omówienie wyników i wnioski

Jak widać, spostrzeżenia i intuicja, które doprowadziły do wprowadzenia metody drugiej oceny skuteczności parsingu płytkiego, okazały się słuszne, gdyż miary obliczane dla wyników uzyskanych tą metodą są wyższe. Wzrost wartości miar nastąpił zwłaszcza dla ocen rozpoznawania fraz rzeczownikowych i przymiotnikowych, przy stosunkowo niewielkiej poprawie dla fraz przymiotnikowych i czasownikowych, co może świadczyć o słuszności tezy, że problem różnic między strukturami powstałymi w wyniku płytkiego i pełnego parsingu dotyczy przede wszystkim fraz rzeczownikowych i przymiotnikowych.

Ponadto zaobserwowaliśmy, że znaczną część błędnie rozpoznanych fraz rzeczownikowych (a także fraz przyimkowych zawierających takie frazy rzeczownikowe) stanowiły wyrażenia będące jednostkami nazwanymi (ang. *named entities*) lub je przypominające. W zbiorze reguł dla parsera *puddle* zawarliśmy m. in. reguły, które mają opisywać elementy, które, jak zauważyliśmy podczas opracowywania reguł i analizy dużych korpusów tekstów francuskojęzycznych, bardzo często okazywały się być nazwami przedsiębiorstw, instytucji, nazwiskami, itp. Elementy te często składają się z wyrazów, które nie są słowami francuskimi, obcych imion i nazwisk, a także zawierają (bądź nawet są nimi w całości) skróty. Prawidłowe rozpoznananie takich elementów jest zatem utrudnione. Jednak z punktu widzenia wykorzystania wyników płytkiego parsingu w tłumaczeniu automatycznym, wydaje się być korzystne, aby tego typu jednostki nazwane były włączone do struktury składniowej zdania generowanej przez parser.

Z tego względu zdecydowaliśmy się na stworzenie reguł dla parsera *puddle*, które służą rozpoznawaniu takich tworów. Przyjęliśmy stosunkowo proste kryteria: reguły opisują jako jednostkę nazwaną ciąg elementów, które nie zostały rozpoznane jako wyrazy języka francuskiego (nie posiadają oznaczenia części mowy, oznaczone są etykietą *ign*), które są pisane wielką literą (bądź w całości wielkimi literami). Elementy te mogą także zawierać charakterystyczne znaki jak np. &[1]. Mimo, że takie podejście może wydawać się stosunkowo naiwne, to uzyskane wyniki okazują się zadowalające. Uwzględnienie wspomnianych prostych jednostek nazwanych wydaje się mieć korzystny wpływ na przydatność struktur składniowych generowanych przez parser *puddle* dla celów tłumaczenia automatycznego.

Opisane powyżej jednostki nazwane traktowaliśmy jako frazy rzeczownikowe. We wzor-

[1] ang. *ampersand*, za (PN-I-06000) polska nazwa tego znaku brzmi *handlowe "i"*.

cowych strukturach składniowych nie występowały frazy, które parser *puddle* oznaczał jako frazy rzeczownikowe, bo odpowiadały opisanym wcześniej jednostkom nazwanym. Z tego powodu wartości miar uzyskanych dla fraz rzeczownikowych (a także w pewnym stopniu przyimkowych, gdyż w stosowanym przez nas podejściu, fraza przyimkowa składa się *de facto* z frazy rzeczownikowej oraz przyimka) są względnie niskie. Dotyczy to zwłaszcza precyzji, co sugeruje, że względnie spora liczba fraz rzeczownikowych, które odpowiadają jednostkom nazwanym, nie stanowi fraz rzeczownikowych we wzorcowych strukturach składniowych.

Zauważyliśmy, że niskie wartości miar dla rozpoznawania fraz czasownikowych wynikają w dużym stopniu z faktu, że w strukturach zawartych w banku drzew frazy zawierajace imiesłowy są zawsze oznaczone jako frazy czasownikowe. Z kolei w parserze *puddle* przyjęliśmy, że w konstrukcjach języka francuskiego, w których imiesłów jest określeniem rzeczownika, traktujemy imiesłów tak, jak przymiotnik – w naszym podejściu w takiej konstrukcji imiesłów tworzy frazę przymiotnikową. Takie podejście wynika także z faktu zastosowania płytkiego parsera *puddle* w procesie tłumaczenia automatycznego – warto, aby elementy określające rzeczownik były z nim ściślej związane w strukturze składniowej zdania, na podstawie której ma być generowane tłumaczenie z języka francuskiego. Ta różnica w traktowaniu imiesłowu spowodowała znaczące obniżenie oceny skuteczności rozpoznawania przez parser płytki fraz czasownikowych, a także w pewnym stopniu fraz przymiotnikowych.

Uzyskane przy użyciu opisanej w niniejszym rozdziale procedury wyniki dotyczące skuteczności rozpoznawania przez parser *puddle* podstawowych typów fraz mogą sugerować, że dość duża część fraz jest rozpoznawana błędnie (wartości miary precyzja) oraz wiele fraz nie zostaje rozpoznanych (wartości miary pokrycie). Przedstawiliśmy uzasadnienie, dlaczego, przynajmniej w części przypadków, wyniki parsera *puddle* różnią się od wyników wzorcowych. Wspomnieliśmy, że bank drzew przyjęty jako punkt odniesienia w ewaluacji został stworzony dla innych celów i przy użyciu innego podejścia do analizy składniowej, niż uczyniliśmy to tworząc płytki parser dla potrzeb tłumaczenia automatycznego. Uzyskiwane przez nas wyniki wydają się być zadowalające z punktu widzenia procesu tłumaczenia. Należy jednak pamiętać, że wszechostronna ocena wyników parsera *puddle* w kontekście tłumaczenia automatycznego możliwa będzie przez porównanie wyników tłumaczenia uzyskiwanego przez system tłumaczący z wykorzystaniem różnych systemów parsujących (np. parsera głębokiego). Jest to jednak kwestia wykraczająca poza zakres niniejszej pracy.

Podsumowanie

W niniejszej pracy przedstawiliśmy podejście do powierzchniowej analizy składniowej oparte na gramatykach regularnych. Przedstawiliśmy płytki parser *puddle*, który jest narzędziem niezależnym od języka tekstu, którego analizy składniowej dokonuje. Przedstawiliśmy formalizm reguł, które opisują struktury składniowe rozpoznawane przez płytki parser *puddle*. Pokazaliśmy, że podejście do gramatyki języka francuskiego przyjęte na potrzeby niniejszej pracy jest różne od podejścia typowo lingwistycznego. Przedstawiliśmy format danych wejściowych i wyjściowych używanych przez parser, a także rozwiązania stosowane w aplikacji. Dokonaliśmy ewaluacji zaimplementowanego parsera i interpretacji otrzymanych wyników. Zauważyliśmy szereg problemów wynikających z zastosowanego podejścia i zinterpretowaliśmy je w kontekście wykorzystania płytkiego parsera *puddle* na potrzeby tłumaczenia automatycznego.

Parser *puddle* wraz ze zbiorem reguł parsingu znajduje się na płycie CD-ROM dołączonej do niniejszej pracy. Dalszymi działaniami związanymi z parserem puddle będzie zintegrowanie z programem tłumaczącym *bonsai* oraz ocena skuteczności wykorzystania parsera *puddle* w procesie tłumaczenia automatycznego. Innym kierunkiem dalszych działań będzie wykorzystanie parsera *puddle* do analizy składniowej innych języków, np. do płytkiego parsingu języka polskiego i angielskiego.

DODATEK A

Zbiór reguł

Przedstawiamy poniżej zbiór reguł dla parsera *puddle* opracowany w ramach niniejszej pracy magisterskiej. Zbiór reguł znajduje się również wraz z programem na płycie CD-ROM dołączonej do pracy.

Reguły związane ze spójnikami wielowyrazowymi

```
Rule "compound conjunction 1"
Match: ([base~"afin|ainsi|alors|apres|attendu|aussitôt|autant|avant|bien|ce
         |cependant|considérant|considéré|depuis|des|encore|maintenant|malgré
         |non|outre|parce|pdt|pendant|plus|plutôt|pour|pourvu|quoi|sans|sauf
         |selon|sinon|sitôt|suivant|tandis|tant|tel|telle|telles|tels"]
        | (([base~"ce"] [base~"est-a-dire"]) | ([base~"de"] [base~"autant"]))
       ) [base~"que"];
Eval:  syntok("csub");

Rule "compound conjunction 2"
Match: (([base~"a"] [base~"condition|mesure|moins|présent|supposer"])
        | ([base~"alors"] [base~"meme"]) | ([base~"au"] [base~"point"])
        | ([base~"aussi"] [base~"bien"]) | ([base~"autant"] [base~"dire"])
        | ([base~"avant"] [base~"meme"]) | ([base~"chaque"] [base~"fois"])
        | ([base~"de"] (([base~"autant"] [base~"plus"])
        | [base~"ce|maniere|meme|parce|sorte"])) | ([base~"des"] [base~"fois"])
        | ([base~"des"] [base~"lors"]) | ([base~"du"] [base~"fait|moment"])
        | ([base~"en"] [base~"ce|sorte|tant"]) | ([base~"étant"] [base~"donné"])
        | ([base~"jusque"] [base~"a"] [base~"ce"]) | ([base~"le"]
        [base~"moins|temps"]) | ([base~"non"] [base~"plus"]) |
        ([base~"pour"] [base~"autant|peu"]) | ([base~"si"] [base~"bien|peu"])
        | ([base~"sous"] [base~"prétexte|réserve"])
        | ([base~"une"] [base~"fois"])
       ) [base~"que"];
Eval:  syntok("csub");
```

```
Rule "compound conjunction 3"
Match: (([base~"a"] (([base~"chaque"] [base~"fois"]) | ([base~"tel"] [base~"point"])
        | ([base~"telle"] [base~"enseigne"]))) | ([base~"au"] [base~"meme"]
         [base~"titre"]) | ([base~"ce"] [orth~"est"/i] [base~"a"] [base~"dire"])
        | ([base~"dans"] [base~"le"] [base~"but"]) | ([base~"de"] [base~"telle"]
         [base~"sorte"]) | ([base~"en"] (([base~"ce"] [base~"sens"])
         | ([base~"meme"] [base~"temps"]))) | ([base~"il"] [orth~"est"/i]
         [base~"vrai"]) | ([base~"par"] [base~"le"] [base~"fait"]) | ([base~"pour"]
        (([base~"cette"] [base~"raison"]) | ([base~"une"] [base~"fois"])))
       ) [base~"que"];
Eval:  syntok("csub");

Rule "compound conjunction 4"
Match: (([base~"de"] [base~"maniere"] [base~"a"] [base~"ce"])
        | ([base~"tant"] [base~"et"] [base~"si"] [base~"bien"])
       ) [base~"que"];
Eval:  syntok("csub");

Rule "compound conjunction 5"
Match: [base~"comme|meme"] [base~"si"];
Eval:  syntok("csub");

Rule "compound conjunction 6"
Match: [base~"en"] [base~"vue"] [base~"de"];
Eval:  syntok("csub");

Rule "compound conjunction 7"
Match: [base~"grand"] [base~"bien"] [base~"meme"];
Eval:  syntok("csub");

Rule "compound conjunction 8"
Match: [base~"ce"] [base~"est"] [base~"pourquoi"];
Eval:  delete(pos~"N", 3);
       syntok("csub");
```

```
Rule "compund conjunction 9"
Match: [base~"et"] [base~"alors|aussi|donc|encore|meme|puis"];
Eval:  syntok("conj");

Rule "compound conjunction 10"
Match: [base~"mais"] [base~"alors|aussi|encore"];
Eval:  syntok("conj");

Rule "compound conjunction 11"
Match: [base~"ou"] ([base~"alors|bien|encore"] | ([base~"bien"] [base~"encore"]));
Eval:  syntok("conj");

Rule "compound conjunction 12"
Match: [base~"tout"] [base~"comme"];
Eval:  syntok("conj");

Rule "compound conjunction 13"
Match: [base~"voire"] [base~"meme"];
Eval:  syntok("conj");

Rule "compound conjunction 14"
Match: [base~"de"] [base~"ou"];
Eval:  syntok("conj");
```

Reguły pomocnicze

```
Rule "cardinal numbers"
Match: [orth~"zéro|un|deux|trois|quatre|cinq|six|sept|huit|neuf|dix|onze
       |douze|treize|quatorze|quinze|seize|dix-sept|dix-huit|dix-neuf|vingt
       |trente|quarante|cinquante|soixante|soixante-dix|quatre-vingts
       |quatre-vingt dix|cent"/i];
Eval:  add("num", base, 1);
       delete(pos~"N", 1);

Rule "état membre"
Match: [base~"état"] [base~"membre"];
Eval:  syntok("N");
```

```
Rule "ainsi que"
Match: [orth~"ainsi qu(\’|e)"/i];
Eval:  add("conj", base, 1);
       add("prep", base, 1);

Rule "au meme titre que"
Match: [orth~"au meme titre qu(\’|e)"/i];
Eval:  add("conj", base, 1);

Rule "c’est pourquoi"
Match: [orth~"c’ est pourquoi"/i];
Eval:  add("conj", base, 1);

Rule "c’est a dire"
Match: [orth~"c’ est a dire"/i];
Eval:  add("conj", base, 1);

Rule "deux"
Match: [base~"deux"] [pos~"N"];
Eval:  delete(pos~"A", 1);

Rule "avoir"
Match: [orth~"a"/i];
Eval:  delete(pos~"N", 1);

Rule "plus"
Match: [base~"plus|moins"];
Right: [pos~"adv|A|part"];
Eval:  delete(pos~"N|V|part", 1);

Rule "etre"
Match: [base~"etre|avoir"];
Right: [pos~"part"];
Eval:  delete(pos!~"V|part", 1);
```

```
Rule "devoir + infinitive + participle"
Match: [base~"devoir"] [pos~"V" && mood~"N"] [pos~"part"];
Eval:  delete(pos!~"V", 1);
       delete(pos!~"V", 2);
       delete(mood!~"N", 2);
       delete(pos!~"part", 3);

Rule "devoir + infinitive"
Match: [base~"devoir"] [pos~"adv"]* [pos~"V" && mood~"N"];
Eval:  delete(pos!~"V|part", 1);
       delete(pos!~"adv", 2);
       delete(pos!~"V" && mood!~"N", 3);

Rule "du du"
Match: [base~"du"] [base~"du|de|des"];
Eval:  delete(pos!~"N", 1);

Rule "du"
Left:  [pos~"V" && base~"avoir|etre"];
Match: [orth~"du|due|dus"/i];
Eval:  delete(pos!~"part", 2);

Rule "celle"
Match: [base~"celle"];
Right: [base~"que|qui"];
Eval:  delete(pos!~"pron", 1);

Rule "a bien"
Left:  [base~"a"];
Match: [base~"bien"];
Right: [pos~"V"];
Eval:  delete(pos~"A", 2);

Rule "une"
Match: [base~"une?"];
Right: [pos~"A|N"];
Eval:  delete(pos~"N", 1);
```

```
Rule "haut risque"
Match: [base~"haut"] [base~"risque"];
Eval:  unify(number gender, 1, 2);
       delete(pos!~"A", 1);
       delete(pos!~"N", 2);

ule "pas"
Left:  [base~"ne"] [pos~"art" && base~"le"]? [pos~"V"];
Match: [base~"pas"];
Eval:  delete(pos!~"adv", 4);

Rule "pas 2"
Left:  [base~"ne"] [pos~"part"];
Match: [base~"pas"];
Eval:  delete(pos!~"adv", 3);

Rule "pas 3"
Match: [base~"pas"];
Right: [base~"que|qui"];
Eval:  delete(pos!~"adv", 1);

Rule "pas 4"
Left:  [pos~"V"];
Match: ([orth~","]? [pos~"adv"] [orth~","]?)? [base~"pas"];
Eval:  delete(pos!~"adv|interp", 2);
       delete(pos!~"adv", 3);

Rule "ce qui/que/dont"
Match: [base~"ce"] [base~"que|qui|dont"];
Eval:  delete(pos!~"pron", 1);
       syntok("relpron");

Rule "est"
Left:  [base~"ne"] [pos~"art" && base~"le"]?;
Match: [orth~"est"/i];
Right: [base~"pas"];
```

```
Eval:  delete(pos!~"V", 3);

Rule "cela étant"
Match: [base~"cela"] [pos~"part"];
Eval:  delete(pos~"N", 2);

Rule "en tant que"
Match: [base~"en"] [base~"tant"] [base~"que"];
Eval:  syntok("prep");

Rule "le lui"
Match: [pos~"art"]? [base~"lui"];
Right: [pos~"V"];
Eval:  delete(pos~"part", 2);

Rule "c' est"
Left:  [base~"ce|il|elle|on"] [base~"ne"]?;
Match: [orth~"est"/i];
Eval:  delete(pos!~"V", 3);
       delete(pos!~"pron", 1);

Rule "est + article"
Match: [orth~"est"/i];
Right: [pos~"art"];
Eval:  delete(pos!~"V", 1);

Rule "est + adverb + participle"
Match: [orth~"est"/i] [pos~"adv"]? [pos~"part"];
Eval:  delete(pos!~"V", 1);

Rule "art + pret"
Left:  [pos~"art"];
Match: [base~"pret"];
Eval:  delete(pos!~"N", 2);

Rule "tel"
Match: [base~"un"] [base~"tel"];
```

```
Right: [pos~"N"]?;
Eval:  unify(number gender, 1, 2);
       add("A:*:*",  base, 2);
       unify(number gender, 1, 2);
       delete(pos~"N", 1);
       delete(pos~"art", 2);
       delete(pos~"A", 3);

Rule "le/la"
Match: [base~"le"];
Eval:  delete(pos!~"art|pron", 1);

Rule "sous"
Match: [base~"sous"];
Right: [pos~"N"];
Eval:  delete(pos!~"prep", 1);

Rule "autre"
Match: [base~"autre"] [pos~"N"];
Eval:  unify(number gender, 1, 2);
       delete(pos!~"A", 1);
       delete(pos!~"N", 2);

Rule "droit"
Match: ([pos~"art|prep"] | sb)? [base~"droit"] [pos~"A"];
Eval:  delete(pos!~"N", 2);

Rule "pour part"
Match: [base~"pour"] [base~"sa"]? [base~"part"];
Eval:  delete(pos!~"N", 3);

Rule "il"
Match: [orth~"il|elle"/i];
Eval:  delete(pos!~"pron", 1);

Rule "je"
Match: [base~"je"];
```

```
Right: [pos~"pron"]? [pos~"V"];
Eval:  delete(pos!~"pron", 1);

Rule "que"
Match: [base~"que"];
Eval:  delete(pos~"adv", 1);

Rule "pour"
Match: [base~"pour"];
Right: [pos~"art|N|V|adv|pron"];
Eval:  delete(pos!~"prep", 1);

Rule "pour rien"
Left:  [base~"pour"];
Match: [base~"rien"];
Eval:  delete(pos~"N", 2);

Rule "sois"
Left:  [pos~"N|pron"];
Match: [orth~"sois"/i];
Right: [pos~"part" && part-tense~"past"];
Eval:  delete(pos!~"V", 2);

Rule "trop"
Match: [base~"trop"];
Right: [pos~"adv|A"];
Eval:  delete(pos!~"adv", 1);

Rule "de"
Match: [base~"de"];
Eval:  delete(pos!~"prep", 1);

Rule "premier"
Match: [base~"premier"];
Right: [pos~"N"] [pos~"A" && base!~"sur"]?;
Eval:  delete(pos~"N", 1);
       delete(pos!~"A", 3);
```

```
Rule "point"
Left:  [pos~"A"];
Match: [base~"point"];
Eval:  unify(number gender, 1, 2);
       delete(pos!~"N", 2);

Rule "adjective + noun"
Match: [pos~"A"] [pos~~"N"];
Eval:  unify(number gender, 1, 2);
       delete(pos~"N", 1);

Rule "demain"
Match: [base~"demain"];
Eval:  delete(pos~"A", 1);

Rule "du des"
Match: [base~"du|des"];
Eval:  add("prep", base, 1);

Rule "par sur"
Match: [base~"par|sur"];
Eval:  delete(pos!~"prep", 1);

Rule "aujourd' hui"
Match: [base~"aujourd'"] [base~"hui"];
Eval:  syntok("adv");

Rule "a ce que"
Match: [base~"a"] [base~"ce"] [base~"que"];
Eval:  syntok("csub");

Rule "est celle"
Left:  [orth~"est"/i];
Match: [base~"celle"];
Eval:  delete(pos~"N", 2);
```

```
Rule "adjective + regle"
Left:  [pos~"art"]? [pos~"A"];
Match: [pos~"N" && base~"regle"];
Eval:  unify(gender number, 1, 2, 3);
       delete(pos!~"A", 2);
       delete(pos~"V", 3);

Rule "regle + adjective"
Left:  [pos~"art"]?;
Match: [pos~"N" && base~"regle"] [pos~"A"];
Eval:  unify(gender number, 1, 2, 3);
       delete(pos~"V", 2);
       delete(pos!~"A", 3);

Rule "ce cas-la"
Match: [base~"en"]? [base~"ce"] [base~"cas-la"];
Eval:  syntok("prep");

Rule "tout a fait"
Match: [base~"tout"] [base~"a"] [base~"fait"];
Eval:  syntok("adv");

Rule "tout ceci"
Match: [base~"tout"];
Right: [base~"ceci"];
Eval:  delete(pos~"A", 1);

Rule "en"
Match: [base~"en"];
Eval:  delete(pos~"adv", 1);

Rule "entre as verb"
Match: [orth~"entre"];
Right: [pos~"prep"];
Eval:  delete(pos~"prep", 1);
```

```
Rule "rendu public"
Match: [base~"rendu"] [base~"public"];
Eval:  delete(pos!~"N", 1);

Rule "en + noun"
Match: [base~"en"] [pos~"N"];
Eval:  delete(pos~"A", 2);

Rule "gerund"
Match: [base~~"en"] [base~"ne"]? [base~"leur"]?
       [pos~"part" && part-tense~"present"] [base~"pas"]?;
Eval:  add("grn:m.f:sg.pl",  base, 4);

Rule "infinitive"
Left:  [base~"venir"] [base~"de"];
Match: [pos~"V" && mood~"N"];
Eval:  delete(pos!~"V", 3);

Rule "single letter"
Match: [orth~"[[:L*:]]" && base~"c|l|j|s|i|t"];
Eval:  add("ign",  base, 1);
       delete(pos!~"ign", 1);

Rule "a)"
Left:  (sb | [orth!~"\("]);
Match: [orth~"a"/i] [orth~"\)"];
Eval:  add("ign",  base , 2);
       delete(pos!~"ign", 2);

Rule "part present"
Match: [pos~"part" && part-tense~"present" && base!~"importer"];
Eval:  delete(pos~"A", 1);

Rule "past participle"
Match: [base~"avoir"] [pos~"adv"]? [pos~"part" && part-tense~"past"];
Eval:  delete(pos!~"part", 3);
       delete(part-tense!~"past", 3);
```

```
Rule "etre + A/part"
Match: [pos~"V" && base~"etre"] [pos~"adv"]? [pos~"A"];
Eval:  delete(pos!~"V", 1);
       delete(pos!~"A", 3);

Rule "etre + A/part 2"
Match: [pos~"V" && base~"etre"] [pos~"part"];
Eval:  delete(pos!~"V", 1);
       delete(pos!~"part", 2);

Rule "etre"
Left:  [pos~"V"];
Match: [orth~"etre"/i];
Eval:  delete(pos~"N", 2);

Rule "fixe"
Left:  [pos~"N"];
Match: [base~"fixer"];
Eval:  delete(pos~"A", 2);

Rule "nombre"
Match: [pos~"A"] [base~"nombre"];
Eval:  delete(pos~"V", 2);

Rule "chaque"
Match: [base~"chaque"] [pos~"N"];
Eval:  delete(pos~"A", 2);

Rule "neutre"
Left:  [pos~"V"];
Match: [base~"neutre"];
Eval:  delete(pos~"N", 2);

Rule "deuxieme"
Left:  [pos~"art"]?;
Match: [base~"deuxieme"] [pos~"N"];
```

```
Eval:  delete(pos!~"A", 2);
       unify(gender number, 1, 2, 3);

Rule "important"
Left:  [pos~"N"];
Match: [orth~"important"/i];
Eval:  delete(base~"importer", 2);

Rule "nouveau + noun"
Left:  ([pos~"art"] | [base~"des|de|du|au|aux"])? [base~"nouveau" && pos~"A"];
Match: [pos~"N"];
Eval:  unify(number gender, 2, 3);
       delete(pos!~"A", 2);
       delete(pos!~"N", 3);

Rule "article + single noun"
Left:  [pos~"art"] [pos~"num"]?;
Match: [pos~"N" && base!~"plus|moins"];
Right: [pos~~"A|part"] ([pos~"interp|adv|prep|conj"] | se);
Eval:  delete(pos!~"N", 3);

Rule "adj + noun"
Left:  [base~"de|d'|des|au|aux"] [pos~~"A|part"];
Match: [pos~"N"];
Right: ([pos~"interp|adv|prep|conj"] | se);
Eval:  delete(pos~"A", 3);

Rule "de + single noun"
Left:  [base~"d'"];
Match: [pos~"N"];
Right: ([pos!~"N"] | se);
Eval:  delete(pos!~"N", 2);

Rule "single noun + de"
Left: (sb | [pos~"interp"]);
Match: [pos~"N"];
Right: [base~"de"];
```

```
Eval:  delete(pos!~"N", 2);

Rule "prep + single noun"
Left:  [pos~"prep"] [pos~"art"]?;
Match: [pos~"N" && base!~"certain|plus"];
Right: (se | [pos~"interp"] | [base~"i|I"] | [pos~"V|prep"]);
Eval:  delete(pos!~"N", 3);

Rule "art + noun + adjective"
Left:  ([pos~"art"] | [base~"de|du|d'|des|au|aux"]) [pos~"num"]?;
Match: [pos~"N" && base!~"plus|moins"] [pos~"adv"]?;
Right: [pos~"A|part"] ([pos~"interp|adv|prep|conj"] | se);
Eval:  unify(gender number, 3, 5);
       delete(pos~"A", 3);
       delete(pos~"N", 5);

Rule "adverb + noun"
Left:  ([pos~"art"] | [base~"de|du|d'|des|au|aux"])? [pos~"adv" && base!~"moins|plus"];
Match: [pos~"N"];
Right: ([pos~"interp|adv|prep|conj"] | se);
Eval:  delete(pos~"A", 3);

Rule "garantie + de"
Match: [pos~"N" && base~"garantie"] [base~"de|du|des"];
Eval:  delete(pos~"A", 1);

Rule "garantie + adjective"
Match: [pos~"N" && base~"garantie"] [pos~"A"];
Eval:  delete(pos~"A", 1);

Rule "adjective + garantie"
Left:  [pos~"A"];
Match: [pos~"N" && base~"garantie"];
Right: (se | [pos!~"N"]);
Eval:  delete(pos~"A", 2);
```

```
Rule "ce + noun"
Match: [base~"ce"] [pos~"N"];
Right: (se | [pos~"prep|V|art|conj"]);
Eval:  delete(pos~"A", 2);

Rule "non"
Match: [base~"non"];
Right: [pos~"N|adv"];
Eval:  delete(pos!~"adv", 1);

Rule "restant"
Left:  [pos~"N"];
Match: [base~"restant"];
Eval:  delete(pos~"N", 2);

Rule "un/une"
Match: [base~"un"];
Right: [pos~"N"];
Eval:  delete(pos!~"art", 1);

Rule "pronouns"
Match: [base~"tout|tous|toutes"];
Eval:  delete(pos~"N", 1);

Rule "troisieme"
Match: [base~"troisieme"];
Eval:  delete(pos~"N", 1);

Rule "sauf si"
Match: [base~"sauf"] [base~"si"];
Eval:  syntok("csub");

Rule "été"
Match: [orth~"été"/i];
Right: [pos~"part"];
Eval:  delete(pos!~"part", 1);
```

```
Rule "été 2"
Match: [base~"non"] [orth~"été"/i];
Eval:  delete(pos~"N", 2);

Rule "été 3"
Left:  [pos~"V" && base~"etre|avoir"] [pos~"adv"]*;
Match: [orth~"été"/i];
Eval:  delete(pos~"N", 3);

Rule "au + noun"
Match: [base~"au"] [pos~"N"];
Eval:  delete(pos~"V", 2);

Rule ", noun ,"
Match: [pos~"interp"] [pos~"N"] [pos~"interp"];
Eval:  delete(pos~"A", 2);

Rule "(CE)"
Left:  [orth~"\("];
Match: [orth~"CE"];
Right: [orth~"\)"];
Eval:  add("ign",  base, 2);
       delete(pos!~"ign", 2);
       group(NE, 2);

Rule "CE"
Left:  ([orth~"\/"] | [orth~"\d+"]);
Match: [orth~"CE"];
Eval:  add("ign",  base, 2);
       delete(pos!~"ign", 2);
       group(NE, 2);

Rule "SE"
Left:  [pos~"ign" && base~"[[:Lu:]].+"];
Match: [orth~"SE"];
Eval:  add("ign",  base, 2);
       delete(pos!~"ign", 2);
```

```
Rule "en + noun"
Match: [base~"en"] [pos~"N"];
Eval:  delete(pos~"pron", 1);

Rule "commun"
Match: [pos~"N"] [base~"commun"];
Eval:  unify(number gender, 1, 2);
       delete(pos!~"A", 2);

Rule "verb"
Match: [pos~"V"];
Right: [base~"que"];
Eval:  delete(pos~"A", 1);

Rule "participle + a"
Match: [pos~"part" && part-tense~"past"];
Right: [base~"a"];
Eval:  delete(pos~"A", 1);

Rule "avec"
Match: [base~"avec"];
Right: [pos~"art"];
Eval:  delete(pos!~"prep", 1);

Rule "certain + noun"
Match: [base~"certain"];
Right: [pos~"N"];
Eval:  unify(gender number, 1, 2);
       delete(pos~"N", 1);

Rule "the same form of verb and noun"
Left:  [pos~"A"];
Match: [pos~"V" && mood~"I|S|D" && base!~"plus"];
Eval:  unify(number gender, 1, 2);
       delete(pos~"N", 1);
       delete(pos~"V", 2);
```

```
Rule "the same form of verb and noun 2"
Left:  [pos~"N"];
Match: [pos~"V" && mood~"I|S|D" && base!~"plus"];
Eval:  delete(pos~"N", 2);

Rule "plus + adverb"
Match: [base~"le"]? [base~"plus|moins"];
Right: [pos~"adv|A|part"];
Eval:  delete(pos!~"adv", 2);
       delete(pos!~"art", 1);

ule "le plus + adverb"
Match: [base~"le"] [base~"plus|moins"];
Right: [pos~"adv|A|part"];
Eval:  unify(number gender, 1, 3);
       delete(pos!~"adv", 2);
       delete(pos!~"art", 1);
       syntok("adv");

Rule "plus/moins + adj"
Left:  [base~"plus|moins"]; # [pos~"adv"]?;
Match: [pos~"A"];
Right: (se | [pos!~"N"]);
Eval:  delete(pos~"N", 2);

Rule "au plus"
Match: [base~"au"] [base~"plus|moins"];
Eval:  syntok("adv");

Rule "plus de"
Match: [base~"plus|moins"];
Right: [base~"de"];
Eval:  delete(pos!~"adv", 1);

Rule "verb + si"
Left:  [pos~"V"];
```

```
Match: [base~"si"];
Right: [pos!~"V"];
Eval:  delete(pos!~"csub", 2);

Rule "si + article"
Match: [base~"si"];
Right: [pos~"art"]? [pos~"N"];
Eval:  delete(pos~"N", 1);

Rule "meme + noun"
Match: [pos~"art"] [base~"meme"] [pos~"N"];
Eval:  delete(pos~"N", 2);

Rule "si + pronoun"
Match: [base~"si"] [pos~"pron"];
Eval:  delete(pos~"N", 1);

Rule "par conséquent"
Match: [base~"par"] [base~"conséquent"];
Eval:  syntok("adv");

Rule "par + noun"
Match: [base~"par"] [base~"le"]? [pos~"N"];
Eval:  delete(pos~"N", 1);

Rule "en dernier ressort"
Left:  [base~"en"] [base~"dernier"];
Match: [base~"ressort"];
Eval:  delete(pos~"V", 3);

Rule "msp"
Match: [pos~"art"]? [pos~"A"] [pos~"conj"] [pos~"A"] [pos~"N"];
Eval:  delete(pos!~"A", 2);
       delete(pos!~"A", 4);
       delete(pos!~"N", 5);
```

```
Rule "noun + conjuncted adjectives"
Match: [pos~"art"]? [pos~"N"] [pos~"A"] [pos~"conj"] [pos~"A"];
Right: (se | [pos!~"N"]);
Eval:  unify(number gender, 1, 2, 3, 5);
       delete(pos!~"N", 2);
       delete(pos!~"A", 3);
       delete(pos!~"A", 5);

Rule "prep + noun"
Match: [pos~"prep"] [pos~"N"];
Eval:  delete(pos~"V", 2);

Rule "prep + noun + prep"
Left:  [pos~"prep"];
Match: [pos~"N"];
Right: [pos~"prep"];
Eval:  delete(pos~"A", 2);

Rule "past participle + adjective"
Match: [pos~"part" && part-tense~"past"] [pos~~"adv"]? [pos~"A"]+;
Eval:  unify(number gender, 1, 3);

Rule "ces memes"
Match: [base~"ce"] [base~"memes?"];
Eval:  delete(pos!~"art", 1);

Rule "est"
Match: [orth~"est"/i];
Right: [pos~"adv"];
Eval:  delete(pos~"N", 1);

Rule "non + adverb"
Match: [base~"non"] [pos~"adv"];
Eval:  syntok("adv");

Rule "H1.1: help"
Match: ([pos~"art"] | [base~~"de|du|des|au|aux"]) [pos~"A"] [pos~~"N"];
```

```
Eval:  unify(number gender, 2, 3);
       delete(pos!~"A", 2);

Rule "H1.2: help"
Left:  ([pos~"art"] | [pos~"prep"] | [pos~"conj"] | sb);
Match: [pos~"A"] [pos~~"N"];
Right: se;
Eval:  unify(number gender, 2, 3);
       delete(pos!~"A", 2);

Rule "H1: help"
Match: [pos~"A"] [pos~"N"];
Eval:  unify(number gender, 1, 2);

Rule "H2: art + n"
Match: [pos~"art" && base~"le"] [pos~"N"];
Eval:  unify(number gender, 1, 2);
```

Reguły związane z jednostkami nazwanymi

```
Rule "NE1.1: Naive named entity - starts with a capital letter and consists
      of several unknown items"
Match: [base~"le"]? [pos~"ign" && base~"[[:Lu:]][[:L*:]\'\-\/\.\d]+"] [pos~"ign"]*;
Eval:  delete(pos!~"art", 1);
       group(NE, 2);

Rule "NE1.2: naive named entity"
Left:  [orth~"\""];
Match: [pos~"ign"] [pos~"ign"]*;
Right: [orth~"\""];
Eval:  group(NE, 2);

Rule "DATE1: Date with month name"
Match: [base~"\d\d?"] [orth~"er"/i]? [base~"janvier|février|mars|avril|mai|juin|
        juillet|aout|septembre|octobre|novembre|décembre"] [base~"\d\d\d\d"]?;
Eval:  group(DATE, 3);
```

```
Rule "DATE2: Numerical date"
Match: [base~"\d?\d\.\d?\d\.\d\d\d\d"];
Eval:  group(DATE, 1);

Rule "NUM1.1: no followed by NUM1"
Match: [orth~"no"]? [orth~"\d.*"] [orth~"er"/i]? [orth~"\%"]?;
Right: ([base!~"interp"] | [head=[]] | se);
Eval:  group(NUM, 2);

Rule "NUM1: All tokens that begin with a number, optional ) and % (eg. 34 %)"
Left:  ([orth!~"\("] | sb);
Match: [orth~"\d.*"] [orth~"[\)\%]"]?;
Eval:  group(NUM, 2);

Rule "NUM2: Roman numbers from I to X with captial letters"
Match: [orth~"(I|II|III|IV|V|VI|VII|VIII|IX|X)"];
Eval:  group(NUM, 1);

Rule "NUM3: Single letters followed by ')'"
Left:  ([orth!~"\("] | [head=[]] | sb);
Match: [orth~"[[:L*:]]"] [orth~"\)"];
Eval:  group(NUM, 2);

Rule "NUM4.1: Numerical tokens surrounded by rounded parentheses"
Match: [orth~"\("] [type=NUM] [orth~"\)"];
Eval:  group(NUM, 2);

Rule "NUM5: Conjunction of numerical phrases"
Match: [type=NUM] (([pos~"conj"] | [orth~","]) [type=NUM])+ ;
Eval:  group(NUM, 1);

Rule "NE2: named entity surrounded by quotes"
Match: [orth~"\""] [type=NE] [orth~"\""];
Eval:  group(NE, 2);
```

```
Rule "NE3: named entity in parenthesis"
Match: [orth~"[\[\(]"] ([type=NE] | [pos~"ign" && orth~"\d.*"]) [orth~"[\]\)]"];
Eval:  group(NE, 2);

Rule "NE4: conjunction of named entities"
Match: [type=NE] ([pos~"conj"]? ([type=NE] | [type=NUM]));
Eval:  group(NE, 1);
```

Reguły związane z frazami rzeczownikowymi i przymiotnikowymi

```
Rule "AP0: adverb + que + adjective"  #autant que possible
Match: [pos~"adv"] [base~"que"] [pos~~"A"];
Eval:  delete(pos!~"adv", 1);
       delete(pos!~"conj", 2);
       delete(pos!~"A", 3);
       group(AP, 3);

Rule "AP1.1: adverb + at least 1 adjective"
Match: [pos~"adv"]+ [pos~"A"] [pos~"A"]+;
Eval:  unify(number gender, 2, 3);
       delete(pos!~"adv", 1);
       delete(pos!~"A", 2);
       delete(pos!~"A", 3);
       group(AP, 2);

Rule "AP1.2: at least 1 adjective"
Left:  ([base!~"de|du|des|le" && pos!~"conj"] | sb | [head=[]]);
Match: [pos~"A" && base!~"etre"] [pos~"A"]+ [pos~"adv"]?;
Eval:  unify(number gender, 2, 3);
       delete(pos!~"A", 2);
       delete(pos!~"A", 3);
       delete(pos!~"adv", 4);
       group(AP, 2);

Rule "AP1.3: adverb + 1 adjective"
Match: [pos~"adv" && base!~"pas|tout"]+ [pos~"A"];
Eval:  delete(pos!~"adv", 1);
```

```
       delete(pos!~"A", 2);
       group(AP, 2);

Rule "AP1.4: 1 adjective"
Left:  ([base!~"de|du|des|au|aux|avoir|etre" && pos!~"conj|art|pron"] | sb
        | [head=[pos!~"pron"]] | ([pos~"interp"] [pos~"conj"]));
Match: [pos~"A" && base!~"etre"] [pos~"adv"]?;
Eval:  delete(pos!~"A", 2);
       delete(pos!~"adv", 3);
       group(AP, 2);

Rule "AP1.4.1: 1 adjective"
Left:  ([pos~"conj|prep"] | [pos~"art" && base~"le|un|sa"]);
Match: [pos~~"A"];
Right: [pos~"N"];
Eval:  group(AP, 2);

ule "AP1.4.2: 1 adjective"
Left:  [pos~"art"] [pos~"N"];
Match: [pos~"A" && base!~"etre"];
Eval:  delete(pos!~"A", 3);
       group(AP, 3);

Rule "AP1.5: adverb + 2 adjectives"
Match: [pos~"adv"]+ [pos~"A"] [pos~"A"];
Eval:  unify(number gender, 2, 3);
       delete(pos!~"adv", 1);
       delete(pos!~"A", 2);
       delete(pos!~"A", 3);
       group(AP, 2);

Rule "AP1.6: 2 adjectives"
Left:  ([base!~"de|du|des" && pos!~"conj"] | sb | [head=[]]);
Match: [pos~"A" && base!~"etre"] [pos~"A"];
Eval:  unify(number gender, 2, 3);
       delete(pos!~"A", 2);
       delete(pos!~"A", 3);
```

```
       group(AP, 2);

Rule "AP1.4.3: 1 adjective"
Match: [pos~~"A"];
Eval:  group(AP, 1);

Rule "AP2.1: present participle as adjective (after noun)"
Left:  (([pos~"N"] [pos~"A"]*) | [type=NP]);
Match: [pos~"adv"]* [pos~"pron" && base~"me|te|se|nous|vous"]?
       [pos~"part" && part-tense~"present"] [pos~"adv"];
Right: ([pos!~"A|N|art"] | se);
Eval:  unify(number gender, 1, 4);
       delete(pos!~"adv", 2);
       delete(pos!~"part" && part-tense!~"present", 4);
       delete(pos!~"adv", 5);
       group(AP, 4);

Rule "AP2.2.1: past participle as adjective (after noun)"
Left:  ([pos~"N"] | [type=NP]);
Match: [pos~"adv"]* [pos~"part" && part-tense~"past"]
       ([pos~"A"] | [type=AP && head=[pos~"A"]]) [pos~"adv" && base!~"pas"]*;
Eval:  unify(number gender, 1, 3, 4);
       delete(pos!~"part" && part-tense!~"past", 3);
       delete(pos!~"A", 4);
       delete(pos!~"adv", 5);
       group(AP, 3);

Rule "AP2.2: past participle as adjective (after noun)"
Left:  (([pos~"N"] [pos~"A"]*) | [type=NP]);
Match: [pos~"adv"]* [pos~"part" && part-tense~"past"] [pos~"adv" && base!~"pas"]*;
Eval:  unify(number gender, 1, 3);
       delete(pos!~"part" && part-tense!~"past", 3);
       delete(pos!~"adv", 4);
       group(AP, 3);

Rule "AP3.1: conjunction + adjective phrase"
Left:  [type=AP];
```

```
Match: [pos~"conj"] [type=AP];
Eval:  unify(number gender, 1, 3);
       delete(pos!~"conj", 2);
       group(CAP, 3);

Rule "AP4: adjective phrase + conjunctional adjective phrase"
Match: [type=AP] [type=CAP]+;
Eval:  unify(number gender, 1, 2);
       group(AP, 1);

Rule "AP5: adverb in comperior or superior degree"
Match: [base~"plus|le plus|la plus"] [pos~"adv"];
Eval:  delete(pos!~"adv", 2);
       group(AP, 2);

Rule "NP1.1: adjective phrase + noun + adjective phrase"
Match: [pos~"pron" && base~"tout|tous|toutes"]? [pos~"art"]? [base~"memes?"]?
       [type=AP] [pos~"N"] [type=AP];
Eval:  unify(number gender, 2, 4, 5, 6);
       delete(pos!~"pron", 1);
       delete(pos!~"art", 2);
       delete(pos!~"N", 5);
       group(NP, 5);

Rule "NP1.2: noun + adjective phrase"
Match: [pos~"pron" && base~"tout|tous|toutes"]? [pos~"art"]? [base~"memes?"]?
       [pos~"N" && base!~"etre"] [type=AP];
Eval:  unify(number gender, 2, 4, 5);
       delete(pos!~"pron", 1);
       delete(pos!~"art", 2);
       delete(pos!~"N", 4);
       group(NP, 4);

Rule "NP1.3: adjective phrase + noun"
Match: [pos~"pron" && base~"tout|tous|toutes"]? [pos~"art"]? [base~"memes?"]?
       [type=AP] [pos~"N"] [pos~"adv"]?;
Eval:  unify(number gender, 2, 4, 5);
```

```
       delete(pos!~"pron", 1);
       delete(pos!~"art", 2);
       delete(pos!~"N", 5);
       delete(pos!~"adv", 6);
       group(NP, 5);

Rule "NP2.2: gerund + adjective phrase"
Match: [pos~"pron" && base~"tout|tous|toutes"]? [pos~"art"]? [base~"memes?"]?
       [base~~"en"] [pos~"grn"] [type=AP];
Eval:  unify(number gender, 2, 5, 6);
       delete(pos!~"pron", 1);
       delete(pos!~"art", 2);
       delete(pos!~"prep", 4);
       delete(pos!~"grn", 5);
       group(NP, 5);

Rule "NP3.1.1: article + noun"
Left:  (sb | [pos!~"pron"]); # | [pos~"pron" && base!~"tout|tous|toutes"]);
Match: [pos~"art"] [base~"memes?"]? [pos~"N"];
Eval:  unify(number gender, 2, 4);
       delete(pos!~"art", 2);
       delete(pos!~"pron", 3);
       delete(pos!~"N", 4);
       group(NP, 4);

Rule "NP3.1: article + noun"
Match: [pos~"pron" && base~"tou(t|s|es)"]? [pos~"art"] [base~"memes?"]? [pos~"N"];
Eval:  unify(number gender, 2, 4);
       delete(pos!~"pron", 1);
       delete(pos!~"art", 2);
       delete(pos!~"pron", 3);
       delete(pos!~"N", 4);
       group(NP, 4);

Rule "NP3.2: single noun"
Left:  ([pos!~"art|pron"] | sb | [head=[]]);
Match: [base~"non"]? [pos~"N" && base!~"est"];
```

```
Eval:  delete(pos!~"N", 3);
       delete(pos!~"adv", 2);
       group(NP, 3);

Rule "NP3.3: single gerund"
Left:  [base~~"en"];
Match: [base~"ne"]? [base~"leur"]? [pos~"grn"] [base~"pas"]?;
Eval:  unify(gender number, 2, 3);
       delete(pos!~"prep", 1);
       delete(pos!~"grn", 4);
       delete(pos!~"adv", 5);
       group(NP, 4);

Rule "NP3.4: just noun"
Match: [pos~~"N"];
Eval:  group(NP, 1);

Rule "NP5.1: cardinal number + noun phrase"
Match: [pos~"num"]+ [type=NP];
Eval:  delete(pos!~"num", 1);
       group(NP, 2);

Rule "NP6.1: pronoun as noun phrase"
Match: [pos~"pron" && base~"je|tu|il|elle|on|nous|vous|ils|elles|y"];
Eval:  delete(pos!~"pron", 1);
       group(NP, 1);

Rule "NP6.1.1: pronoun as noun phrase"
Match: [pos~"pron" && base~"ce"];
Right: [base!~"que|qui|dont"];
Eval:  delete(pos!~"pron", 1);
       group(NP, 1);

Rule "NP6.2: pronoun as noun phrase"
Left:  [pos~"prep"];
Match: [pos~"pron" && base~"tous|tout|toutes|celui|ceux|celles?"];
Eval:  delete(pos!~"pron", 2);
```

```
       group(NP, 2);

Rule "NP6.3: pronoun as noun phrase"
Match: [pos~"art" && base~"le"] [pos~"pron" && base~"meme|lui"];
Eval:  delete(pos!~"art", 1);
       delete(pos!~"pron", 2);
       group(NP, 2);

Rule "NP6.4: ou + les"
Left:  [base~"ou"];
Match: [base~"les"]; # la, le?
Eval:  delete(pos!~"pron", 2);
       group(NP, 2);

Rule "NP+NUM1: numerical phrases as apposition"
Match: [type=NP] [type=NUM]+;
Eval:  group(NP, 1);

Rule "NP+NE1: named entites as apposition"
Match: [type=NP] [type=NE]+;
Eval:  group(NP, 1);

Rule "NP+DATE1: le + date"
Match: [pos~"art"] [type=DATE];
Eval:  delete(pos!~"art", 1);
       group(NP, 2);

Rule "NP+DATE2: Date as apposition"
Match: [type=NP] [type=DATE];
Eval:  group(NP, 1);
```

Reguły związane z frazami dopełniaczowymi i frazami przymiotnikowymi

```
Rule "prep+NUM1: numerical phrase as prepositional phrase"
Match: [pos~"prep"]+ [type=NUM];
Eval:  group(PP, 2);
```

```
Rule "prep+DATE1: Date as prepositional phrase"
Match: [pos~"prep"]+ [type=DATE];
Eval:  group(PP, 2);

Rule "PP4: de + noun phrase at the beginning of sentence"
Left:  sb;
Match: [pos~"prep" && base~"du|de|des|d'"] [type=NP];
Eval:  delete(pos!~"prep", 2);
       group(PP, 3);

Rule "GP1: genitive phrase"
Match: [pos~"prep" && base~"du|de|des|d'"] [type=NP];
Eval:  delete(pos!~"prep", 1);
       group(GP, 2);

Rule "GP1.2: named entity genitive phrase"
Match: [pos~"prep" && base~"du|de|des|d'"] [type=NE];
Eval:  delete(pos!~"prep", 1);
       group(GP, 2);

Rule "GP2: adjective phrase + genitive phrase"
Match: [type=AP] [type=GP];
Eval:  group(AP, 1);

Rule "NP12.2: genitive phrase + genitive phrases + adjective phrase"
Match: [type=GP] [type=GP] [type=AP];
Eval:  unify(number gender, 1, 3);
       group(GP, 1);

Rule "GP4.1: conjunction + genitive phrase"
Match: [pos~"conj"] [type=GP];
Eval:  delete(pos!~"conj", 1);
       group(CGP, 2);

Rule "GP5: genitive phrase + conjunctional genitive phrase"
Match: [type=GP] [type=CGP]+;
Eval:  group(GP, 1);
```

```
Rule "NP12.1: noun phrase + genitive phrases + adjective phrase"
Match: [type=NP] [type=GP]+ [type=AP];
Eval:  unify(number gender, 1, 3);
       group(NP, 1);

Rule "NP12: noun phrase + genitive phrase"
Match: [type=NP] [type=GP]+;
Eval:  group(NP, 1);

Rule "GP3: series of genitive phrases"
Match: [type=GP] [type=GP]+;
Eval:  group(NP, 1);

Rule "NP9.1: conjunction + noun phrase"
Match: [pos~"adv"]? [pos~"conj"] [pos~"adv"]? [type=NP];
Eval:  delete(pos!~"adv", 1);
       delete(pos!~"conj", 2);
       delete(pos!~"adv", 3);
       group(CNP, 4);

Rule "NP10: noun phrase + conjunctional noun phrase"
Match: [type=NP] [type=CNP]+;
Eval:  group(NP, 1);

Rule "PP1.1.2: relative + a + noun phrase"
Match: [type=AP && head=[base~"relative"]] [pos~"prep"] [type=NP];
Eval:  delete(pos!~"prep", 2);
       group(PP, 3);

Rule "PP1.1: preposition + noun phrase"
Match: [pos~"adv" && base!~"pas|comme"]* [pos~"prep" && base!~"que"]+
       [pos~"adv" && base!~"pas|comme"]* [type=NP && head=[pos!~"pron"]];
Eval:  delete(pos!~"adv", 1);
       delete(pos!~"prep", 2);
       delete(pos!~"adv", 3);
       group(PP, 4);
```

```
Rule "PP1.1.1: preposition + noun phrase"
Match: [pos~"adv" && base!~"pas"]* [pos~"prep" && base!~"que"]+
       [type=NP && head=[pos~"pron" && base~"tous|tout|toutes|lequel"]];
Eval:  delete(pos!~"adv", 1);
       delete(pos!~"prep", 2);
       group(PP, 3);

Rule "PP1.2: preposition + genitive phrase"
Match: [pos~"adv" && base!~"pas|comme"]* [pos~"prep" && base!~"que"]+ [type=GP];
Eval:  delete(pos!~"adv", 1);
       delete(pos!~"prep", 1);
       group(PP, 2);

Rule "PP1.3: prepositional-like infinitive"
Match: [pos~"adv" && base!~"pas|comme"]* [pos~"prep" && base!~"que"]+
       [base~"me|te|se|nous|vous|le|lui"]? [pos~"V" && mood~"N"]
       ([pos~"adv"] | [type=AP && head=[pos~"adv"]])?;
Eval:  delete(pos!~"adv", 1);
       delete(pos!~"prep", 2);
       delete(pos!~"pron", 3);
       delete(pos!~"V", 4);
       delete(mood!~"N", 4);
       delete(pos!~"adv", 5);
       group(PP, 4);

Rule "PP1.4: preposition + named entity phrase"
Match: [pos~"prep"]+ [type=NE];
Eval:  delete(pos!~"prep", 1);
       group(PP, 2);

Rule "PP2.1: conjunction + prepositional phrase"
Match: [pos~"conj"] [type=PP];
Eval:  delete(pos!~"conj", 1);
       group(CPP, 2);
```

```
Rule "PP3: prepositional phrase + conjunctional prepositional phrase"
Match: [type=PP] [type=CPP]+;
Eval:  group(PP, 1);

Rule "NP11: noun phrase + prepositional phrases"
Match: [type=NP] [type=PP]+;
Eval:  group(NP, 1);

Rule "NP13.1: noun phrase surrounded by quotes"
Match: [pos~"art"]? [base~~"\""] [type=NP] [base~~"\""];
Eval:  unify(number gender, 1, 3);
       group(NP, 3);

Rule "NP13.2: noun phrase surrounded by parenthesis"
Match: [pos~"art"]? [base~~"\("] [type=NP] [base~~"\)"];
Eval:  unify(number gender, 1, 3);
       group(NP, 3);
```

Reguły związane z frazami czasownikowymi i strukturami reprezentującymi zdanie

```
Rule "VP2.7: verbal phrase: have to + infinitive"
Match: [base~"ne"]? [base~"avoir"] [base~"pas"]? [pos~"adv" && base!~"pas"]*
       [base~"devoir|pouvoir" && pos~"part" && part-tense~"past"] [base~"ne"]?
       [base~"pas"]? [pos~"adv" && base!~"pas"]? [base~"me|te|se|nous|vous|le|lui"]?
       [pos~"V" && mood~"N"] ([pos~"adv"] | [type=AP && head=[pos~"adv"]])?;
Eval:  delete(pos!~"V", 2);
       delete(pos!~"adv", 3);
       delete(pos!~"adv", 4);
       delete(pos!~"part", 5);
       delete(part-tense!~"past", 5);
       delete(pos!~"adv", 7);
       delete(pos!~"adv", 8);
       delete(pos!~"pron", 9);
       delete(pos!~"V", 10);
       delete(mood!~"N", 10);
       delete(pos!~"adv", 11);
```

```
       group(VP, 10);

Rule "HVP: verbal phrase help"
Match: [pos~"part|A"] [type=AP]+;
Eval:  unify(number gender, 1, 2);

Rule "VP2.6: verbal phrase: verb + infinitive"
Match: [base~"ne"]? [pos~"V"] (([orth~","] [pos~"adv" && base!~"pas"] [orth~","])
        | [pos~"adv" && base!~"pas"])? [base~"pas"]? [pos~"adv"]? [base~"que"]?
        ([base~"me|te|se|nous|vous|le|lui"] | [type=NP && head=[pos~"pron"]])?
        [pos~"V" && mood~"N"] ([pos~"A|part|adv"] | [type=AP])*;
Eval:  delete(pos!~"V", 2);
       delete(pos!~"adv|interp", 3);
       delete(base~"pas" && pos!~"adv", 4);
       delete(pos!~"conj", 6);
       delete(pos!~"pron", 7);
       delete(pos!~"V", 8);
       delete(mood!~"N", 8);
       delete(pos!~"A|part|adv", 9);
       group(VP, 8);

Rule "VP2.2: verbal phrase with auxilary verb"
Match: [base~"me|te|se|nous|vous|la"]? [base~"ne|non"]? [base~"pas"]?
       [pos~"V" && base~"avoir|etre"] [pos~"adv" && base!~"pas"]* [base~"pas"]?
       [orth~"été"/i]? ([pos~"adv"] | [type=AP && head=[pos~"adv"]])?
       [pos~"part" && part-tense~"past"] [type=AP]?;
Eval:  delete(pos!~"pron", 1);
       delete(pos!~"adv", 3);
       delete(pos!~"V", 4);
       delete(pos!~"adv", 5);
       delete(pos!~"adv", 6);
       delete(pos!~"part", 7);
       delete(pos!~"adv", 8);
       delete(pos!~"part", 9);
       delete(part-tense!~"past", 9);
       group(VP, 9);
```

```
Rule "VP2.2.1: verbal phrase with auxilary verb"
Match: [base~"me|te|se|nous|vous|la"]? [base~"ne|non"]? [base~"pas"]?
       [pos~"V" && base~"avoir|etre"] [pos~"adv" && base!~"pas"]* [base~"pas"]?
       ([pos~"adv"] | [type=AP && head=[pos~"adv"]])?
       [pos~"part" && part-tense~"past"] [type=AP]?;
Eval:  delete(pos!~"pron", 1);
       delete(pos!~"adv", 3);
       delete(pos!~"V", 4);
       delete(pos!~"adv", 5);
       delete(pos!~"adv", 6);
       delete(pos!~"adv", 7);
       delete(pos!~"part", 8);
       delete(part-tense!~"past", 8);
       group(VP, 8);

Rule "VP2.4: verbal phrase 'le futur proche' tense"
Match: [base~"aller"] [pos~"V" && mood~"N"];
Eval:  delete(pos!~"V", 2);
       delete(mood!~"N", 2);
       group(VP, 2);

Rule "VP2.5: verbal phrase: 'le passe recent' tense"
Match: [base~"venir"] [base~"de"] [base~"ne"]? [base~"pas"]?
       [base~"me|te|se|nous|vous|le|lui"]? [pos~"V" && mood~"N"];
Eval:  delete(pos!~"adv", 4);
       delete(pos!~"pron", 5);
       delete(pos!~"V", 6);
       delete(mood!~"N", 6);
       delete(pos!~"prep", 2);
       group(VP, 6);

Rule "VP1.1: verbal phrase with predicative"
Match: [base~"ne"]? [base~"pas"]? [base~"me|te|se|nous|vous|le|lui"]?
       [pos~"V" && base~"etre"] [base~"pas"]? ([pos~"A|part"] | [type=AP]);
Eval:  delete(pos!~"adv", 2);
       delete(pos!~"pron", 3);
       delete(pos!~"V", 4);
```

```
       delete(pos!~"adv", 5);
       delete(pos!~"A|part", 6);
       group(VP, 4);

Rule "VP2.3: verbal phrase; 'passe surcompose' tense"
Match: [type=VP && head=[base~"avoir|etre" && pos~"part" && part-tense~"past"]]
       [pos~"part" && part-tense~"past"];
Eval:  delete(pos!~"part", 2);
       delete(part-tense!~"past", 2);
       group(VP, 2);

Rule "VP2.8: verbal phrase: article + infinitive"
Match: [pos~"art"] [base~"ne"]? [base~"pas"]? [base~"me|te|se|nous|vous|le|lui"]?
       [pos~"V" && mood~"N"] ([type=AP] | [pos~"A|adv"])+;
Eval:  delete(pos!~"art", 1);
       delete(pos!~"adv", 3);
       delete(pos!~"pron", 4);
       delete(pos!~"V", 5);
       delete(mood!~"N", 5);
       group(VP, 5);

Rule "VP2.10: verbal phrase: reflexive verb + adj phrase"
Match: [base~"ne"]? ([base~"me|te|se|nous|vous|le|lui"]
        | [type=NP && head=[base~"y"]])
       [pos~"V" && mood!~"N"] [pos~"adv" && base!~"pas"]?
       [base~"pas"]?  ([type=AP] | [pos~"adv" && base!~"pas"])+;
Right: (se | [base!~"pas"] | [head=[]]);
Eval:  delete(pos!~"pron", 2);
       delete(pos!~"V", 3);
       delete(pos!~"adv", 4);
       delete(pos!~"adv", 5);
       delete(pos!~"A|adv|part", 6);
       group(VP, 3);

Rule "VP2.11: verbal phrase: verb + de + infinitive"
Match: [base~"ne"]? [pos~"V" && mood!~"N"] [base~"pas"]?
       [type=PP && head=[pos~"V" && mood~"N"]];
```

```
Eval:  delete(pos!~"V", 2);
       delete(pos!~"adv", 3);
       group(VP, 2);

Rule "VP2.1: general verbal phrase (non-infinitive)"
Match: [base~"ne"]? ([base~"me|te|se|nous|vous|le|lui|tout"]
        | [type=NP && head=[base~"y"]])? [pos~"V" && mood!~"N"] (([orth~","]
        [pos~"adv" && base!~"pas"] [orth~","]) | [pos~"adv" && base!~"pas"])?
        [base~"pas"]? [pos~"adv" && base!~"pas"]*;
Eval:  delete(pos!~"pron", 2);
       delete(pos!~"V", 3);
       delete(mood~"N", 3);
       delete(pos!~"adv", 4);
       delete(base~"pas" && pos!~"adv", 5);
       delete(pos!~"adv", 6);
       group(VP, 3);

Rule "VP2.9: general verbal phrase (infinitive)"
Match: [base~"ne"]? [base~"pas"]? [base~"me|te|se|nous|vous|le|lui"]?
       [pos~"V" && mood~"N"] [pos~"adv"]*;
Eval:  delete(pos!~"V", 4);
       group(VP, 4);

Rule "GP6: genitive-like infinitive"
Match: [pos~"prep" && base~"du|de|des|d'"] [type=VP && head=[pos~"V" && mood~"N"]];
Eval:  group(GP, 2);

Rule "IP1.1: participle phrase containing adjective phrase"
Match: [pos~"pron"]? [base~"ne|non"]? [pos~"adv"]? [pos~"part"] [base~"pas"]?
       [type=AP]+ (([type=NP] | [type=PP] | [type=GP] | [pos~"adv"])+)?;
Eval:  unify(number gender, 4, 6);
       delete(pos!~"pron", 1);
       delete(pos!~"adv", 2);
       delete(pos!~"part", 4);
       group(IP, 4);
```

```
Rule "IP1: participle phrase"
Match: [pos~"pron"]? [base~"ne|non"]? [pos~"adv"]? [pos~"part"] [base~"pas"]?
       (([type=NP] | [type=PP] | [type=GP] | [pos~"adv"])+)?;
Eval:  delete(pos!~"pron", 1);
       delete(pos!~"adv", 2);
       delete(pos!~"part", 4);
       group(IP, 4);

Rule "IP2.1: conjunction + participle phrase"
Match: [pos~"conj"] [type=IP];
Eval:  delete(pos!~"conj", 1);
       group(CIP, 2);

Rule "IP3: participle phrase + conjunctional participle phrase"
Match: [type=IP] [type=CIP]+;
Eval:  group(IP, 1);

Rule "NP14: noun phrase + participle phrase"
Match: [type=NP] ([type=IP] | [type=CIP])+;
Eval:  unify(number gender, 1, 2);
       group(NP, 1);

Rule "GP10: genitive phrase + participle phrase"
Match: [type=GP] ([type=IP] | [type=CIP])+;
Eval:  unify(number gender, 1, 2);
       group(GP, 1);


Rule "NP13: cast NUMs and NEs into noun phrases"
Match: ([type=NUM] | [type=NE]);
Eval:  group(NP, 1);

Rule "G1: series of non-verbal phrases"
Match: ([type=NP] | [type=CNP] | [type=PP] | [type=CPP] | [type=IP] |
       [type=CIP] | [type=GP] | [type=CGP] | [type=AP])
       ([type=NP] | [type=CNP] | [type=PP] | [type=CPP] | [type=IP] |
       [type=CIP] | [type=GP] | [type=CGP] | [type=AP])+;
```

```
Eval:  group(SNP, 1);

Rule "G3: series of non-verbal phrases surrounded with parenthesis"
Match: [orth~"\("] [type=SNP] [orth~"\)"];
Eval:  group(SNP, 2);

Rule "G2: series of non-verbal phrases separated with conjunction"
Match: [type=SNP] ([orth~","] [pos~"conj"] [type=SNP])+;
Eval:  group(SNP, 1);

Rule "S1.1: Normal sentence ..."
Match: (([pos~"prep"]? [pos~"adv"]? [pos~"relpron|intpron"]) | [pos~"csub"]
        | [pos~"conj"])?  [pos~"adv|prep|pron" && base!~"ne"]*
       ([pos~"adv|prep|pron"]* ([type=NP] | [type=SNP] | [type=IP] | [type=CIP])
       [pos~"adv|prep|pron"]*)* [type=VP] ([pos~"conj"] [type=VP])?
       ([pos~"adv|prep|pron"]* ([type=NP] | [type=SNP] | [type=PP] | [type=GP]
       | [type=IP] | [type=CIP] | [type=CNP]))+;
Eval:  delete(pos~"N", 2);
       delete(pos!~"relpron|intpron|csub|conj|adv|prep", 1);
       group(S, 4);

Rule "S1.2: Normal sentence ..."
Match: ([pos~"prep"]? [pos~"relpron|intpron"] | [pos~"csub"] | [pos~"conj"])?
       ([pos~"adv|prep|pron"]* ([type=NP] | [type=SNP] | [type=CNP] | [type=IP] |
       [type=CIP]) [pos~"adv|prep|pron"]*)+ [orth~","]? [type=VP] ([pos~"conj"]
       [type=VP])? [pos~"adv|prep|pron" && base!~"que|ne"]*
       ([pos~"adv|prep|pron" && base!~"que"]* ([type=NP] | [type=SNP] | [type=PP] |
       [type=GP] | [type=IP] | [type=CIP] | [type=CNP]))*;
Eval:  delete(pos!~"relpron|intpron|csub|conj|prep", 1);
       group(S, 4);
```

DODATEK B

Zawartość płyty CD-ROM

Na płycie CD-ROM znajduje się płytki parser *puddle* w postaci kodów źródłowych w języku C++. Do parsera dołączony jest zbiór reguł parsingu dla języka francuskiego wraz z plikiem zawierającym opis tagsetu. Ponadto, zawarty jest także słownik form fleksyjnych języka francuskiego w postaci wykorzystywanej przez wewnętrzny tagger parsera *puddle*. Szczegóły dotyczące instalacji i uruchomienia parsera znajdują się w pliku tekstowym *readme* znajdującej się na płycie CD-ROM.

Ponadto na płycie dołączamy program *dictionary compiler*, który służy do generowania słowników form fleksyjnych na potrzeby parsera *puddle*. Za pomocą programu *dictionary compiler* można wygenerować słownik form fleksyjnych języka francuskiego z innych źródeł niż wykorzystane w niniejszej pracy bądź też tworzyć słowniki form fleksyjnych dla innych języków. Program *dictionary compiler* umożliwia zatem wykorzystanie parsera puddle do analizy składniowej języków innych niż francuski. Szczegóły dotyczące programu *dictionary compiler* znajdują się w dołączonym pliku *readme*.

Płyta CD-ROM zawiera także niniejszą pracę w postaci dokumentu w formacie PDF.

Bibliografia

Abeillé A., Clément L., Kinyon A. (2000) Building a treebank for French [w:] *Proccedings of the Second Conference on Language Resources and Evaluation (LREC2000)*. Ateny, str. 87–94.

Abeillé A., Clément C., Toussenel F. (2003) Building a treebank for French [w:] *Treebanks: building and using parsed corpora*, red. Abeillé A. Springer, str. 165-188.

Appelt Douglas E., Israel David (1999) *Introduction to Information Extraction Technology. A Tutorial for IJCAI-99*. Sztokholm.

Boost C++ Libraries Documentation. Dostępne pod adresem: `http://www.boost.org/doc/` [Dostęp 18 maja 2009].

British Standards Institution (1981) *BS 6154:1981 Method of defining – syntactic metalanguage*. ISBN 0-580-12530-0.

Buczyński A., Przepiórkowski A. (2008) ♠ Demo: An Open Source Tool for Partial Parsing and Morphosyntactic Disambiguation [w:] *Proceedings of LREC*.

Chomsky, N. (1956) *Three models for the description language*. Cambridge, Massachussets.

Clément L. (2001) *Construction et exploitation d'un corpus syntaxiquement annoté pour le français*. Praca doktorska. Paryż.

Clément L., Sagot B., Lang B. (2004) *Morphology based automatic acquisition of large-coverage lexica*. Lizbona, LREC'04.

DOT Graphviz - Graph Visualization Software. Dostępne pod adresem: `http://www.graphviz.org/Documentation.php` [Dostęp 18 maja 2009].

Evert S. (2005) *The CQP Query Language Tutorial*. Raport techniczny. University of Stuttgart.

Evert S., Kermes H. (2003) Annotation, Storage, and Retrieval of Mildly Recursive Structures [w:] *Workshop for Shallow Processing of Large Corpora (SProLaC)*. Lancaster.

Gazdar G., Mellish C. (1989) *Natural Language Processing in Prolog*. Edinburgh.

Grune D., Jacobs C.J.H. (1998) *Parsing techniques - second edition*. Amsterdam.

ICU International Components for Unicode. Dostępne pod adresem: `http://site.icu-project.org/` [Dostęp 18 maja 2009].

Joshi A., Hopely P. (1997) A parser from antiquity [w:] *Natural Language Engineering*. Cambridge University Press, str. 291-294.

Jurafsky D., Martin J.H. (2000) *Speech and Language Processing*. New Jersey, Prentice & Hall.

Kinyon A. (2001) A language-independent shallow-parser compiler [w:] *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*. Tuluza, str. 330-337.

Marciniak M., Mykowiecka A., Przepiórkowski A. and Kupść A. (2000) Construction of an HPSG Treebank for Polish [w:] *Journée ATALA, 18–19 juin, Corpus annotés pour la syntaxe*. Paryż.

Marcus M. P., Santorini B., Marcinkiewicz M. A. (1993) Building a large annotated corpus of English: the Penn Treebank [w:] *Computational Linguistics*. Vol. 19.

Marimon M., Porta J. (2000) PoS Tagging and Partial Parsing - Bidirectional Interaction [w:] *Proceedings of II International Conference on Language Resources and Evaluation (LREC-2000)*. Ateny.

McCallum A., Li W. (2003) Early Results for Named Entity Recognition with Conditional Random Fields, Feature Induction and Web-Enhanced Lexicons [w:] *Proceedings of the Seventh Conference on Natural Language Learning*. Edmonton.

Neumann G., Braun C., Piskorski J. (2000) A divide-and-conquer strategy for shallow parsing of german free texts [w:] *Proceedings of ANLP-2000*. Seattle.

(New 2006) New B. (2006) Lexique3: une nouvelle base de doneées lexicales [w:] *Verbum ex machina. Actes de la Conférence Traitement Automatique des Langues Naturelles (TALN 2006)*. Louvain, Belgia, str. 892-900.

NLTK. Bird S., Klein E., Loper E. *Analyzing Text with Python and the Natural Language Toolkit*. Dostępne pod adresem: `http://www.nltk.org` [Dostęp 18 maja 2009].

Polski Komitet Normalizacyjny (1979) Polska Norma PN-I-06000. *PN-I-06000:1997 Wyposażenie biurowe – Maszyny do pisania i do przetwarzania tekstów – Układy znaków na klawiaturach alfanumerycznych.*

Przepiórkowski A. (2004) *The IPI PAN Corpus: Preliminary version*. Warszawa.

Przepiórkowski A. (2007) On heads and coordination in valence acquisition [w:] *Computational Linguistics and Intelligent Text Processing (CICLing 2007)*. Berlin, str. 50-61.

Przepiórkowski A. (2008) *Powierzchniowe przetwarzanie języka polskiego*. Wrocław, Akademicka Oficyna Wydawnicza EXIT.

Przepiórkowski A., Buczyński A. (2007) Spade: Shallow Parsing and Disambiguation Engine [w:] *Proceedings of the 3rd Language & Technology Conference*. Poznań, str. 340-344.

Przepiórkowski A., Krynicki Z., Dębkowski Ł., Woliński M., Janus D., Bański P. (2004) A search tool for corpora with positional tagsets and ambiguities [w:] *Proceedings of the Fourth International Conference on Language Resources and Evaluation*. Lizbona, str. 1235-1238.

Przestaszewski L. (2006) *Gramatyka języka francuskiego*. Warszawa, Wiedza Powszechna.

RFC 3629 (2003) *UTF-8, a transformation format of ISO 10646*. Dostępne pod adresem: `www.rfc-editor.org/rfc/rfc3629.txt` [Dostęp 18 maja 2009].

Sagot B. (2006) *Analyse automatique du français: lexiques, formalismes, analyseurs*. Praca doktorska. Paryż.

Sagot B., Clément L., de la Clergerie É. V., Boullier P. (2006) *The Lefff 2 syntactic lexicon for French: architecture, acquisition, use*. Genua, LREC'06.

Sha F., Pereira F. (2003) Shallow Parsing with Conditional Random Fields [w:] *Proceedings of HLT-NAACL, 2003*. Edmonton.

Steinberger R., Pouliquen B., Widiger A., Ignat C., Erjavec T., Tufiş D., Varga D. (2006) The JRC-Acquis: A multilingual aligned parallel corpus with 20+ languages [w:] *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC'2006)*. Genua.

Tiedemann J. (2007a) Improved Sentence Alignment for Movie Subtitles [w:] *Proceedings of RANLP '07*. Borovets, Bułgaria.

Tiedemann J. (2007b) Building a Multilingual Parallel Subtitle Corpus [w:] *Proceedings of CLIN 17*. Leuven, Belgia.

van Rijsbergen C. J. (1979) *Information Retrieval*. Londyn, Information Retrieval Group, University of Glasgow.

XCES *XML Corpus Encoding Standard*. Dostępne pod adresem: `http://www.xces.org` [Dostęp 18 maja 2009].

Xerces-C++ XML Parser. Dostępne pod adresem: `http://xerces.apache.org/xerces-c/` [Dostęp 18 maja 2009].