Utworzenie uniwersalnej, otwartej, repozytoryjnej platformy hostingowej i komunikacyjnej dla sieciowych zasobów wiedzy dla nauki, edukacji i otwartego społeczeństwa wiedzy

Program Strategicznego Narodowego Centrum Badań i Rozwoju

Raport 14.2.2 Konstrukcja słownika polszczyzny dawnej dla wybranego zbioru tekstów

Agnieszka Mykowiecka, Piotr Rychlik, Jakub Waszczuk

---

1. Pierwszy etap budowy słownika

1.1 Ogólna charakterystyka zadania budowy słownika

Niniejszy raport prezentuje prace prowadzone w ramach projektu SYNAT, a których celem jest zaproponowanie metody budowy słownika języka polskiego zawierającego formy dawne. Zgodnie z opracowanymi na początku projektu i przedstawionymi w raporcie R14.1.1 założeniami, proces tworzenia słownika rozpoczyna się od przetworzenia już istniejącego słownika papierowego, który następnie jest rozszerzany informacjami uzyskanymi bezpośrednio z tekstów i odbywa się w następujący sposób:

- skanowanie papierowej wersji słownika, zamiana wyników skanowania na tekst (OCR), poprawa błędów,
- zdefiniowanie struktury wewnętrznej słownika,
- konwersja ciągłego tekstu na zdefiniowaną strukturę,
- wzbogacanie zawartości słownika na podstawie informacji wewnętrznych oraz innych dostępnych źródeł.

Raport 14.1.4 zawierał opis zagadnień związanych z pierwszymi 3 wymienionymi etapami tworzenia słownika. W niniejszym raporcie w pierwszej części zostaną pokrótce przedstawione te omówione już uprzednio etapy tworzenia słownika z uwzględnieniem wprowadzonych zmian i uzupełnień. W kolejnym rozdziale opisane zostaną zaproponowane pierwsze wprowadzone w słowniku rozszerzenia. Ostatnia część raportu poświęcona jest opisowi opracowanego programu do manualnej edycji słownika.

1.2 Przetwarzanie wstępne tradycyjnego słownika

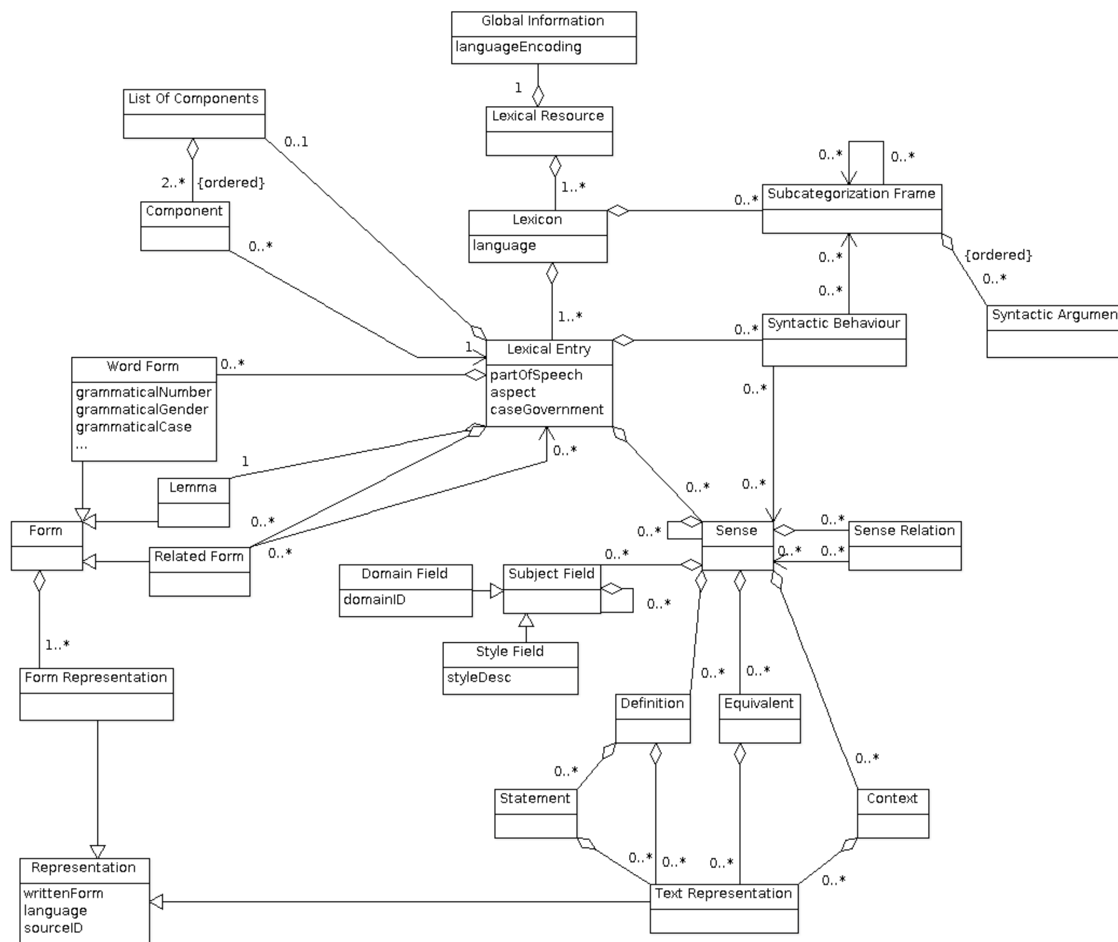
Pierwszym krokiem przy tworzeniu elektronicznej wersji słownika było zeskanowanie książkowej wersji i przetworzenie uzyskanych obrazów standardowym programem OCR. Plik wynikowy (w dwukolumnowym formacie Word) został przekształcony na postać czysto tekstową, w której wyróżnione zostały hasła. Program konwertujący tekst w pierwszym kroku wydzielił zasadniczą część słownika zawierającą opisywane hasła od wstępu, strony tytułowej i stron redakcyjnych. Na tym etapie przetwarzania program wykorzystał układ tekstu na stronach słownika, który w części zawierającej hasła (i tylko w niej) jest układem dwukolumnowym. Proces identyfikacji haseł okazał się dość złożony. W wersji książkowej hasła oraz ich opisy umieszczone są w jednym paragrafie, przy czym definiowane hasło, jak również podhasła, wyróżnione są pogrubioną czcionką. Niestety, nie najlepsza jakość skanów spowodowała, że program OCR często nie rozpoznawał prawidłowo czy użyta została czcionka pogrubiona, czy też nie. Aby rozpoznać prawidłowo hasła wykorzystano dodatkowo informację dotyczącą układu typograficznego, a dokładniej wielkości wcięcia wierszy.

W pliku wynikowym każde rozpoznane hasło wraz z podhasłami, ich definicjami i przykładami użycia, zostało zapisane w jednej linii tekstu. Plik ten został następnie przejrzany i część zauważonych systematycznych błędów została poprawiona automatycznie. Następnie całość została poddana ręcznej korekcie.

1.3 Ustalenie struktury słownika

W raporcie 14.1.1 opisano główne założenia, które przyjęto przy opracowywaniu standardu zapisu treści słownika. Słownik będzie miał postać plików XML w formacie i o zawartości

zgodnej z obowiązującym obecnie standardem LMF (*Lexical Markup Framework*, http://www.lexicalmarkupframework.org/); Wybrane rozszerzenie MRD określające schemat typowego słownika w wersji obsługiwanej komputerowo (*Machine-readable Dictionary*). Rysunek 1 przedstawia diagram klas słownika historycznego zgodny z modelem MRD opisanym w specyfikacji standardu LMF. Informacja o pochodzeniu słowa jest reprezentowana przez atrybut sourceID klasy <Representation>. Dzięki temu informacja o tekście źródłowym jest zachowana zarówno dla każdej formy <Form> hasła słownikowego, jak również dla każdego kontekstu <Context>, w ramach którego hasło występuje. Ponadto projekt słownika historycznego korzysta (chociaż w małym stopniu) z rozszerzeń NLP syntax (klasy <Syntactic Behaviour>, <Subcategorization Frame> i <Syntactic Argument>) oraz NLP semantic (klasa <Sense Relation>) głównego pakietu LMF.



Rysunek 1: Schemat hasła

1.4 Struktura słownika

Konstruowany słownik w formacie LMF jest plikiem XML o następującej strukturze:

```
<LexicalResource dtdVersion="16">
  <GlobalInformation>
    <feat att="languageCoding" val="ISO 639-6" />
  </GlobalInformation>
  <Lexicon>
    <feat att="language" val="polh" />
```

```
    <LexicalEntry id="lex.1">
...
    </LexicalEntry>
    ...
    <LexicalEntry id="lex.n">
...
    </LexicalEntry>
  </Lexicon>
</LexicalResource>
```

Główną częścią tej struktury jest leksykon (<Lexicon>) stanowiący zbiór haseł leksykalnych (<LexicalEntry>), z których każde hasło reprezentuje jeden leksem języka historycznego (polh). Każdy element <LexicalEntry> posiada atrybut o nazwie id określający unikalny identyfikator hasła w leksykonie. Ogólna postać wartości tego atrybutu jest następująca: lex.n lub lex.n:m, gdzie n i m są dodatnimi liczbami całkowitymi i oznaczają numer hasła głównego (n) oraz numer podhasła (m). Każde hasło reprezentowane przez element <LexicalEntry > opatrzone jest komentarzem:

<!—INDEX nnn –>,

gdzie liczba nnn oznacza numer linii w wejściowym pliku tekstowym, w której dane hasło zostało umieszczone. Dodatkowo, w komentarzu może się znajdować słowo VERIFY, ale tylko w tych przypadkach, w których konieczna jest weryfikacja poprawności translacji przez lingwistę.

Z każdym hasłem związana jest jedna forma podstawowa (<Lemma>), oraz zbiór form wyrazowych (<WordForm>). Z każdą formą związana jest jej reprezentacja w języku pisanym (<FormRepresentation>). W szczególności, jeśli dane słowo posiada kilka wariantów ortograficznych, bedą one przedstawione w osobnych instancjach klasy <FormRepresentation>. Leksem może posiadać wiele znaczeń. Każde z nich jest reprezentowane przez klasę <Sense>. Znaczenie leksemu może być określone za pomocą definicji (<Definition>) (wraz z uzupełnieniem w polu <Statement>), lub poprzez podanie przykładowych kontekstów użycia słowa (klasa <Context>). W każdym z tych przypadków do opisu znaczenia leksemu służy klasa <TextRepresentation>.

1.5 Reguły translacji

Konwersja wersji tekstowej na wersję w formacie LMF przeprowadzana jest w kilku etapach. W pierwszym etapie następuje klasyfikacja haseł do odrębnych klas. Opisy haseł należących do jednej klasy mają podobną strukturę. W drugim kroku następuje właściwa konwersja zgodnie z określonymi dla danej klasy haseł regułami. Po przetworzeniu tekstu całego słownika następuje faza trzecia, w której ustalane są identyfikatory tych haseł, do których istnieje odwołanie w tekście innych haseł. Usuwane są wtedy redundantne wpisy, jak również tworzona jest lista haseł niezidentyfikowanych, do których odwołuje się w tekście, a które nie znalazły się w słowniku. Ostatnią fazą będzie ręczna weryfikacja wygenerowanych struktur LMF.

Około 82% wszystkich haseł w przetwarzanym słowniku ma następujący wzorzec:

{słowo} «definicja»: cytowania.

lub

{słowo} 1. «definicja-1»: cytowania-1;

2. «definicja-2»: cytowania-2;

...

n. «definicja-n»: cytowania-n.

Zaraz po definiowanym słowie (pierwotne pogrubienie jest tu zastąpione przez nawiasy klamrowe) następuje jeden lub więcej ponumerowanych bloków. Każdy z tych bloków zaczyna się definicją otoczoną nawiasami kątowymi '«' i '»', po której występuje dwukropek a następnie przykłady użycia definiowanego wyrazu rozdzielone średnikami. Prawie każdy przykład zawiera na samym końcu informację na temat źródła pochodzenia.

Przykłady interpretacji różnego typu haseł znajdują się w raporcie 14.1.4. Tu przypomnimy tylko jeden z nich:

Przykład {błaźnić} «mamić, oszukiwać, otumaniać, ogłupiać»: Nie dajcie sie błaźnić żonam swoim JWuj; Ich umiejętność błaźnię Bar; ∼ z kogo «drwić, kpić z kogo»: Naśmiewam się, błaźnię z kogo. Błaźnisz ze mnie SMącz. Hasło to ma następującą postać w formacie LMF:

```
<LexicalEntry id="lex.704">
  <!-- INDEX 629 VERIFY -->
  <Lemma>
    <FormRepresentation>
      <feat att="writtenForm" val="błaźnić" />
      ...
    </FormRepresentation>
  </Lemma>
  <SyntacticBehaviour senses="sense.179">
    <feat att="writtenForm" val="błaźnić z kogo" />
    <feat att="language" val="polh" />
    <feat att="sourceID" val="srpsdp:SMącz" />
  </SyntacticBehaviour>
  <Sense>
    <Definition>
      <TextRepresentation>
        <feat att="writtenForm"
              val="mamić, oszukiwać, otumaniać, ogłupiać" />
        <feat att="language" val="pol" />
        <feat att="sourceID" val="srpsdp" />
      </TextRepresentation>
    </Definition>
    <Context>
      <TextRepresentation>
        <feat att="writtenForm" val="Nie dajcie sie błaźnić żonam swoim" />
        <feat att="language" val="polh" />
        <feat att="sourceID" val="srpsdp:JWuj" />
      </TextRepresentation>
    </Context>
    <Context>
      <TextRepresentation>
        <feat att="writtenForm" val="Ich umiejętność błaźnię" />
        <feat att="language" val="polh" />
        <feat att="sourceID" val="srpsdp:Bar" />
      </TextRepresentation>
    </Context>
  </Sense>
  <Sense id="sense.179">
    <Definition>
      <TextRepresentation>
```

```
        <feat att="writtenForm" val="drwić, kpić z kogo" />
        <feat att="language" val="pol" />
        <feat att="sourceID" val="srpsdp" />
      </TextRepresentation>
    </Definition>
    <Context>
      <TextRepresentation>
        <feat att="writtenForm"
              val="Naśmiewam się, błaźnię z kogo. Błaźnisz ze mnie" />
        <feat att="language" val="polh" />
        <feat att="sourceID" val="srpsdp:SMącz" />
      </TextRepresentation>
    </Context>
  </Sense>
</LexicalEntry>
```

Przyjęte reguły translacji haseł można podsumować a sposób następujący:

- W przypadku haseł wielowyrazowych (np. *zableszczyć oczy*) ich składniki opisane są jako elementy listy <ListOfComponents> i opatrzone identyfikatorami lex.xxx1, lex.xxx2 ... lex.xxxn, gdzie lex.xxxi jest identyfikatorem hasła odpowiadającego i-temu elementowi ciągu wielowyrazowego. Jeśli składowa hasła złożonego nie występuje w słowniku dokładnie w takiej formie, w jakiej występuje w tym haśle, program automatycznie dołącza do słownika tę formę jako propozycję nowego hasła z komentarzem o koniecznej weryfikacji wpisu.
- W przypadku czasowników zwrotnych, czyli takie, które zawierają zaimek zwrotny „się”, zaimek ten jest traktowany jako nieodłączna część czasownika i dla takich haseł nie jest tworzona struktura <ListOfComponents>.
- W bardziej złożonych hasłach definiowane są także jego podhasła (np. *plac, ∼ dać, ∼ otrzymać*) Podhasła zawierające związki frazeologiczne są umieszczone jako osobne wpisy w słowniku. COm
- Podana w niektórych hasłach rekcja (np. *błaźnić* {∼ *z kogo*} reprezentowana jest przez element ¡SyntacticBehaviour¿, przy czym treść wymagań nie jest interpretowana, tylko zapisywana w postaci tekstowej jako wartość atrybutu ”writtenForm”.
- Niektóre hasła zawierają listę wariantywnych jego form. Wszystkie warianty są wymienione w elemencie ¡Lemma¿ struktury LMF jako kolejne <FormRepresentation>.
- W niektórych hsłach podane są formy odmienione, np. *zakon, -a* || *-u* W tym przypadku do hasła *zakon* dołączone zostaną dwie struktury <WordForm> odpowiadające formom „zakona” (zakon-a) i „zakonu” (zakon-u).
- Osobną (nieliczną) klasą haseł są takie, które składają się z dwóch członów rozdzielonych znakiem średnika lub dwukropkiem, np. *liść; ∼ wolności* W tych przypadkach, zarówno definicje jak i cytaty dotyczą jedynie fragmentu tekstu występującego po średniku (dwukropku). Hasło takie rozbijane jest na dwa wejścia słwonikowe.

Teksty haseł, które nie pasują do żadnego z przewidzianych wzorców zostaną przekształcone w elementy <LexicalEntry> opatrzone komentarzem ERROR i zawierające tylko element <Lemma>. Będą one następnie ręcznie uzupełnione.

```
< LexicalEntry id="lex.xxxx">
  <-- INDEX yyyy ERROR -->
    <FormRepresentation>
      <feat att="writtenForm" val="text, który może jest lematem" />
      <feat att="language" val="polh" />
```

```
      <feat att="sourceID" val="srpsdp" />
    </FormRepresentation>
</LexicalEntry>
```

2. Uzupełnianie słownika o oznaczenia części mowy

Opisy haseł w słowniku Reczka nie zawierają informacji o części mowy. Ponieważ przy automatycznym przetwarzaniu tekstów jest to informacja podstawowa, pierwszym dokonanym rozszerzeniem zawartości słownika było automatyczne określenie części mowy (POS — ang. *part-of-speech*). Wyniki tych prac przedstawione zostały na warsztacie *Cultural Heritage* konferencji LREC (Mykowiecka, Rychlik, Waszczuk, 2012). Oznaczenia zaproponowane w wyniku działania algorytmu będą sprawdzone na etapie ręcznej korekty słownika.

Opracowany algorytm wykorzystuje informacje morfologiczne dostępne dla ekwiwalentów współczesnych, czyli słów opisujących znaczenie danego wyrazu. Analiza haseł wykazała, że możliwe są następujące sytuacje:

- opis zawiera tylko jedno słowo, np. {palcat} «berło»
- opis zawiera wiele słów, np. {panownik} «władca, panujący»
- jeden z ekwiwalentów jest frazą, np. {parsk} «dół, kopiec ziemny na kartofle, piwnica (za domem), ziemianka, loch»
- samo hasło stanowi frazę, np. {na barzego wsadzić}.

Algorytm nie zakłada dokonywania analizy składniowej opisu hasła. Wszystkie elementy opisów wyszukiwane są w elektronicznej bazie leksykalnej współczesnej polszczyzny – Polimorf. Jeżeli w wyniku analizy morfologicznej otrzymany zostanie więcej niż jeden tag POS, wybór najbardziej prawdopodobnego tagu odbywa się w dwóch krokach. W pierwszym, uzyskane wyniki analizowane są za pomocą kilku dedykowanych reguł, które w niektórych sytuacjach pozwalają na podjęcie wiarygodnej decyzji. Poniżej zacytowane są wszystkie zdefiniowane reguły:

- ˆ [pos=inf] [orth="się"] => inf, np. gładzić się {upiękniać się}
- ˆ [pos=pact] [orth="się"] => subst
- ˆ [pos=pact] [orth! ="się"] => pact
- ˆ [orth="o"] [pos=adj] => adj, np. garbonosy$_{adj}$ {o orlim$_{adj}$ nosie$_{subst}$}
- [pos=inf] => inf
- [pos=subst, case=nom] => subst, *hiszpańskie*$_{adj}$ *wino*$_{subst}$ *czerwone*$_{adj}$ *(z*$_{prep}$ *Alcante*$_{subst}$*)*
- [pos=ger, case=nom] => ger, np. *wytyczanie*$_{ger,subst}$ *granic*$_{subst}$

W powyższych regułach ˆ oznacza początek sekwencji, 'orth' oznacza odwołanie się do samego słowa, a 'pos' do nazwy części mowy. Możliwe jest także wyspecyfikowanie przypadka ('case') lub wartości innych cech morfologicznych. Dopuszczone jest zarówno żądanie równości wartości jak i ich nierówności (!=). Wynikowa nazwa części mowy podana jest po napisie '=>'.

Jeżeli żadna ze zdefiniowanych reguł nie pozwoli na wybór oznaczenia części mowy, wybierany jest wynik najczęstszy. Ze względu na fakt, że praktyczny brak kontekstu nie umożliwia użycia taggera, w głosowaniu biorą udział wszystkie interpretacje morfologiczne elementów

Tablica 1: Statystyki tagów POS

liczba tagów POS	liczba haseł
-	605
0	174
1	17236
2	888
3	78
4	11

POS labels	number of entries
subst	8828
verb	4718
adj	3051
adv	868
ger	522
ppas	229
pact	141
part	146
comp	62
conj	46

opisu. Jeżeli więcej niż jedna część mowy dostanie taką samą największą liczbę głosów, wynikiem algorytmu jest lista propozycji. W tabeli 1 podane są statystyki uzyskanych wyników, z których wynika, że zdecydowana większość opisów uzyskała jednoznaczny opis. Hasła oznaczone przez '-', to hasła, które są tylko odnośnikami do innych haseł, nie mają swojego opisu, wiec nie mogły być analizowane przy wykorzystaniu opisywanego algorytmu. Uzyskają one taką samą etykietę POS jak słowa, do których się odwołują. 174 hasła posiadają opisy, które nie zostały umieszczone w słowniku języka współczesnego Polimorf i opisywana metoda nie pozwala na przypisanie im żadnej części mowy.

Wstępna ewaluacja algorytmu dokonana na 108 pierwszych hasłach wykazała, że 87% otrzymała prawidłowe etykiety POS, a 4 słowa nie otrzymały żadnej etykiety.

3. Uzupełnianie słownika o nowe formy

3.1 Wstęp

Uzupełnianie słownika nowymi formami w oparciu o teksty historyczne wymaga realizacji następujących zadań:

- Podział tekstu wejściowego na słowa.
- Transliteracja poszczególnych słów.
- Wyszukiwanie w słowniku form podobnych.

3.2 Segmentacja na słowa

Podział tekstu na słowa jest zwykle zadanie łatwym, czasem wymagającym dodatkowych reguł przetwarzania w zależności od pochodzenia tekstu. Przykładowo, w danych IMPACT, oprócz standardowego podziału na znakach kontrolnych, należy uwzględnić dodatkowo przenoszenie słów z wykorzystaniem dywizu. Dla transkrypcji tekstów pochodzących z danych IMPACT przygotowany został niezależny moduł, który można znaleźć w repozytorium pod adresem `svn://chopin.ipipan.waw.pl/synat/tools/impact-tokenizer`.

3.3 Transliteracja

Przyjmujemy założenie, że tranlisteracja odbywa się na poziomie słów. Przygotowanie modułu transliteracji sprowadzało się do implementacji reguł przygotowanych przez lingwistkę w postaci programu komputerowego. Przygotowany zestaw reguł można scharakteryzować następująco:

- Reguły są postaci $s_1 \to s_2$, gdzie s_1 oznacza ciąg znaków tekstu wejściowego, który w ramach transliteracji powinien zostać zamieniony na ciąg znaków s_2 w alfabecie wyjściowym.
- Reguły mają charakter kontekstowy: można zadawać warunki na napisy bezpośrednio po lewej i po prawej stronie wejściowego ciągu s_1.
- Zestaw reguł tworzy kaskadę: na regułach określony jest porządek liniowy i jeśli względem dwóch zachodzących na siebie podsłów s_1 i s_2 słowa wejściowego można zastosować dwie reguły z zestawu, stosowana jest pierwsza z nich (mniejsza w relacji porządku).
- Reguły mogą być niedeterministyczne.

Na poziomie implementacji dodajemy następujący element specyfikacji działania kaskady: reguły powinny być stosowane tak długo, jak długo istnieje przynajmniej jedna reguła postaci $s_1 \to s_2$ taka, że s_1 jest podsłowem słowa wejściowego. UWAGA: można skonstruować taki zestaw reguł, że działanie zgodne z tym założeniem będzie prowadzić do „zapętlenia się” procesu transliteracji.

Obecna implementacja nie uwzględnia reguł niedeterministycznych postaci $s_1 \to s_2 \vee s_3$. Reguły tego typu upraszczane są do wersji deterministycznej $s_1 \to s_2$. Jedną z przyczyn tej decyzji jest fakt, że w większości zastosowań chcemy aby transliteracja prowadziła do jednego wyniku, czyli działała w sposób deterministyczny.

W celu implementacji modułu transliteracji zgodnego z powyższym opisem przygotowany został prosty EDSL (*Embedded Domain Specific Language*), pozwalający definiować reguły jak najbardziej zbliżone w swojej formie do oryginalnej postaci zadanej przez lingwistkę. EDSL został skonstruowany na bazie *kombinarów* biblioteki Parsec (`http://hackage.haskell.org/package/parsec-3.1.2`), popularnej bibilioteki języka Haskell pozwalającej na definiowanie parserów dla szerokiej klasy gramatyk, również kontekstowych. Następująca lista przedstawia część udostępnionych przez powstałą bibliotekę transliteracji kombinatorów:

- `>#>`, `#>` – reguła transliteracji, przykładowo `"ow" #> "ów"`.
- `.|`, `.|.` – operator lub, pozwala po lewej stronie reguły transliteracji deklarować listę napisów: `"a" .|  "á" .|. "ä" .|. "?" >#> "a"`.
- `>+>` – sekwencyjne połączenie dwóch reguł, pozwala np. na zadawanie prawego i lewego kontekstu: `vowel >+> ("y" #> "j")`

Ponieważ definiowanie reguł transliteracji sprowadza się w rzeczywistości do definiowania parserów, przygotowując zestaw reguł można wykorzystywać również funkcje zdefiniowane na poziomie samej biblioteki Parsec.

Kod modułu transliteracji można znaleźć w repozytorium pod adresem `svn://chopin.ipipan.waw.pl/synat/tools/transcription`, natomiast zestaw reguł transliteracji dla danych IMPACT można znaleźć w pliku `svn://chopin.ipipan.waw.pl/synat/tools/transcription/Text/Transcript/Impact.hs`.

3.4 Wyszukiwanie form podobnych

Niech D będzie zbiorem słów nad alfabetem Σ. Zbiór D będziemy nazywać *słownikiem*. Przez $d(x, y)$ będziemy oznaczać miarę odmienności między słowami x oraz y przyjmującą wartości z przedziału $[0, \infty)$. Wyszukiwanie form podobnych względem słowa x przy zadanym progu k polega na znalezieniu wszystkich elementów y ze zbioru D takich, że $d(x, y) \leq k$.

Podstawową miarą odmienności stosowaną w kontekście języków naturalnych jest odległość Levenshteina (edycyjna). Odległość ta zdefiniowana jest jako liczba (być może zachodzących na siebie) operacji edycji potrzebnych do otrzymania słowa y ze słowa x, przy czym możliwe są trzy typy operacji: zamiana znaku a na b dla $a, b \in \Sigma$, usunięcie znaku oraz dodanie znaku. Wszystkie operacje edycji mogą mieć miejsce na dowolnej pozycji słowa. Odległość edycyjną można również zdefiniować jako liczbę *niezachodzących* na siebie operacji edycji (wersja ta znana jest pod nazwą *restricted edit distance*). Obie definicje mogą prowadzić do otrzymania różnych miar odmienności, tutaj przyjmiemy drugą z nich.

Istnieje wiele rozszerzeń odległości Levenshteina, a wśród nich:

- Odległość Damerau-Levenshteina, która rozszerza zbiór operacji edycji o operację zamiany sąsiednich znaków.
- Generalized edit distance – koszty operacji zamiany, dodania i usunięcia mogą zależeć od pozycji znaków biorących udział w operacji jak również od samych wartości tych znaków.
- Extended Levenshtein distance – operacja zamiany zostaje uogólniona do podstawienia dowolnej długości napisu s_1 innym napisem s_2.

Precyzyjny opis pojęcia odległości edycyjnej, niektórych z jej rozszerzeń oraz wyczerpujące porównanie metod wyszukiwania form podobnych można znaleźć w pracy (Boytsov, 2011).

Przygotowane narzędzie implementuje wyszukiwanie form podobnych w drugiej z opisanych wyżej wersji (*restricted generalized edit distance*). Koszt poszczególnych operacji reprezentujemy za pomocą następującej struktury danych, dla uproszczenia zwanej w dalszej części tego rozdziału funkcją kosztu:

```
data Cost a = Cost
    { insert :: (Pos, a) -> Double
    , delete :: (Pos, a) -> Double
    , subst  :: (Pos, a) -> (Pos, a) -> Double }
```

Przy czy przyjmujemy założenie, że:

Założenie 1 *Poszczególne składowe funkcji kosztu (* insert*,* delete *oraz* subst*) są funkcjami nieujemnymi.*

W oparciu o zdefiniowaną wyżej funkcję kosztu można obliczyć odmienność pomiędzy dwoma zadanymi słowami, jak również wyszukać w słowniku formy podobne. Drugie z tych zadań jest rozwiniętą wersją pierwszego, dlatego na początek podamy rekurencyjny wzór na uogólnioną odległość edycyjną. Niech $x.i$ oznacza i-ty znak słowa x, x_i oznacza prefiks x długości i oraz niech $d_{i,j}$ będzie skrótową formą $d(x_i, y_j)$. Wzór na $d_{i,j}$ można przedstawić

w sposób rekurencyjny:

$$d_{0,0} = 0 \tag{1}$$
$$d_{i,0} = d_{i-1,0} + \text{delete } (i,\ x.i) \tag{2}$$
$$d_{0,j} = d_{0,j-1} + \text{insert } (j,\ y.j) \tag{3}$$
$$d_{i,j} = \min \begin{cases} d_{i-1,j-1} & + \quad \text{subst } (i,\ x.i)\ (j,\ y.j) \\ d_{i-1,j} & + \quad \text{delete } (i,\ x.i) \\ d_{i,j-1} & + \quad \text{insert } (j,\ y.j) \end{cases} \tag{4}$$

Przy czym koszt zamiany znaku a na a będzie w ramach sensownej funkcji kosztu wynosił 0. Ostateczna wartość miary odmienności pomiędzy x a y dana jest przez $d_{|x|,|y|}$. Wartość tą można obliczyć w czasie $\mathcal{O}(|x| \cdot |y|)$ wykorzystując np. programowanie dynamiczne.

Wyszukiwanie form podobnych jest zadaniem trudniejszym, ponieważ wymaga znalezienia w zbiorze D wszystkich słów o odmienności od zadanego słowa x mniejszej niż zadany próg. Najprostszym, lecz zbyt wolnym rozwiązaniem jest wyliczenie $d(x, y)$ dla każdego y ze zbioru D i wybranie tych, które są wystarczająco podobne do zadanego słowa x. Lepszym rozwiązaniem jest systematyczne przeglądanie słownika D w poszukiwaniu form podobnych, ktore zakłada że słownik przechowywany jest w drzewie trie. Przechowywanie słownika w drzewie trie samo w sobie jest zaletą, ponieważ pozwala na współdzielenie wspólnych prefiksów poszczególnych elementów słownika i w ten sposób zmiejsza zużycie pamięci potrzebnej do jego przechowywania.

Z każdym wierzchołkiem n drzewa związane jest słowo $y(n)$, które stanowi napis powstały przez sklejenie etykiet na krawędziach ścieżki łączącej korzeń drzewa z wierzchołkiem n. Wartości y dla poszczególnych wierzchołków można generować dynamicznie podczas przeszukiwania drzewa w głąb. Z każdym wierzchołkiem n związana jest również informacja, czy $y(n)$ stanowi element słownika (nie musi tak być, $y(n)$ może być jedynie prefiksem innych słów znajdujących się w słowniku).

Niech $par(n)$ oznacza rodzica wierzchołka n w drzewie trie. Wyszukiwanie form podobnych względem zadanego słowa x sprowadza się do przeszukiwania drzewa trie w głąb przy jednoczesnym wyliczaniu wektora odmienności $v(n) = (v_i(n))_{i \in [1..|x|]}$, gdzie $v_i(n) = d(x_i,\ y(n))$. Wartość $v_i(n)$ oznacza miarę odmienności pomiędzy prefiksem słowa x długości i a aktualnym w kontekście odwiedzanego wierzchołka n słowem $y(n)$. Niech $j = |y(n)|$ oraz *root* oznacza korzeń drzewa trie. Wartości wektora $v(n)$ wylicza się zgodnie ze wzorem 1, który w kontekście przeszukiwania drzewa trie w głąb przekłada się na:

$$v_0(root) = 0 \tag{5}$$
$$v_i(root) = v_{i-1}(root) + \text{delete } (i,\ x.i) \tag{6}$$
$$v_0(n) = v_0(par(n)) + \text{insert } (j,\ y(n).j) \tag{7}$$
$$v_i(n) = \min \begin{cases} v_{i-1}(par(n)) & + \quad \text{subst } (i,\ x.i)\ (j,\ y(n).j) \\ v_{i-1}(n) & + \quad \text{delete } (i,\ x.i) \\ v_i(par(n)) & + \quad \text{insert } (j,\ y(n).j) \end{cases} \tag{8}$$

Twierdzenie 1 *Niech n będzie wierzchołkiem drzewa trie różnym od korzenia. Zachodzi następująca nierówność:*

$$\left(\min_{i \in \{0..|x|\}} v_i(n) \right) \geq \left(\min_{i \in \{0..|x|\}} v_i(par(n)) \right) \tag{9}$$

Nierówność można udowodnić korzystając z założenia 1 oraz ze wzorów 7 i 8

Obliczane w trakcie przeszukiwania drzewa wektory $v(n)$ wykorzystywane są w następujący sposób:

- Jeśli $v_{|x|}(n) \leq k$ oraz $y(n) \in D$, słowo $y(n)$ dodawane jest do wyników wyszukiwania form podobnych (z definicji $v_{|x|}(n) = d(x, y(n))$).
- Gdy $\forall_{i \in \{0..|x|\}} v_i(n) > k$, przeszukiwanie obliczeń w głąb jest przerywane – potomkowie wierzchołka n w drzewie słownikowym nie zostaną odwiedzeni. Na mocy twierdzenia 1, wartość $v_{|x|}(m)$ dla dowolnego potomka m wierzchołka n w drzewie trie byłaby większa niż k.

Opisana wyżej metoda wyszukiwania przybliżonego została zaimplementowana jako moduł języka Haskell. Jego kod można znaleźć w publiczenie dostępnym repozytorium pod adresem `https://github.com/kawu/adict`. Spójność pomiędzy zaimplementowaną metodą a prostszą, opierającą się na sekwencyjnym przeglądaniu całego słownika metodą wyszukiwania przybliżonego została przetestowana przy użyciu biblioteki *QuickCheck*, `http://hackage.haskell.org/package/QuickCheck`.

3.5 Uzupełnianie słownika historycznego

Przygotowana biblioteka wyszukiwania słownikowego może zostać wykorzystana do wypełnienia słownika nowymi formami z tekstów historycznych. Dodawane są wyłącznie nowe formy istniejących już w słowniku leksemów. Metoda opiera się na dwóch zasobach słownikowych:

- Powstający słownik historyczny – źródło form podstawowych poszczególnych leksemów.
- Słownik form współczesnych PoliMorf, `http://zil.ipipan.waw.pl/PoliMorf`. Dla każdego słowa ze słownika PoliMorf zapamiętywana jest dodatkowa informacja o formie podstawowej tego słowa.

Proces uzupełniania słownika wymaga uprzedniej segmentacji tekstu wejściowego na słowa oraz (opcjonalnie) transliteracji. Opisy metod realizujących te podzadania można znaleźć w rozdziałach 3.2 i 3.3.

Przez $D(x)$ będziemy oznaczać najpodobniejsze do x słowo ze słownika D, przy czym miara odmienności liczona jest zgodnie z zadaną funkcją kosztu. Niech H oznacza słownik historyczny, P oznacza słownik PoliMorf, a $lemma(P(x))$ będzie formą podstawową formy przybliżonej $P(x)$. Określone są dwa parametry α i β stanowiące progi dla algorytmu wyszukiwania przybliżonego, przy czym zakładamy że $\alpha < \beta$. Wydobyte z tekstu wejściowego słowo x zostanie dodane do słownika historycznego jeśli zachodzi jeden (lub więcej) z poniższych warunków.

$$d(x, H(x)) \leq \alpha \wedge d(x, P(x)) > \alpha \tag{10}$$

$$d(x, H(x)) \leq d(x, P(x)) \leq \alpha \tag{11}$$

$$d(x, H(x)) \leq \beta \wedge d(x, P(x)) \leq \alpha \wedge lemma(P(x)) = H(x) \tag{12}$$

Pierwszy z powyższych warunków reprezentuje sytuację, gdy w słowniku historycznym znaleziona została bardzo podobna forma podstawowa, natomiast w słowniku form współczesnych nie ma formy (podstawowej lub odmienionej) bardzo podobnej do x. W tym przypadku jest duża szansa, że szukane słowo x jest formą odmienioną znalezionej formy podstawowej (oczywiście z wyjątkiem sytuacji, gdy $d(x, H(x)) = 0$, która świadczy o tym że samo słowo x jest formą podstawową).

Drugi warunek modeluje sytuację, gdy bardzo podobne formy zostały znalezione w obu słownikach, H i P. Wtedy słowo x zostanie dodane do słownika historycznego jeśli forma podstawowa znaleziona w H jest „bardziej” podobna niż ta znaleziona w słowniku P.

Trzeci warunek jest natomiast wyjątkiem od drugiego. Dopuszczamy sytuację, gdy forma podobna została znaleziona w słowniku H ($d(x, H(x)) \leq \beta$) oraz forma bardzo podobna została znaleziona w P ($d(x, P(x)) \leq \alpha$), jeśli forma podstawowa formy z P jest równa formie podstawowej $H(x)$. Dzięki temu warunkowi słownik będzie uzupełniany o nowe formy tych leksemów, których opis znajduje się zarówno w H jak i w P, co może mieć miejsce w przynajmniej dwóch sytuacjach:

- PoliMorf zawiera opis leksemu historycznego, który nie jest już używany we współczesnym języku.
- Leksem opisany w słowniku historycznym zachował się w języku, lecz zmieniło się jego znaczenia (stąd jego opis w słowniku historycznym się znajduje).

Kluczowa dla jakości wyszukiwania przybliżonego jest postać funkcji kosztu. Funkcja ta przypisuje mniejsze wagi operacjom edycji odbywającym się na dalszych pozycjach słów. Ponadto funkcja definiuje grupy znaków (przykładowo „yij", „aą", „eę") w ramach których zamiana niesie za sobą niższy koszt niż zamiana dwóch dowolnych znaków alfabetu. Implementację funkcji kosztu, jak również całej metody uzupełniania słownika opisanej w tym rozdziale, można znaleść w repozytorium pod adresem `svn://chopin.ipipan.waw.pl/synat/tools/gather/GatherSimple.hs`. Metoda uzupełniania słownika została wstępnie przetestowana na danych historycznych zebranych w ramach projektu IMPACT.

4. Obsługa słownika w formacie LMF

Do obsługi słownika LMF wykorzystywane są dwie metody:

- Komunikacja z binarną wersją słownika przechowywaną w bazie BaseX.
- Bezpośrednie parsowanie XML-owego pliku ze słownikiem.

Pierwsza z powyższych metod daje większe możliwości, pozwala np. na uzupełnianie bazodanowej wersji słownika nowymi formami przy użyciu komendy *xquery insert*. Nakładkę na API bazy BaseX stanowi biblioteka *basex* rozwijana pod adresem `svn://chopin.ipipan.waw.pl/synat/haskell-libs/basex`. Kolejna biblioteka o nazwie *basex-lmf* rozwijana jest pod kątem zdalnej obsługi XML-owego pliku zgodnego ze standardem LMF przechowywanego w bazie BaseX. Jej kod można znaleźć pod adresem `svn://chopin.ipipan.waw.pl/synat/haskell-libs/basex-lmf`.

Parsowanie pliku XML jest metodą prostszą i w niektórych zastosowaniach szybszą (w szczególności, gdy narzut na komunikację pomiędzy aplikacją a bazą danych jest istotny). W ramach pracy nad narzędziami do obsługi słownika powstała prosta biblioteka kombinatorów ułatwiających parsowanie plików XML. Biblioteka pozwala na definiowanie parserów XML w sposób deklaratywny, podczas gdy (dzięki leniwej charakterystyce języka Haskell) samo parsowanie odbywa się *online*, co pozwala na wydajne przetwarzanie danych XML o bardzo dużych rozmiarach. Kod biblioteki można znaleźć w repozytorium pod adresem `https://github.com/kawu/polysoup`, natomiast parser słownika LMF znajduje się obecnie pod adresem `svn://chopin.ipipan.waw.pl/synat/tools/gather/Lmf2Trie.hs`.

5. Program LMFEditor (wersja 1.2)

5.1 Przeznaczenie programu LMFEditor

Program przeznaczony jest do edycji tworzonego w ramach projektu SYNAT elektronicznego słownika dawnej polszczyzny. Słownik ten zapisany jest w pliku XML i reprezentowany przez

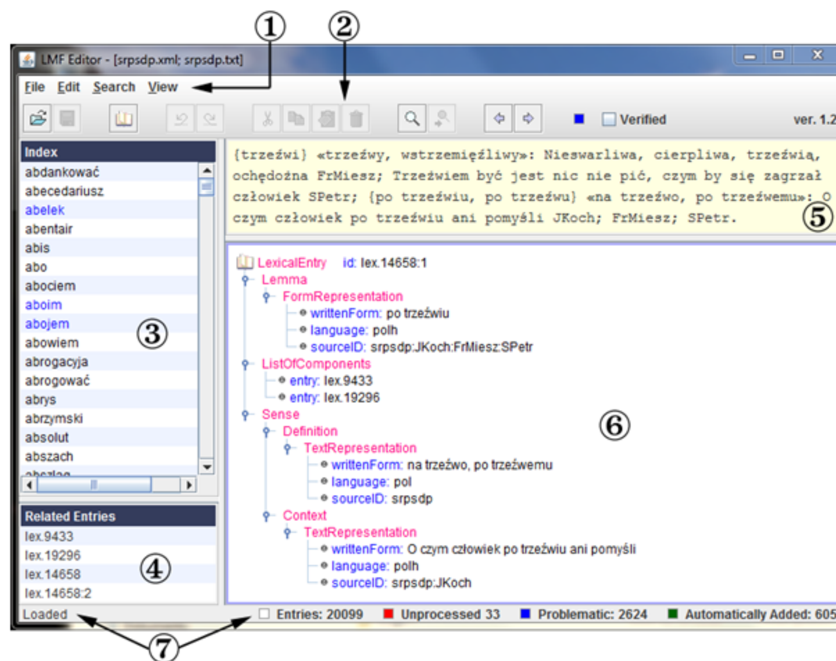
strukturę LMF (Lexical Markup Framework). Program ten nie jest uniwersalnym narzędziem przeznaczonym do edycji dowolnych struktur LMF, lecz tylko tego ich podzbioru, który został wybrany do reprezentowania tworzonego słownika. Danymi wejściowymi programu są dwa pliki: plik XML zawierający strukturę LMF słownika oraz plik tekstowy, na podstawie którego ta struktura powstała. Ten drugi plik jest opcjonalny, ale niewątpliwie ułatwia proces weryfikacji i edycji słownika. Program modyfikuje tylko dane z pliku XML pozostawiając dane z pliku tekstowego w niezmienionej postaci.

Graficzny interfejs użytkownika Program posiada bardzo prosty graficzny interfejs. Posiada on 7 komponentów przedstawionych na rys. 1:

1. Główne menu
2. Pasek narzędzi
3. Indeks haseł słownika
4. Lista identyfikatorów haseł powiązanych z aktualnie wybranym hasłem
5. Tekst źródłowy na podstawie którego powstała struktura LMF hasła
6. Struktura LMF aktualnie wybranego hasła
7. Pasek informacyjny

Wszystkie narzędzia służące edycji słownika znajdują się w głównym menu programu (1). Dla wygody, niektóre z nich zostały umieszczone na pasku narzędzi (2). Indeks (3) jest listą haseł słownika. Hasła w indeksie podzielone są na cztery grupy. Pierwszą grupę stanowią hasła zaznaczone na liście czcionką w czarnym kolorze. Są to hasła proste (najczęściej jednowyrazowe), których opis w pliku tekstowym pasował do konwencji stosowanej w słowniku w wersji papierowej. Niestety, konwencje te łamane były wielokrotnie. Poza tym wiele błędów w wersji tekstowej zostało wprowadzonych przez program OCR. Stąd pojawiła się grupa haseł (unprocessed – czcionka czerwona), których automatyczne przetworzenie nie było możliwe oraz takie (problematic – czcionka niebieska), co do których istnieje obawa, że zostały błędnie przetworzone do postaci LMF. Istnieje ponadto jeszcze jedna grupa haseł (automatically added – czcionka zielona) zawierająca hasła będące częściami składowymi haseł złożonych, a które nie występują w słowniku, albo występują w innej, odmienionej formie. Hasła z indeksu wybieramy przy pomocy myszki lub klawiatury (klawisze ˆˇ). Wybranie hasła spowoduje wyświetlenie struktury LMF hasła (6), tekstu źródłowego (5), na podstawie którego ta struktura powstała oraz listy identyfikatorów (4) struktur związanych z aktualnie wybranym hasłem. Program umożliwia usuwanie haseł z indeksu przy pomocy polecenia Delete z menu Edit. W polu wyświetlającym tekst źródłowy (5) użytkownik może zaznaczyć interesujący go fragment tekstu, skopiować go i następnie wkleić w odpowiednie miejsce w strukturze LMF. Struktura LMF (6) jest strukturą drzewiastą. Nazwy poszczególnych węzłów drzewa są wyświetlane w kolorze ciemnoróżowym. Nazwy liści nie są wyświetlane. Liśćmi drzewa mogą być jedynie dwie podstruktury LMF: <feat> oraz <Component>. W przypadku tylko tych podstruktur, w celu uzyskania większej przejrzystości, wyświetlane są tylko ich atrybuty. Nazwy atrybutów zaznaczone są czcionką niebieską, zaś ich wartości, czcionką czarną. Pasek informacyjny (7) zawiera różnego rodzaju komunikaty oraz statystykę dotyczącą liczby haseł w różnych ich grupach.

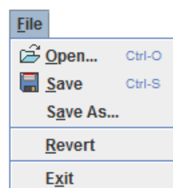
Strukturę LMF (6) można edytować albo przy pomocy menu Edit, albo przy pomocy menu kontekstowego wyświetlanego po naciśnięciu prawego klawisza myszy w momencie, gdy kursor wskazuje na któryś z węzłów drzewa. W tym pierwszym przypadku należy najpierw uaktywnić polecenia z menu Edit poprzez wybranie struktury, którą chcemy edytować. Strukturę można wybrać poprzez kliknięcie lewym klawiszem myszy lub przy pomocy klawiatury (klawisze ˆˇ). Możemy wtedy usunąć wskazywaną strukturę, dodać do niej inną strukturę, dodać lub usunąć atrybut. Zarówno nazwa jak i wartość atrybutu też mogą być edytowane. W tym celu należy dwukrotnie kliknąć na nazwę lub wartość atrybutu. W przy-



Rysunek 2: Główne okno edytora

padku nazwy atrybutu pojawi się wówczas lista zawierająca wszystkie dozwolone nazwy, z której należy wybrać jedną. W przypadku, gdy na liście nie ma nazwy, którą użytkownik chciałby wybrać, program umożliwia utworzenie nowej. W przypadku wartości atrybutów mogą zachodzić dwa przypadki: wartość jest dowolnym tekstem, lub elementem określonego zbioru wartości. W pierwszym przypadku, pojawi się edytowalne pole tekstowe. W drugim przypadku, podobnie jak dla nazw atrybutów, pojawi się lista możliwych wartości do wyboru, którą także można uzupełnić o wartość, która na niej nie występuje. Przy edytowaniu nazw i wartości atrybutów dostępne są standardowe polecenia edytujące: Cut, Copy , Pase, Delete, Undo i Redo.

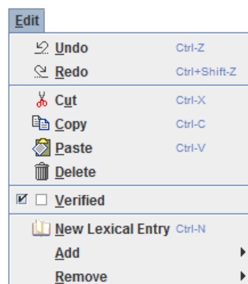
5.2 Menu programu



Rysunek 3: Menu **File**

Po pierwszym uruchomieniu programu należy wczytać do pamięci strukturę słownika. Służy temu polecenie Open. Po wybraniu tego polecenia z menu wyświetli się standardowe okno dialogowe, w którym należy wybrać odpowiedni plik o rozszerzeniu .xml. Następnie, po raz kolejny program poleci wybrać z listy plików jeden element (zazwyczaj o tej samej nazwie, ale z rozszerzeniem .txt) zawierający tekst, na podstawie którego utworzono strukturę LMF słownika. Wybór tego pliku jest opcjonalny, ale ułatwia on poprawianie struktury LMF zawartej w pliku .xml. Przy następnym wywołaniu programu, oba wybrane pliki są wczytywane automatycznie. Polecenia Save i Save As. . . służą zapisaniu na dysku wprowadzonych poprawek. Polecenie Revert służy otworzeniu danych do postaci w jakiej one były po ostatnim wykonaniu polecenia Save. Pracę programu można zakończyć zamykając jego główne okno lub wybierając z menu polecenie Exit. Zanim program zakończy pracę,

poprosi użytkownika aby zapisał wprowadzone poprawki, o ile takie poprawki zostały dokonane. Następnie zapamięta na dysku różne ustawienia programu: ścieżki plików wejściowych, parametry wyszukiwania oraz parametry sortowania i wyświetlania indeksu słownika. Informacje te zapisane są w prywatnym katalogu użytkownika w pliku LMFEditorPrefs.



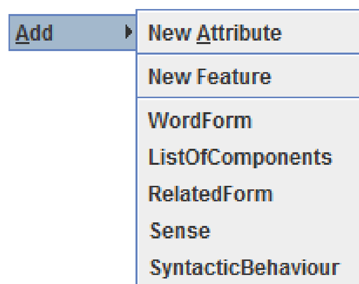
Rysunek 4: Menu **Edit**

Polecenie Undo (cofnij modyfikację) i Redo (zastosuj ponownie) odpowiednio przywraca stan sprzed ostatniej modyfikacji i dokonuje ponownej modyfikacji po tym jak użytkownik wykonał polecenie Undo. Program pamięta 20 ostatnio wykonywanych modyfikacji.

Standardowe polecenia Cut (wytnij), Copy (kopiuj), Paste (wklej) oraz Delete (usuń) są aktywne lub nieaktywne w zależności od tego, który z elementów interfejsu jest w danej chwili aktywny.

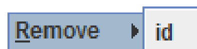
Poleceniem Verified zmieniamy status danego hasła. Jeśli hasło należało do jednej z grup: *unprocessed*, *problematic* lub *automatically added*, to wykonaniu tego polecenia staje się zweryfikowanym. W przeciwnym przypadku hasło jest zakwalifikowane do grupy haseł *problematic*.

Nowe hasło główne wstawiamy do słownika przy pomocy polecenia New Lexical Entry. Nowa struktura otrzyma automatycznie unikalny identyfikator.



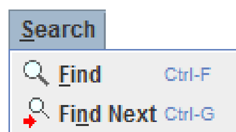
Rysunek 5: Menu **Add**

Menu Add jest tworzone dynamicznie i zależy o tego jaka podstruktura LMF jest w danej chwili wybrana. Powyższy przykład dotyczy przypadku, w którym wybrana została struktura odpowiadająca całemu hasłu. Można więc dodać nowy atrybut (New Attribute) dla tej struktury oraz nową własność (New Feature). Można także dodać podstruktury (<WordForm>, <ListOfComponents>, <RelatedForm>, <Sense> i <SyntacticBehaviour>). W menu umieszczone są tylko nazwy tych podstruktur, które mogą być dodane do aktualnie wybranej struktury.



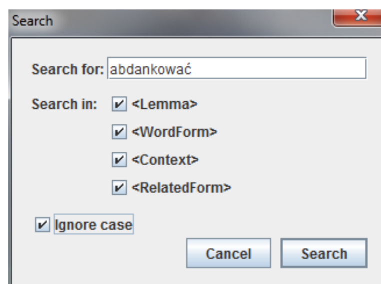
Rysunek 6: Menu **Remove**

Podobnie jak w przypadku menu *Add*, menu *Remove* jest tworzone dynamicznie. Na liście tego menu znajdują się atrybuty aktualnie wybranej struktury, które możemy usunąć.



Rysunek 7: Menu **Search**

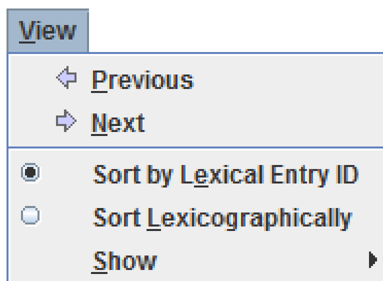
Po wybraniu z menu polecenia Find program wyświetla dialog (rysunek 8). Dialog ten pozwala użytkownikowi na ustalenie pewnych parametrów wyszukiwania tekstu w słowniku. Szukany tekst jest reprezentowany przez wyrażenie regularne, które należy wpisać w polu Search for dialogu. Użytkownik może wyspecyfikować do których fragmentów opisów haseł słownika należy ograniczyć wyszukiwanie. Może być to forma podstawowa (<Lemma>), zbiór form wyrazowych (<WordForm>), zbiór cytatów z tekstów źródłowych (<Context>) lub zbiór form pokrewnych (<RelatedForm>). Wyszukiwanie może brać pod uwagę lub ignorować wielkość liter w tekście. Wyszukiwanie zawsze rozpoczyna się od pierwszego hasła w słowniku niezależnie od wybranej metody sortowania. Program wyświetla pierwszą strukturę zawierającą szukane wyrażenie lub informuje użytkownika o niepowodzeniu przy pomocy sygnału dźwiękowego. W wyświetlanych strukturach miejsca odpowiadające szukanemu wyrażeniu są zaznaczone kontrastowym żółtym kolorem.



Rysunek 8: Dialog wyszukiwania tekstu

Polecenie Find Next staje się dostępne w przypadku powodzenia polecenia Find. Słownik przeszukiwany jest wtedy od ostatnio znalezionej struktury aż do końca słownika, a potem z powrotem od jego początku.

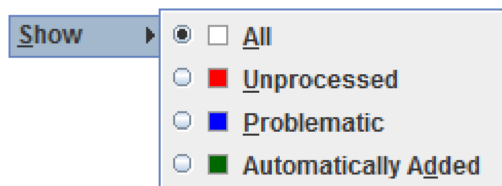
Polecenia Previous i Next służą do nawigowania po ostatnio wyświetlanych strukturach. Działają one podobnie jak w przypadku przeglądarek internetowych, które umożliwiają szybki powrót do poprzednio odwiedzanych stron. Program pamięta 20 ostatnio wyświetlanych struktur.



Rysunek 9: Menu **View**

Hasła słownika mogą być sortowane na dwa sposoby: po identyfikatorze hasła (Sort by

Lexical Entry ID) lub alfabetycznie (Sort Lexicographically). W przypadku pierwszej metody sortowania hasło główne i jego podhasła będą się znajdować obok siebie na liście indeksu.



Rysunek 10: Menu **Show**

Wyświetlany indeks haseł można zawęzić tylko do jednej ze wspomnianych na wstępie grup. Domyślnie program wyświetla pełen indeks.

5.3 Wymagania sprzętowe

Program napisany jest w języku Java i pracuje pod dowolnym systemem, na którym zainstalowana jest Java w wersji 1.6 lub wyższej. Program potrzebuje do pracy ok. 500MB pamięci RAM.

6. Literatura

1. S. Bąk, M. R. Mayenowa, and F. Pepłowski, editors. 1966-2002. Słownik polszczyzny XVI wieku, vol. I-XXX. Zakład Narodowy Imienia Ossolińskich. Wydawnictwo PAN.
2. L. Boytsov, Indexing methods for approximate dictionary searching: Comparative analysis. *J. Exp. Algorithmics*, 16:1, May 2011.
3. F. de Vriend, L. Boves, H. van den Heuvel, and R. van Hout aand J. Swanenberg. 2006. A unified structure for dutch dialect dictionary data. In Proceedings of The fifth international conference on Language Resources and Evaluation, LREC, Genoa, Italy.
4. G. Francopoulo, N. Bel, M. George, M. Calzolari, M. Pet, and C. Soria. 2009. Multilingual resources for NLP in the lexical markup framework (LMF). Language Resources and Evaluation, 43:57–70.
5. A. Mykowiecka, K. Głowińska, P. Rychlik, and J. Waszczuk. 2011. A construction of an electronic dictionary on the base of a paper source. In Proceedings of Lnaguage and Technology Conference, Poznań.
6. A. Mykowiecka, P. Rychlik, and J. Waszczuk. 2012. Building an electronic dictionary of Old Polish on the base of the paper resource. In Proceedings of Cultural Heritage LREC 2012 Workshop.
7. Instytut Języka Polskiego PAN. 1996-. Słownik języka polskiego XVII i 1. połowy XVIII wieku. http://sxvii.pl. S. Reczek. 1968. Podręczny słownik dawnej polszczyzny. Ossolineum.
8. L. Romary, S. Salmon-Alt, and G. Francopoulo. 2004. Standards going concrete: from LMF to Morphalou. In Proceedings of the 20th International Conference on Computational Linguistics, COLING 2004, Geneve.
9. Z. Saloni, W. Gruszczyński, M. Woliński, and R. Wołosz. 2007. Słownik gramatyczny języka polskiego. Wiedza Powszechna.

Spis treści

1 Pierwszy etap budowy słownika **2**

1.1 Ogólna charakterystyka zadania budowy słownika 2

1.2 Przetwarzanie wstępne tradycyjnego słownika 2

1.3 Ustalenie struktury słownika 2

1.4 Struktura słownika 3

1.5 Reguły translacji 4

2 Uzupełnianie słownika o oznaczenia części mowy **7**

3 Uzupełnianie słownika o nowe formy **8**

3.1 Wstęp 8

3.2 Segmentacja na słowa 8

3.3 Transliteracja 9

3.4 Wyszukiwanie form podobnych 10

3.5 Uzupełnianie słownika historycznego 12

4 Obsługa słownika w formacie LMF **13**

5 Program LMFEditor (wersja 1.2) **13**

5.1 Przeznaczenie programu LMFEditor 13

5.2 Menu programu 15

5.3 Wymagania sprzętowe 18

6 Literatura **18**

Spis treści **19**