Marcin Woliński

Morfeusz 2

Dokumentacja techniczna i użytkowa

4 października 2019

Spis treści

Morfeusz 2 (`http://sgjp.pl/morfeusz`) stanowi nową implementację analizatora morfologicznego dla języka polskiego (Woliński, 2014; Kieraś and Woliński, 2017). Poprzednie wersje programu (Woliński, 2006) stały się podstawą wielu narzędzi przetwarzania języka, w szczególności kilku tagerów i parserów języka polskiego. Jednak znane ograniczenia programu sprawiły, że podjęliśmy decyzję o całkowitej reimplementacji.

Wersja druga programu, opracowana jako część infrastruktury Clarin-PL, różni się od porzednika kilkoma ważnymi ulepszeniami: oprócz modułu analizy zawiera również moduł generujący zadane formy; informację generowaną przez program wzbogacono o klasyfikację nazw własnych oraz kwalifikatory; dodano możliwość ładowania słowników w czasie pracy programu; wprowadzono podsystem modelujący zjawiska segmentacyjne, który pozwala np. opisać historyczne warianty języka różniące się zasadami pisowni łącznej i rozdzielnej słów określonych klas gramatycznych.

Morfeusz jest udostępniany z dwoma słownikami fleksyjnymi: domyślnym słownikiem SGJP, opartym na danych *Słownika gramatycznego języka polskiego* (Saloni et al., 2007, 2012, 2015) oraz słownikiem Polimorf (Woliński et al., 2012) łączącym SGJP z obszernymi, ale słabiej dopracowanymi w zakresie informacji gramatycznej danymi społecznościowego słownika `SJP.pl`.

1. Wprowadzenie – podstawowe pojęcia

SEGMENTEM (ang. *token*) nazywamy ciąg znaków w tekście w języku naturalnym interpretowany pod względem fleksyjnym. Ciąg taki z założenia nie zawiera odstępów, zwykle nie zawiera znaków interpunkcyjnych i zwykle pokrywa się ze słowem tekstowym; odstępstwa opisano dalej.

LEKSEM to abstrakcyjna jednostka języka zwana też wyrazem słownikowym. Jego umowny identyfikator nazywamy LEMATEM lub formą podstawową (ponieważ zwykle jest on wykładnikiem konwencjonalnie wybranej formy leksemu, np. bezokolicznika czasownika).

FORMĄ WYRAZOWĄ nazywamy segment zinterpretowany — przypisany do konkretnego leksemu i opisany co do jego funkcji gramatycznej. O segmentach w tekście mówimy, że są WYKŁADNIKAMI form fleksyjnych (a na zasadzie skrótu myślowego — wykładnikami leksemów).

Technicznie leksem uważamy za zbiór form wyrazowych o wspólnym lemacie, a formę wyrazową za trójkę ⟨*wykładnik, lemat, znacznik fleksyjny*⟩.

ANALIZA MORFOLOGICZNA polega na określeniu dla danego segmentu wszystkich form wszystkich leksemów, których może on być wykładnikiem. W procesie tym nie uwzględnia się kontekstu, w którym wystąpił dany segment. W językoznawstwie termin analiza morfologiczna odnosi się raczej do rozkładania wyrazów na na elementarne składniki morfologiczne (morfemy), być może sensowniej byłoby więc mówić o ANALIZIE FLEKSYJNEJ, niestety wydaje się, że to ten pierwszy termin utarł się w środowisku językoznawstwa komputerowego.

UJEDNOZNACZNIANIEM MORFOLOGICZNYM nazywamy określanie na podstawie kontekstu, którą z możliwych form wyrazowych realizuje dane wystąpienie segmentu.

Następujące po sobie analizę i ujednoznacznianie morfologiczne nazywa się żargonowo TAGOWANIEM.

Celem HASŁOWANIA (LEMATYZACJI) jest wskazanie dla każdego segmentu opisującej go jednostki słownika morfologicznego (leksemu). Jest to więc analiza morfologiczna (lub tagowanie) ograniczona tylko do części informacji o formach — do lematów.

Przybliżone hasłowanie polegające na odcięciu ze słów części zmieniającej się przy odmianie bywa nazywane stemowaniem. Metoda ta ma sens dla języków o ograniczonej fleksji, ale dla polskiego daje wyniki wysoce niezadowalające. W kontekście Morfeusza mówimy więc o hasłowaniu.

Operacją odwrotną do analizy morfologicznej jest SYNTEZA MORFOLOGICZNA — utworzenie wykładnika formy odmiany danej przez wskazanie lematu (identyfikatora leksemu) i żądanej charakterystyki fleksyjnej.

1.1. Analiza morfologiczna

Program Morfeusz wykonuje analizę morfologiczną dla języka polskiego. W obecnej wersji nie zawiera modułu zgadującego nieznane słowa (można więc powiedzieć, że jest słownikiem fleksyjnym).

W trybie analizy wejściem jest ciąg znaków, a wynikiem lista interpretacji wykrytych w tym ciągu segmentów (reprezentująca acykliczny graf fleksyjny). Interpretacja obejmuje: lemat, znacznik morfosyntaktyczny, informację o byciu nazwą własną, ewentualne kwalifikatory.

Oto przykład wyników działania programu dla tekstu „Mam próbkę analizy morfologicznej.”:

0	1	*Mam*	MAMA MAMIĆ MIEĆ	subst:pl:gen:f impt:sg:sec:imperf fin:sg:pri:imperf
1	2	*próbkę*	PRÓBKA	subst:sg:acc:f
2	3	*analizy*	ANALIZA	subst:sg:gen:f subst:pl:nom.acc.voc:f
3	4	*morfologicznej*	MORFOLOGICZNY	adj:sg:gen.dat.loc:f:pos
4	5	.	.	interp

Każdy wiersz tabeli zawiera jedną interpretację morfologiczną, kreski oddzielają grupy interpretacji dla poszczególnych segmentów. Tekst wejściowy został podzielony na segmenty (w szczególności kropka została oddzielona od napisu *morfologicznej*). Na prawo od segmentów podano odpowiadające im lematy, a w następnej kolumnie — znaczniki opisujące wartości kategorii gramatycznych charakteryzujące poszczególne formy.

Segmentowi *mam* zostały przypisane trzy interpretacje: jako forma liczby mnogiej rzeczownika MAMA, jako forma trybu rozkazującego czasownika MAMIĆ i wreszcie jako forma czasu teraźniejszego czasownika MIEĆ. Segment *analizy* został jednoznacznie przypisany do lematu ANALIZA, może on jednak być interpretowany zarówno jako forma liczby pojedynczej jak i mnogiej — w różnych przypadkach.

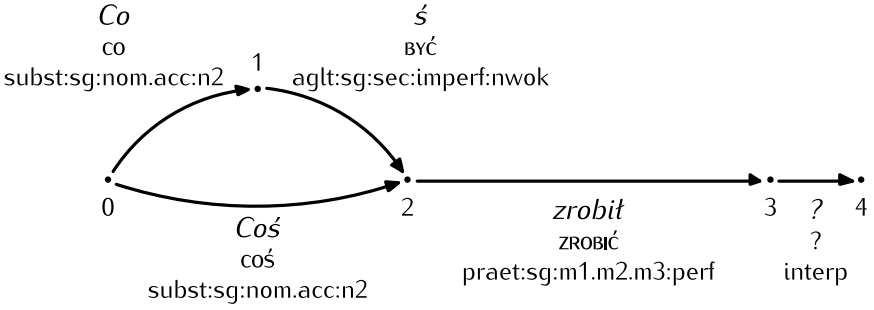
Znaczniki używane w programie Morfeusz są pozycyjne. Pierwsza pozycja określa klasę gramatyczną („część mowy”), następne pozycje reprezentują wartości kategorii gramatycznych przysługujących danej klasie. Na przykład znacznik subst oznacza rzeczownik, a po nim stoją wartości liczby, przypadka i rodzaju. Oznaczenia są w większości skrótami łacińskich nazw wartości.

Lematy podawane przez program są wykładnikami tradycyjnie wybranych form podstawowych leksemu, chyba że wymagane jest rozróżnienie homonimów. Wyróżniamy więc dwa leksemy PIEC: rzeczownikowy z lematem PIEC:S i czasownikowy — PIEC:V. W wypadku homonimii w obrębie tej samej klasy gramatycznej dodajemy jeszcze oznaczenia cyfrowe. Na przykład wyróżniamy dwa leksemy SŁAĆ: jeden z nich ma m.in. formę *ściele*, a drugi *śle*. Morfeusz dla wymienionych słów odpowie odpowiednio SŁAĆ:V1 i SŁAĆ:V2. Dla słowa *słał*, które może być interpretowane jako wykładnik obu rozważanych leksemów, Morfeusz przedstawi dwie interpretacje z tym samym znacznikiem, ale różnymi lematami.

Interpretacje podawane przez program zawierają oprócz wymienionych w przykładzie jeszcze informację, czy dany leksem jest nazwą własną, oraz kwalifikatory.

Morfeusz nie rozpatruje zależności przekraczających granicę odstępu międzywyrazowego. Czasami jednak ciąg nie zawierający spacji jest dzielony na kilka segmentów. Najbardziej oczywistym takim elementem są znaki interpunkcyjne oddzielane od poprzedzającego słowa. Ponadto dzielone są formacje typu *biało-czerwony* (ale nie *ping-pong* i *PRL-u*, jest to zadane przez słownik). Występujące w nich formy specjalne takie jak *biało* zostaną rozpoznane tylko w odpowiednim kontekście (a więc *biało-jakiś* lub *biało-*, to ostatnie ze względu na konteksty typu *biało- lub żółto-niebieski*).

Zdarza się, że podział danego ciągu znaków na segmenty jest niejednoznaczny, a w związku z tym interpretacje generowane przez Morfeusza w ogólności nie tworzą listy, ale graf acykliczny. Na przykład zdanie *Coś zrobił?* jest niejednoznaczne. Można je sparafrazować jako *Co zrobiłeś?*, gdzie *coś* jest interpretowane jako rzeczownik z doklejoną końcówką czasu przeszłego czasownika, lub też jako *Czy [on] coś zrobił?*, gdzie *coś* jest w całości interpretowane jako rzeczownik. Oto generowany dla tego przykładu graf fleksyjny:



Jest on reprezentowany w postaci następującego ciągu interpretacyj:

0	1	*Co*	CO	subst:sg:nom.acc:n2
1	2	*ś*	BYĆ	aglt:sg:sec:imperf:nwok
0	2	*Coś*	COŚ	subst:sg:nom.acc:n2
2	3	*zrobił*	ZROBIĆ	praet:sg:m1.m2.m3:perf
3	4	*?*	?	interp

Liczby w pierwszych dwóch kolumnach tabeli oznaczają punkty pomiędzy poszczególnymi segmentami tekstu interpretowanymi przez program. W tym przykładzie fragment tekstu od punktu 0 do 2 (czyli *coś*) może być interpretowany jako dwa segmenty: 0–1 *co* i 1–2 *ś* lub pojedynczy segment 0–2. Numery są przydzielane rosnąco, ale, jak widać, nie w każdym wariancie interpretacji tekstu numeracja musi być ciągła.

1.2. Synteza morfologiczna

Synteza morfologiczna może się odbywać w dwóch wariantach. W pierwszym podaje się lemat (np. *piec:s*) i otrzymuje listę wszystkich form zadanego leksemu — wykładników i znaczników morfosyntaktycznych (*piec, pieca, piecowi, ..., piece, pieców, ...*). W drugim wariancie podaje się lemat i pełny znacznik morfosyntaktyczny żądanej formy (np. subst:sg:dat:m3), a otrzymuje się jej wykładnik (np. *piecowi*) lub wykładniki, jeżeli istnieje kilka.

Lemat podawany jako podstawa do generowania musi mieć taką samą postać, jaką zwraca moduł analizy, ewentualnie z pominięciem części po dwukropku. Podanie modułowi odmiany lematu SŁAĆ:V1 spowoduje wygenerowanie form tego tylko leksemu (w wyniku znajdzie się więc forma *śle*, ale nie *ściele*). Podanie lematu SŁAĆ da w wyniku połączoną odmianę obu leksemów SŁAĆ z pełnymi lematami pozwalającymi przypisać poszczególne formy do poszczególnych leksemów.

W wypadku nielicznych czasowników z wariantywnym bezokolicznikiem konieczne jest podanie właściwego lematu. Na przykład dla BIEC/BIEGNĄĆ Morfeusz w analizie podaje lemat BIEC. Wygenerowanie formy inf:imperf dla tego lematu da wykładniki zarówno *biec*, jak i *biegnąć*.

2. Zawartość paczek dystrybucyjnych i instalacja

Morfeusz został pomyślany przede wszystkim jako moduł, który może być wykorzystywany programach pisanych przez użytkowników. W związku z tym ma on postać biblioteki dynamicznej (.so, .dll). Moduł jest napisany w C++. Podstawowy interfejs programistyczny (API) biblioteki jest obiektowy i wyrażony w C++.

Dla ułatwienia użytkowania Morfeusza z poprzednio stworzonymi programami zachowano proceduralny interfejs wyrażony w języku C. Wersja C jest binarnie zgodna z interfejsem poprzednich wersji Morfeusza i pozwala na użycie bieżącej wersji ze starszymi programami, ale nie daje dostępu do informacji, których stara wersja nie udostępniała.

Poprzednie wersje Morfeusza miały słownik wkompilowany w kod programu. W obecnej wersji chcemy pozwolić na łatwą manipulację słownika-

mi, w szczególności wymianę używanego słownika. Dlatego też możliwe są dwa sposoby skompilowania programu: z domyślnym słownikiem wkompilowanym i bez żadnego słownika, w której słownik w osobnym pliku trzeba wskazać przy inicjalizacji obiektu reprezentującego analizator.

Morfeusz może przetwarzać tekst zapisany kodowaniu w ISO8859-2, CP852, CP1250, UTF-8. Wyniki przetwarzania są zapisywane w tym samym kodzie co wejście.

Morfeusz jest kompilowany na serwerze słownika SGJP. Mechanizm generujący sprawdza co tydzień, czy w bazie danych słownika pojawiły się jakieś zmiany wprowadzone przez redaktorów, a jeżeli tak, generowany jest eksport tekstowy danych fleksyjnych, który następnie jest kompilowany do binarnej postaci dla analizatora i generatora Morfeusz 2 oraz budowane są nowe wersje pakietów z Morfeuszem dla wszystkich wspieranych platform sprzętowych (Linux, Windows, MacOSX).

2.1. Instalacja w systemach Debian, Ubuntu i pochodnych

Pakiety dla systemu Debian/Ubuntu są udostępniane w postaci repozytorium zgodnego z systemem Apt. Aby więc zapewnić automatyczną aktualizację Morfeusza w systemie, należy dodać owo repozytorium do listy systemowych źródeł pakietów podanymi poniżej poleceniami. Morfeusz jest obecnie kompilowany dla Ubuntu w trzech wersjach odpowiadających kolejnym dystrybucjom o przedłużonym wsparciu (LTS). Zalecamy następujący wybór repozytorium w zależności od wersji używanego sytemu:

— Ubuntu 14.04, 14.10, 15.04, 15.10 — wersja „trusty”,
— Ubuntu 16.04, 16.10, 17.04, 17.10 — wersja „xenial”,
— Ubuntu 18.04, 18.10, 19.04, ... — wersja „bionic”.

Użycie poniższych poleceń zapewnia dodanie do systemowej listy źródeł właściwej (automatycznie wybranej) wersji repozytorium:[1]

```
wget -O - http://sgjp.pl/apt/sgjp.gpg.key|sudo apt-key add -
sudo apt-add-repository http://download.sgjp.pl/apt/ubuntu
sudo apt update
```

Po ich wykonaniu można zainstalować w systemie wybrane pakiety Morfeusza, na przykład:

```
sudo apt install morfeusz2 morfeusz2-gui python3-morfeusz2
```

Zainstalowane pakiety będą podlegały systemowej procedurze aktualizacji.

2.2. Instalacja w innych systemach Linuksowych

W innych systemach linuksowych konieczne jest skorzystanie z paczek .tar.gz. Zawierają one skompilowaną bibliotekę (z wbudowanym słownikiem SGJP) i programy klienckie.

System Linux szuka bibliotek dynamicznych tylko w predefiniowanych katalogach systemowych. Dobrym miejscem dla bibliotek instalowanych z ręki

[1] Organizacja repozytorium została zmieniona w połowie 2019 r. Użytkowników poprzedniej wersji prosimy o usunięcie z pliku `/etc/apt/sources.list` zapisów dotyczących repozytorium sgjp.pl i ponowienie procedury konfiguracyjnej.

jest katalog `/usr/local/lib`, po skopiowaniu plików konieczne jest odświeżenie bazy bibliotek poleceniem `ldconfig`:

```
sudo cp -a ‹ścieżka_do_rozpakowanej_paczki_z_Morfeuszem›/lib/* /usr/local/lib/
sudo ldconfig
```

Programy wykonywalne można skopiować do katalogu `/usr/local/bin`:

```
sudo cp -a ‹ścieżka_do_rozpakowanej_paczki_z_Morfeuszem›/bin/* /usr/local/lib/
```

Jeżeli niemożliwe jest umieszczenie plików w katalogach systemowych, można wskazać położenie bibliotek ustawiając zmienną środowiskową

```
export LD_LIBRARY_PATH=‹ścieżka_do_rozpakowanej_paczki_z_Morfeuszem›/lib/
```

a programy wykonywalne wywoływać z jawną ścieżką lub dodać katalog do zmiennej `PATH`:

```
export PATH=‹ścieżka_do_rozpakowanej_paczki_z_Morfeuszem›/bin/
```

Moduł pythonowy można zainstalować poleceniem `easy_install` z pliku `.egg` pobranego ze strony (dla odpowiedniej wersji języka Python).

2.3. Instalacja w Windows

Dla systemu Windows przygotowano instalator wersji Morfeusza z interfejsem graficznym.

Do zastosowań programistycznych należy pobrać paczkę `.tar.gz` zawierającą bibliotekę dynamiczną i programy wiersza poleceń.

System Windows szuka bibliotek dynamicznych w katalogu bieżącym, w katalogu używającego programu `.exe` i w katalogach systemowych. W zależności od potrzeb należy umieścić plik `morfeusz2.dll` w jednej z tych lokalizacji.

Moduł pythonowy można zainstalować poleceniem `easy_install` z pliku `.egg` pobranego ze strony (dla odpowiedniej wersji języka Python). Plik `.egg` zawiera również bibliotekę Morfeusz ze słownikiem SGJP, więc do użytkowania Morfeusza wyłącznie z poziomu Pythona nie trzeba wcześniej instalować pod Windows innych modułów.

3. Użycie Morfeusza z poziomu C++

W tym punkcie omawiamy interfejs programistyczny Morfeusza 2 zdefiniowany w pliku `morfeusz2.h`:

```
#include "morfeusz2.h"
```

Objaśnienia podajemy na przykładach, aby pokazać sugerowany sposób korzystania z poszczególnych elementów, dokładne deklaracje poszczególnych składowych można znaleźć w pliku nagłówkowym.

Analizatora i generatora form używa się poprzez obiekt klasy `Morfeusz`. Wszystkie elementy API są definiowane w przestrzeni nazw `morfeusz`. Dalsze przykłady w tym punkcie zakładają, że są wykonywane w zasięgu deklaracji

```
using namespace morfeusz;
```

Jeżeli analiza/synteza ma się odbywać w wielu wątkach programu, to należy dla każdego wątku utworzyć osobną instancję. Bardziej szczegółowo kwestię tę omawiamy w punkcie 3.8.

3.1. Ścieżki wyszukiwania słowników

Klasa Morfeusz zawiera statyczną składową `dictionarySearchPaths` klasy `std::list<std::string>`, która reprezentuje listę nazw katalogów, w których program szuka plików słownikowych. Wyszukiwanie odbywa się od początku tej listy, ładowany jest pierwszy słownik o zadanej nazwie. Listę ścieżek można badać i modyfikować za pomocą metod klasy `std::list`, np. następujące wywołanie dodaje bieżący katalog jako preferowane miejsce wyszukiwania słowników:

```
Morfeusz::dictionarySearchPaths.push_front(".");
```

3.2. Tworzenie instancji modułu

Klasa `Morfeusz` jest abstrakcyjna, żeby uniknąć obecności w pliku nagłówkowym szczegółów implementacyjnych (a przez to uniknąć zależności od nich przy łączeniu z Morfeuszem i przyspieszyć kompilację). W związku z tym instancji Morfeusza nie tworzy się operatorem `new`, lecz za pomocą następującej metody klasowej:

```
Morfeusz *m=Morfeusz::createInstance();
```

Takie wywołanie powoduje utworzenie instancji Morfeusza korzystającej z domyślnego słownika (wkompilowanego w bibliotekę albo słownika o nazwie `SGJP`, jeżeli żaden nie jest wkompilowany).

Aby użyć innego słownika, należy podać jego nazwę w argumencie metody `createInstance()`:

```
Morfeusz *m=Morfeusz::createInstance("Polimorf");
```

Metoda `createInstance` sprawdza, czy zadany słownik jest już załadowany i jeżeli nie jest, to go ładuje. Słownik przechowywany jest w postaci dwóch plików, osobno dane do analizy i do syntezy (np. `SGJP-a.dict` i `SGJP-s.dict` w wypadku słownika `SGJP`). Jeżeli w programie będzie wykonywana tylko jedna z tych funkcji, a zależy nam na oszczędności pamięci, można spowodować załadowanie tylko jednego z nich (zob. typ wyliczeniowy `MorfeuszUsage`), np.:

```
Morfeusz *m=
    Morfeusz::createInstance("Polimorf",ANALYSE_ONLY);
```

Program utrzymuje pulę załadowanych słowników. Ustawienie w kolejnej instancji słownika o wcześniej użytej nazwie powoduje użycie wcześniej załadowanego słownika. Praca z wieloma słownikami wymaga utworzenia osobnej instancji dla każdego z nich.

3.3. Pobieranie informacji o bibliotece i słowniku

Następujące metody klasy `Morfeusz` pozwalają pobrać informacje o bibliotece i załadowanym słowniku.

Następująca funkcja pobiera oznaczenie wersji biblioteki (np. `2.0.0`):

```
std::string v = m->getVersion();
```

Funkcja `getDefaultDictName` podaje nazwę słownika używanego przez bezargumentowe `createInstance`:

```
std::string d = m->getDefaultDictName();
```

Następna funkcja pobiera notę prawnoautorską biblioteki Morfeusz:

```
std::string c = m->getCopyright();
```

Następny zestaw metod pozwala uzyskać informacje o słowniku załadowanym w danej instancji klasy `Morfeusz`. Funkcja `getDictID()` pobiera identyfikator słownika (zapisany w pliku binarnym):

```
std::string di = m->getDictID();
```

W wypadku słowników dystrybuowanych z Morfeuszem zapewniamy, że identyfikatory te jednoznacznie reprezentują każdą wydaną przez nas wersję słownika. Mają one postać np. `pl.sgjp.SGJP-2014.09.25` lub `pl.waw.ipipan.Polimorf-2014.09.2` Jednocześnie rezerwujemy identyfikatory zaczynające się ciągami `pl.sgjp` i `pl.waw.ipipan` dla słowników „firmowanych” odpowiednio przez zespół SGJP i IPI PAN.

Notę prawnoautorską załadowanego słownika można pobrać funkcją

```
std::string c = m->getDictCopyright();
```

3.4. Ustawianie opcyj

Wszystkie poniższe ustawienia dotyczą wyłącznie instancji modułu, na rzecz której są wykonywane. Dla każdej funkcji `set...` istnieje analogiczna bezargumentowa funkcja `get...`, która pozwala sprawdzić bieżąco obowiązującą wartość odpowiedniej opcji.

— `m->setCharset(UTF8)`
  Ustala kodowanie tekstu podawanego do modułu i przezeń oddawanego. Argumentem mogą być wartości typu `Charset` (`ISO8859-2`, `CP852`, `CP1250`, `UTF-8`).
— `m->setAggl("permissive")`
  Ta opcja pozwala wybrać zestaw reguł dołączania aglutynantów *-(e)ś*, *-(e)m*, *-(e)śmy* itd. Zestaw dostępnych wartości zależy od odpowiedniej deklaracji w pliku definiującym reguły łączenia segmentów (zob. 6.2.1). Prekompilowane słowniki dystrybuowane z Morfeuszem dopuszczają następujące wartości:
  — `"strict"` (domyślna) — dopuszczający rozsądny zbiór połączeń pojawiających się w tekście współczesnym, np. „gdyby·m” i „my·śmy”, ale nie „aligatora·ś”;

— `"permissive"` — dopuszczający więcej połączeń, np. „aligatora·ś [widział]”;
— `"isolated"` — tryb specjalny akceptujący wszystkie segmenty ze słownika jako osobne (np. samo *biało*), czyli dający odpowiedź, co mamy w słowniku. Tryb ten jest pomyślany na okoliczność ponownej analizy tekstu wcześniej posegmentowanego zgodnie z regułami Morfeusza, na przykład weryfikacji znakowania korpusu ze zmienioną wersją analizatora.

(Opcja wpływa tylko na analizę).

— `m->setPraet("composite")`
Opcja ta powoduje używanie różnych wersji tagsetu dla form czasu przeszłego czasowników. Jej implementacja i dostępne wartości są zdefiniowane przez plik reguł segmentacyjnych. W wypadku słowników prekompilowanych są to:
— `"split"` (domyślna) — formy syntetyczne czasu przeszłego analizowane jako dwa segmenty: *gniotł* i *em* (jak w poprzednich wersjach Morfeusza),
— `"composite"` — formy syntetyczne analizowane jako jeden segment; aglutynanty pojawiają się, jeżeli forma była faktycznie nieciągła w tekście.

— `m->setCaseHandling(STRICTLY_CASE_SENSITIVE)`
Przy analizie istotny jest problem wielkich i małych liter. Wyróżniamy z tego punktu widzenia następujące tryby (określone stałymi typu wyliczeniowego `CaseHandling`):
— `IGNORE_CASE` — W tym trybie rozróżnienie wielkich i małych liter jest ignorowane.
— `STRICTLY_CASE_SENSITIVE` — Wielka litera w słowniku dopasowuje się tylko do wielkiej litery w analizowanym tekście, ale mała litera dopasowuje się zarówno do małych jak i do wielkich liter w tekście, Na przykład zapis „Gdańskowi” w słowniku powinien powodować zaakceptowanie słów „Gdańskowi”, „GDAŃSKOWI”, „GdaŃsKowi”, ale nie „gdańskowi” ani „gDAŃSKOWI”.
— `CONDITIONALLY_CASE_SENSITIVE` (domyślny) — Ten tryb działa jak poprzedni, chyba że z powodu niezgodności wielkich liter miałyby zostać odrzucone wszystkie interpretacje. W tym trybie zostałyby zaakceptowane wszystkie słowa wymienione w poprzednim przykładzie, ale słowo *ale* dostałoby jedynie interpretacje pospolite, a odrzucone byłyby formy imion Ala i podobnych.

(Wpływa tylko na analizę. Przy generowaniu zawsze tworzony jest wariant zgodny z zapisem lematu w słowniku).

— `m->setWhitespaceHandling(APPEND_WHITESPACES)`
Opcja ta ustanawia sposób traktowania znaków odstępu przez analizator. Argument jest typu wyliczeniowego `WhitespaceHandling`:
— `SKIP_WHITESPACES` (domyślna) — odstępy ignorowane,
— `APPEND_WHITESPACES` — odstępy dołączane do poprzedzającego segmentu,
— `KEEP_WHITESPACES` — odstępy jako osobne segmenty.

W wypadku zastosowania opcji `APPEND_WHITESPACES` i `KEEP_WHITESPACES`

w wyniku skonkatenowania wszystkich tokenów będących wynikiem analizy dostaje się z powrotem wejściowy tekst, co bywa użyteczne. Aby to zapewnić w wypadku opcji `APPEND_WHITESPACES`, ewentualne odstępy na początku analizowanego tekstu są dołączane do pierwszego tokenu. W wypadku opcji `KEEP_WHITESPACES` każdy spójny ciąg odstępów tworzy token z formą hasłową będącą pojedynczym znakiem spacji (U+0020) i znacznikiem `sp` (tag o numerze 1). (Opcja wpływa tylko na analizę).

— `m->set_token_numbering(n)`
  — `separate` — segmenty w każdym wywołaniu funkcji analizującej są numerowane od pozycji 0,
  — `continuous` — w kolejnym wywołaniu funkcji analizującej pierwszy segment dostaje jako punkt początkowy punkt końcowy ostatniego segmentu utworzonego w poprzednim wywołaniu. Operacja ustawienia tej opcji zeruje numer segmentu.

  (Wpływa tylko na analizę).

3.5. Analiza

Operacja analizy fleksyjnej została zaimplementowana w dwóch wariantach: w pierwszym wyniki oddawane są przez iterator klasy `ResultsIterator`, w drugim — zestawiane w `std::vector`.

Przykład analizy z wektorem jako wynikiem:

```
vector<MorphInterpretation> r;
m->analyse("Załóż gąbkę na klawesyn.", r)
```

Przykład analizy w stylu iteratorowym:

```
ResultsIterator *r=m->analyse("Załóż gąbkę na klawesyn.");
while(r->hasNext()) {
    MorphInterpretation i=r->next();
    cout << i.startNode <<" "
        << i.endNode <<" "
        << i.orth  <<" "
        << i.lemma  <<" "
        << i.getTag(*morfeusz) <<" "
        << i.getName(*morfeusz) <<" "
        << i.getLabelsAsString(*morfeusz)
        << endl;
    }
```

Argumentem funkcji `analyse` może być `std::string` lub `char*`. Obiekt reprezentujący interpretację fleksyjną omawiamy w punkcie 3.7

3.6. Synteza

Argumentem funkcji generującej powinien być jeden lemat leksemu:

```
vector<MorphInterpretation> r;
m->generate("słać:v2", r);
```

Funkcja wypełnia wektor będący jej drugim argumentem wszystkimi formami zadanego leksemu. Tak samo jak w wypadku analizy formy są reprezentowane z użyciem klasy `MorphInterpretation`.

Drugi wariant funkcji generującej pozwala ograniczyć wynik do konkretnej formy. Zadaje się ją z użyciem identyfikatora znacznika `tagId`, zob. 3.7.

```
vector<MorphInterpretation> r;
m->generate("słać:v2", 17, r);
```

Wynikiem jak poprzednio jest wektor, ponieważ leksem może mieć wiele wykładników z danym znacznikiem.

3.7. Reprezentacja interpretacji fleksyjnych

W powyższym przykładzie analizy widać też sposób reprezentacji interpretacji morfologicznej (klasa `MorhpInterpretation`). Pola `startNode` i `endNode` typu `int` reprezentują wierzchołki grafu fleksyjnego (por. p. 1.1). Pola `orth` i `lemma` typu `std::string` to odpowiednio wykładnik formy i jej lemat.

Pozostałe elementy interpretacji — znacznik fleksyjny, klasyfikacja nazw własnych i kwalifikatory — są reprezentowane pośrednio. W strukturze `MorhpInterpretation` występują numeryczne identyfikatory `tagId`, `nameId`, `labelsId`. Ich wartości reprezentują skatalogowane elementy występujące w słowniku źródłowym. W wypadku znaczników fleksyjnych (tagów) odwzorowanie numerów w napisy jest zadane przez plik definicyjny tagsetu (por. 6). W wypadku kwalifikatorów numery odpowiadają pojawiającym się kombinacjom kwalifikatorów. Identyfikatory te można zdekodować za pomocą obiektu klasy `IdResolver`. Instancję tej klasy skojarzoną z daną instancją Morfeusza można pobrać za pomocą metody `Morfeusz::getIdResolver()`.

W przykładzie użyto pomocniczych metod klasy `MorphInterpretation`, które wywołują odpowiednie metody klasy `IdResolver` skojarzonej z załadowanym słownikiem. Bezpośrednie wywołania wyglądałyby analogicznie do następującego:

```
IdResolver & idr = m->getIdResolver()
std::string tag = idr.getTag(i.tagID);
```

Odwrotna funkcja `getTagId()` pozwala na podstawie znacznika w postaci napisu uzyskać jego identyfikator (potrzebny np. w trójargumentowej funkcji `generate`). Identyfikatory symboli klasyfikujących nazwy własne rozwiązuje się za pomocą metody `getName()`. Jeśli zaś chodzi o kwalifikatory, zdefiniowane są dwie metody dostępu: `getLabelsAsString()` zwraca zapis kwalifikatorów w takiej postaci, jaka wystąpiła w odpowiedniej kolumnie źródłowego zapisu słownika (np. `"pot.|arch."`); natomiast metoda `getLabels()` zwraca obiekt typu `std::set<std::string>` reprezentujący zbiór kwalifikatorów.

Ideą tej pośredniej konstrukcji jest możliwość wymiany używanego obiektu `IdResolver`, a tym samym dostosowanie reprezentacji odpowiednich elementów do struktur danych konkretnego programu korzystającego z Morfeusza. Na przykład w parserze Świgra znaczniki fleksyjne są termami prologowymi o wewnętrznej strukturze, a nie napisami. W związku z tym zamiast do-

myślnego obiektu `IdResolver` Świgra używa własnej implementacji odwzorowującej identyfikatory tagów w odpowiednie termy.

Klasa `IdResolver` zawiera kilka elementów użytecznych w tym kontekście. Metoda `getTagsetId()` daje dostęp do identyfikatora tagsetu (z numerem wersji) używanego przez załadowany słownik. Dzięki temu identyfikatorowi program korzystający z własnej reprezentacji tagsetu może się upewnić, że wersja tagsetu zapamiętana w programie jest taka sama jak w Morfeuszu. Metoda `getTagsCount()` zwraca liczbę tagów zawartych w danym tagsecie. Identyfikatory tagów to liczby od o do `getTagsCount()-1`. Korzystając z tej wiedzy można np. przejrzeć je wszystkie i przygotować reprezentację tagsetu potrzebną w konkretnym programie.

Analogicznie można postąpić z klasyfikacją nazw własnych i kwalifikatorami.

3.8. Użytkowanie modułu w programach wielowątkowych

Ze względu na problemy z przenośnością między platformami zdecydowaliśmy się nie korzystać w implementacji z mechanizmów synchronizacji między wątkami. W związku z tym korzystanie z Morfeusza w programie wielowątkowym wymaga odrobiny uwagi.

Każdy obiekt klasy `Morfeusz` powinien być używany (jednocześnie) tylko w jednym wątku programu. Szczególnej uwagi wymaga metoda `createInstance`, która potencjalnie ładuje słowniki, a więc jej użycie jednocześnie w wielu wątkach może spowodować załadowanie słownika wielokrotnie, co spowoduje wyciek pamięci. Użycie tej metody w wielu wątkach, jeżeli wiadomo, że dany słownik jest już załadowany, prawdopodobnie nie spowoduje problemów, ale nie gwarantujemy takiego użycia.

Najbezpieczniejszym sposobem użycia Morfeusza w programie wielowątkowym jest zapewne tworzenie instancyj klasy `Morfeusz` w jednym wątku, a następnie oddawanie ich we władanie odpowiednich wątków roboczych, w których będzie odbywać się analiza.

Użyteczna w tym kontekście może być metoda `Morfeusz::clone()`, która tworzy instancję na podstawie już istniejącej, zachowując ustawienia wszystkich opcji, w szczególności z tym samym załadowanym słownikiem.

3.9. Wersja API w czystym C

Alternatywą dla opisanego dotąd interfejsu jest API wyrażone w języku C zdefiniowane w pliku `morfeusz2_c.h`. Powtarza ono dokładnie struktury danych i funkcje używane w poprzednich wersjach analizatora Morfeusz. Definicje te powinny pozwolić na użycie nowej wersji Morfeusza w programach napisanych z myślą o dotychczasowej wersji analizatora. Należy jednak pamiętać, że nie pozwalają one na dostęp do informacji, które w poprzednich wersjach były nieobecne, a więc do symboli nazw własnych i kwalifikatorów. Ten wariant API nie modyfikuje też w żaden sposób prezentowanych danych językowych, tak więc prezentowane jest np. hasłowanie według obecnych zasad Morfeusza.

4. Użycie Morfeusza w języku Java

Interfejs języka Java powtarza dość dokładnie interfejs w języku C++. Szczegółowe informacje można znaleźć w dystrybuowanych plikach JavaDoc.

5. Użycie Morfeusza w języku Python

Interfejs języka Python różni się nieco od interfejsu C++ ze względu na własności języka i przyjęte w środowisku pythonowym obyczaje.

Użycie Morfeusza wymaga zaimportowania w programie biblioteki i stworzenia obiektu reprezentującego Morfeusza: `morfeusz2`:

```
import morfeusz2
morf = morfeusz2.Morfeusz()
```

Obiekt utworzony bez podawania parametrów konstruktora będzie używał domyślnego słownika SGJP i domyślnych ustawień opcji.

Informacje o załadowanym słowniku można pobrać poprzez wywołanie `morf.dict_id()` i `morf.dict_copyright()`.

Dwie najważniejsze metody klasy Morfeusz to `analyse` oraz `generate`. Pierwsza z nich zwraca graf analizy morfoskładniowej dla podanego napisu w postaci listy trójek uporządkowanych reprezentujących pojedyncze interpretacje poszczególnych segmentów (czyli krawędzie w grafie analizy). Każda trójka składa się z indeksów węzła początkowego i końcowego danej krawędzi oraz z interpretacji morfoskładniowej, stanowiącej etykietę krawędzi. Interpretacja to piątka uporządkowana zawierająca:

— formę tekstową,
— lemat (formę bazową/hasłową),
— znacznik morfoskładniowy,
— listę informacji o „pospolitości” rzeczownika (np. nazwa pospolita, marka, nazwisko),
— listę kwalifikatorów stylistycznych (np. daw., pot., środ., wulg.) i dziedzinowych (np. bot., zool.).

Segmenty nieznane słownikowi otrzymują specjalny znacznik ign oraz lemat równy formie tekstowej.

```
for text in (u'Jaś miał kota', u'Coś zrobił?', u'qwerty'):
    print(text)
    analysis = morf.analyse(text)
    for interpretation in analysis:
        print(interpretation)
```

Metoda `generate` zwraca listę interpretacji morfoskładniowych (w postaci piątek, jw.) wszystkich form, dla których podany tekst stanowi formę bazową:

```
morf.generate(u'piec')
```

W odróżnieniu od analyse, generate akceptuje tylko napisy stanowiące pojedyncze słowo (bez spacji), w przeciwnym przypadku zostanie zgłoszony wyjątek.

Parametry konstruktora klasy Morfeusz (podano wartości domyślne):

1. `analyse=True, generate=True`
   Parametry pozwalające zadecydować o (nie)załadowaniu słownika analizatora lub generatora. Niezaładowanie danego słownika oszczędza pamięć programu.
2. `dict_name=None, dict_path=None`
   Parametry pozwalające na załadowanie innego słownika niż domyślny.
3. `expand_dag=False`
   Po ustawieniu tego parametru na True metoda analyse zwraca listę wszystkich ścieżek w grafie analizy (zamiast listy krawędzi). UWAGA: przy tym ustawieniu nie są zwracane identyfikatory węzłów w grafie!
4. `expand_tags=False`
   Przy domyślnym ustawieniu tego parametru na False niektóre interpretacje mogą zawierać znaczniki z tzw. notacją kropkową, stanowiące tak naprawdę spakowaną reprezentację kilku znaczników (np. adj:pl:nom.voc:f.n:pos należy rozumieć jako cztery możliwe znaczniki: adj:pl:nom:f:pos, adj:pl:nom:n:pos, adj:pl:voc:f:pos, adj:pl:voc:n:pos). Ustawienie True powoduje rozbicie takich spakowanych znaczników na osobne krawędzie w grafie analizy. Opcja dotyczy wyników zarówno analizy, jak generacji.
5. `aggl=None`
   Parametr sterujący obsługą interpretacji odpowiadających segmentom niewystępującym samodzielnie (w szczególności cząstka mobilna aglt, skróty wymagające kropki na końcu).
   Ustawienie domyślne None (równoważne strict) pozwala na takie interpretacje tylko w połączeniu z odpowiednim innym segmentem (np. forma pseudoimiesłowowa praet czasownika dla aglt, kropka dla skrótu). Ustawienie isolated powoduje uzwględnienie interpretacyj „niesamodzielnych" również wtedy, kiedy dany segment jest osobnym słowem w tekście. Ustawienie permissive powoduje zastosowanie dodatkowych, warunkowych reguł segmentacyjnych zdefiniowanych w słowniku (np. reguły pozwalającej na łączenie aglt z przymiotnikami).
6. `praet=None`
   Parametr sterujący sposobem interpretacji form czasownikowych praet połączonych z cząstką mobilną aglt lub partykułą przypuszczającą by.
   Ustawienie domyślne None (równoważne split) powoduje rozbicie na formę czasownikową, cząstkę mobilną i by. Ustawienie composite powoduje interpretację jako jeden segment praet lub cond.
7. `separate_numbering=True`
   Przy domyślnym ustawieniu tego parametru na True, przy każdym wywołaniu metody analyse powstały graf ma wierzchołki numerowane od zera. Ustawienie False powoduje w kolejnych wywołaniach kontynuację numerowania wierzchołków z uprzednio stworzonego grafu (czyli pozwala na tworzenie jednego grafu w mniejszych fragmentach).
8. `case_handling=morfeusz2.CONDITIONALLY_CASE_SENSITIVE`
   Parametr sterujący możliwością przypisania lematu zaczynającego się wielką literą.
   Ustawienie domyślne `morfeusz2.CONDITIONALLY_CASE_SENSITIVE` pozwala na przypisanie lematu pisanego wielką literą (o ile taki lemat dla danego segmentu jest notowany w słowniku), jeśli są tylko takie lema-

ty lub segment zaczyna się wielką literą (innymi słowy: jeśli segment zaczyna się małą literą i można mu przypisać lemat pisany małą literą, to nie zostanie mu przypisany lemat pisany wielką literą). Ustawienie `morfeusz2.STRICTLY_CASE_SENSITIVE` nie dopuszcza przypisania lematu pisanego wielką literą segmentowi pisanemu małą literą. Ustawienie `morfeusz2.IGNORE_CASE` powoduje nieuwzględnianie wielkiej/małej litery.

9. `whitespace=morfeusz2.SKIP_WHITESPACES`
   Parametr sterujący obsługą białych znaków.
   Ustawienie domyślne `morfeusz2.SKIP_WHITESPACES` powoduje nieuwzględnianie białych znaków w grafie analizy. Ustawienie `morfeusz2.KEEP_WHITESPACES` powoduje przypisanie ciągom białych znaków pomiędzy segmentami osobnych krawędzi w grafie analizy. Ustawienie `morfeusz2.APPEND_WHITESPACES` powoduje przyłączenie ciągów białych znaków na końcu poprzedzających je segmentów (lub na początku pierwszego segmentu, jeśli tekst zaczyna się od ciągu białych znaków).

6. Przygotowanie własnych słowników

Kompilacja słownika do postaci binarnej wymaga użycia osobnego narzędzia (zaimplementowanego w Pythonie), które dystrybuowane jest w pakiecie `morfeusz_builder`.

6.1. Słownik główny

Zasadniczą częścią danych, z których korzysta program, jest słownik segmentów. Słownik w formie źródłowej ma postać pięciu kolumn rozdzielonych tabulatorami (U+0008):

```
#!DICT-ID pl.sgjp.demo-2014.09.01
#<COPYRIGHT>
This is a demo dictionary, use freely.
No copyright, copyleft, copyup or copydown.
#</COPYRIGHT>
Gdańsk    Gdańsk  subst:sg:acc:m3  geograficzna
Gdańsk    Gdańsk  subst:sg:nom:m3  geograficzna
Gdańska   Gdańsk  subst:sg:gen:m3  geograficzna
Gdański   Gdańsk  subst:pl:nom:m3  geograficzna
Gdańskiem Gdańsk  subst:sg:inst:m3 geograficzna
funkcja   funkcja subst:sg:nom:f   pospolita
funkcjach funkcja subst:pl:loc:f   pospolita
funkcjami funkcja subst:pl:inst:f  pospolita
funkcje   funkcja subst:pl:acc:f   pospolita
funkcje   funkcja subst:pl:nom:f   pospolita
funkcje   funkcja subst:pl:voc:f   pospolita     rzad.
funkcji   funkcja subst:pl:gen:f   pospolita
funkcji   funkcja subst:sg:gen:f   pospolita
funkcjo   funkcja subst:sg:voc:f   pospolita     rzad.
```

```
funkcjom  funkcja subst:pl:dat:f   pospolita
funkcyj   funkcja subst:pl:gen:f   pospolita     arch.|matem.
```

Na początku pliku słownikowego można umieścić informacje identyfikujące, które zostaną przeniesione do postaci binarnej słownika. Jeżeli pierwszy wiersz pliku słownikowego zawiera napis `#!DICT-ID`, dalsza część tego wiersza (po odstępie) zostanie przez program kompilujący słowniki zanotowana jako identyfikator dostępny z API programu przez `getDictID()`. Ponadto jeżeli w następnej linii jest napis `#<COPYRIGHT>`, to wiersze aż do zawierającego wyłącznie napis `#</COPYRIGHT>` zostaną potraktowane jako nota copyrightowa dostępna przez `getDictCopyright()`.

Jeżeli na wejściu podano wiele plików słownikowych, to za informację dotyczącą właśnie kompilowanego słownika zostanie uznana ta z pliku podanego jako pierwszy. (W szczególności w pełni poprawne jest przygotowanie osobnego pliku zawierającego wyłącznie te informacje, a podanie właściwych danych słownikowych w następnych argumentach).

Jak wspomniano wcześniej, każda wersja słownika dostępna ze stron `sgjp.pl` ma w tym polu symbol jednoznacznie identyfikujący słownik i jego wersję. Użytkownicy Morfeusza są uprzejmie proszeni o zadbanie o jednoznaczną identyfikację słowników, które zdecydują się dystrybuować. W szczególności słowniki powstałe przez dołożenie form do dystrybuowanych przez nas powinny mieć identyfikator odróżniający od oryginałów i notę copyrightową tłumaczącą pochodzenie poszczególnych części danych i wynikające z tego prawa.

Każdy wiersz w pliku słownikowym opisuje jedną formę fleksyjną, wykładniki form mają prawo się powtarzać, jeśli towarzyszą im różne lematy lub znaczniki. Wiersz jest podzielony na 5 kolumn rozdzielonych znakiem tabulacji (U+0008). Ostatnia kolumna jest opcjonalna — występuje tylko przy wybranych wierszach. Oto zawartość poszczególnych kolumn:

1 wykładnik formy
2 lemat (identyfikator leksemu)
3 znacznik morfosyntaktyczny
4 klasyfikacja nazw własnych
5 kwalifikator(y rozdzielone pałką)

Większość wierszy opisuje po prostu kompletne słowa, ale są wyjątki, na przykład formy niesamodzielne typu biało-, trój-, przyrostki -ś, -em, -eśmy, ..., -ń, -że, które opisują segmenty występujące niesamodzielnie, a więc wymagające połączenia z innym segmentem, aby utworzyć kompletne słowo (zob. p. 6.2).

6.2. Reguły łączenia segmentów

Przy analizie program rozpoznaje dopuszczalne zbitki segmentów. Każdemu segmentowi przypisywany jest typ, w pliku `segmenty.dat` zdefiniowane są ciągi typów segmentów, które mają być akceptowane. Ciąg taki należy dopasować do fragmentu napisu wejściowego od odstępu do odstępu.

6.2.1. Format zapisu

Informację w pliku interpretuje się w obrębie wiersza.

Znak # sygnalizuje, że część wiersza na prawo od niego jest komentarzem.

Puste wiersze w pliku nie mają znaczenia. Ignorować należy też znaki odstępu na początku i końcu wiersza.

Plik składa się z 3 części rozpoczynających się nagłówkiem o nazwie ujętej w nawiasy kwadratowe, np. `[lexemes]`.

Nazwy typów segmentów są ciągami liter, cyfr i znaków podkreślenia _.

Część `[options]` definiuje zbiór dostępnych opcji dotyczących aglutynacji i wariantów tagsetu.

```
[options]
aggl=strict permissive isolated
praet=split composite
```

Napisy podane po znaku równości stają się dopuszczalnymi wartościami odpowiednich opcji programu. Pierwsza wartość na liście jest domyślna. Opcje te są implementowane poprzez użycie w dalszej części pliku w konstrukcjach warunkowych

```
#ifdef split
… reguły …
#else
… reguły …
#endif
```

Część `[tags]` składa się z dwóch kolumn rozdzielonych odstępem.

Pierwsza kolumna zawiera nazwę typu segmentu, druga kolumna zawiera szablon tagu. Jeżeli tag danej formy ze słownika dopasowuje się do szablonu, to należy tej formie przypisać dany typ segmentu.

Szablon tagu w najprostszej postaci jest kompletnym tagiem. Może on też zawierać symbol `%` oznaczający dowolny ciąg znaków. Szablon musi dopasować się do całego tagu, tak więc `a%` dopasowuje się do tagu zaczynającego się literą ‘a’.

Próbę dopasowania tagu należy prowadzić w kolejności zadanej w pliku i wstrzymać po pierwszym sukcesie (w szczególności na końcu listy jest specyfikacja z szablonem `%`, która podaje domyślny typ segmentu).

Część `[lexemes]` składa się z trzech obowiązkowych kolumn rozdzielonych odstępem i zapisów opcjonalnych.

Pierwsza kolumna zawiera nazwę typu segmentu, druga zawiera lemat, trzecia kolumna zawiera szablon tagu jak w poprzedniej sekcji.

Jeżeli lemat i początek tagu danej formy ze słownika pasują do zapisu w drugiej kolumnie, należy danej formie przypisać podany typ segmentu. Ta informacja przebija typ wynikający z interpretacji poprzedniej części pliku (została pomyślana jako mechanizm traktowania w sposób wyjątkowy leksemów o nietypowych własnościach w obrębie swojej klasy).

Na końcu specyfikacji może znaleźć się zapis

Część `[combinations]` określa dopuszczalne zestawienia typów segmentów.

Sekcja ta zawiera też definicje makr w notacji zapożyczonej z preprocesora C:

```
#define wsz_interp (interp|kropka|przecinek|dywiz|polpauza)*

#define moze_interp(segmenty) wsz_interp segmenty wsz_interp
```

Makra mogą mieć argumenty, jak w drugim przykładzie (istotny jest brak odstępu między nazwą makra i nawiasem!).

Wiersz reprezentujący regułę łączenia segmentów zawiera ciąg typów segmentów rozdzielonych odstępami (niezerowa liczba „białych znaków"), np.:

```
praet_sg_agl agl_sg
```

Rozpoznanie napisu według takiego wiersza powoduje emisję odpowiedniego ciągu segmentów z ich indywidualnymi lematami i tagami.

Między nazwami typów może też stać znak > (z opcjonalnymi odstępami wokół):

```
nie> naj> adj_sup
```

Zapis taki oznacza, że segment po lewej stronie znaku > nie generuje własnego segmentu na wyjściu. Jego forma ortograficzna jest dołączana na początku formy i lematu następnego segmentu. Znacznik takiego segmentu jest ignorowany.

Symbol > można dokleić do dowolnego wyrażenia w nawiasach. Makra jak np. `liczba_rzymska` też muszą być dodatkowo zamknięte w nawiasach, nawiasy wewnątrz definicji makra nie wystarczą. Tak więc poniższy zapis jest błędny:

```
moze_interp( liczba_rzymska> dywiz> (adj_pos|adv_pos) )
```

a kolejny — poprawny:

```
moze_interp( (liczba_rzymska)> dywiz> (adj_pos|adv_pos) )
```

Doklejenie > do wyrażenia w nawiasach powoduje dodanie > do wszystkich atomowych symboli segmentów wewnątrz. Przykłady (z prawej strony równoważne wyrażenie):

```
(a b c d)> ---> (a> b> c> d>)
(a> (b | c) d>)> ---> (a> (b> | c>) d>)
```

Po nazwie typu mogą też stać operatory: `?` — segment opcjonalny, `*` — zero lub więcej wystąpień, `+` — jedno lub więcej wystąpień. Jeżeli operatory te mają się odnosić do więcej niż jednego segmentu, odpowiedni ciąg nazw typów musi zostać ujęty w nawiasy `()`. Przykład:

```
adja+ adj # ceglastoróżowy
(adja dywiz)+ adj # angielsko-francusko-polski
```

W wypadku zbiegu oznaczenia > i symboli krotności, najpierw idzie >:

```
dig>* dig # Liczba zapisana jako ciąg cyfr
```

6.2.2. Przykładowe zestawienia segmentów

Przedstawiamy listę przykładowych połączeń segmentów, ale wiążąca jest definicja w pliku `segmenty.dat` w kodzie źródłowym analizatora:

1. Syntetyczny czas przeszły:
   praet:...(:agl) + agl:...
   np. *gniotł·em*, *czytali·śmy*.
   Reguły łączenia zapewnią zgodność liczby między obu elementami. Wynikiem są dwa segmenty ze swoimi opisami ze słownika.
2. Przyłączenie końcówek czasu przeszłego do innych części mowy:
   ? + agl:...
   np. *gdyby·ście*, *aligatora·ś*
   wynik j.w.; dopuszczalny pierwszy element zależy od wybranej opcji
3. Syntetyczny lub analityczny tryb warunkowy:
   praet:...(:nagl) + *by* + agl:...
   praet:...(:nagl) + *by*
   by + agl:...
   np. *gniótł·by·m*, *gniótł·by*, *by·śmy [gnietli]*
   Wynikiem są wejściowe segmenty ze swoimi opisami ze słownika.
4. Złożone formy przymiotnikowe z adja:
   (adja -)$^{+}$ + adj:...
   adja^{+} + adj:...
   np. *polsko·-·angielski*, *biało·-·żółto·-·czerwony*;
   różowo·czerwony, *blado·żółty*, *trzecio·klasowy*
   Wynikiem są wejściowe segmenty ze swoimi interpretacjami.
5. Złożone formy przymiotnikowe z numcomp
   numcomp^{+} + adj:...
   np. *trzy·klasowy*, *dwudziesto·cztero·miejscowy*
   Wynikiem są wejściowe segmenty ze swoimi interpretacjami.
6. Niesamodzielna forma leksemu *on* dołączająca się do przyimków:
   prep:...(:wok) + *ń*
   np. *do·ń*, *przeze·ń*
   Wynikiem są wejściowe segmenty ze swoimi interpretacjami.
7. Apozycja rzeczowników z dywizem:
   subst:... + - + subst:...
   np. *kobieta·-·prezydent*, *chłop·-·robotnik*
   Wynikiem są wejściowe segmenty ze swoimi interpretacjami.
8. Derywacja prefiksalna:
   pref + ?
   np. *eurosodoma*, *e-długopis*, *e-listowny*, *ponakapywać*, *bezargumentowy*
   Wynikiem jest jeden segment, prefiks przyłączony zarówno do formy tekstowej jak i do lematu, znacznik z drugiego składnika.
   Ten mechanizm ma służyć do uwzględnienia pojawiających się w tekście form derywowanych, których nie mamy w słowniku. Zapewne będzie kilka typów pref różniących się tym, do czego mogą się dołączyć. Ta reguła

powinna być używana tylko wtedy, jeśli formacji z prefiksem nie mamy w słowniku.

9. Formacje z leksemem wzmacniającym *że*:
   [?]+[*ż(e)*]
   czytaj·że, *słuchajcie·ż*, *potrzebował·że·by·ś* (z Lema)
   Wynikiem są wejściowe segmenty ze swoimi interpretacjami.

10. Liczby
    [cyfra]$^{+}$
    np. *12348*
    Uwzględniamy to tutaj, bo liczby są reprezentowane ciągami cyfr potencjalnie dowolnej długości. Dzięki temu słownik główny może pozostać skończonym słownikiem.

Ponadto wszystkie wymienione kombinacje mogą mieć jeszcze dodany znak interpunkcyjny na końcu.

Literatura

Witold Kieraś and Marcin Woliński. Morfeusz 2 – analizator i generator fleksyjny dla języka polskiego. *Język Polski*, XCVII(1):75–83, 2017.

Zygmunt Saloni, Włodzimierz Gruszczyński, Marcin Woliński, and Robert Wołosz. *Słownik gramatyczny języka polskiego*. Wiedza Powszechna, Warszawa, 2007.

Zygmunt Saloni, Marcin Woliński, Robert Wołosz, Włodzimierz Gruszczyński, and Danuta Skowrońska. *Słownik gramatyczny języka polskiego*. Warsaw, 2nd edition, 2012. URL `http://sgjp.pl/`.

Zygmunt Saloni, Marcin Woliński, Robert Wołosz, Włodzimierz Gruszczyński, and Danuta Skowrońska. *Słownik gramatyczny języka polskiego*. `http://sgjp.pl`, 3 edition, 2015.

Marcin Woliński. Morfeusz — a practical tool for the morphological analysis of Polish. In Mieczysław A. Kłopotek, Sławomir T. Wierzchoń, and Krzysztof Trojanowski, editors, *Intelligent Information Processing and Web Mining*, Advances in Soft Computing, pages 503–512. Springer-Verlag, Berlin, 2006.

Marcin Woliński. Morfeusz reloaded. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Hrafn Loftsson, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation, LREC 2014*, pages 1106–1111, Reykjavík, Iceland, 2014. ELRA. ISBN 978-2-9517408-8-4. URL `http://www.lrec-conf.org/proceedings/lrec2014/index.html`.

Marcin Woliński, Marcin Miłkowski, Maciej Ogrodniczuk, Adam Przepiórkowski, and Łukasz Szałkiewicz. PoliMorf: a (not so) new open morphological dictionary for Polish. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation, LREC 2012*, pages 860–864, Istanbul, Turkey, 2012. ELRA.