

INSTITUTE OF COMPUTER SCIENCE, POLISH ACADEMY OF SCIENCES

# tokenizer

---

Podręcznik użytkownika

Piotr Rychlik

## Wprowadzenie

Proces tokenizacji przebiega w kilku etapach:

1. Przetwarzanie wstępne (opcjonalne) polegające na znormalizowaniu znaków UTF-8 występujących w tekście. Jest to etap opcjonalny, zakłada się wówczas, że przetwarzane teksty były już w ten sposób przygotowane.
2. Podział tekstu na paragrafy i zdania. Ten etap realizowany będzie przez istniejący już program do segmentacji tekstów wykorzystujący reguły SRX.
3. Wstępna tokenizacja. Na tym etapie tekst zostanie wstępnie podzielony na tokeny. Zostanie utworzona lista składająca się z tokenów oraz rozdzielających je separatorów. Lista ta będzie w dalszej części procesu tokenizacji modyfikowana przy pomocy systemu regułowego.
4. Identyfikacja skrótów, nazw własnych i wyrażen obcojęzycznych. Ta faza jest planowana w przyszłości
5. Zapisanie wyników tokenizacji w pliku XML lub tekstowym.

Program *tokenizer* ma dwa parametry: *--config* oraz *--input*. Argumentem opcji *--config* jest ścieżka do pliku konfiguracyjnego, zaś opcji *--input* – ścieżka do pliku tekstowego lub katalogu z plikami tekstowymi do tokenizacji.

Program napisany jest w języku Java i wymaga uprzedniej instalacji Javy w wersji 1.7 lub późniejszej. Program wykorzystuje również natywne biblioteki wykorzystywane przez analizator morfologiczny Morfeusz (<http://sgjp.pl/morfeusz/dopobrania.html>).

## Plik konfiguracyjny

Plik konfiguracyjny składa się z listy poleceń. Zadaniem każdego polecenia jest dookreślenie kolejnych etapów tokenizacji oraz wykorzystywanych w tym procesie zasobów, jak również zdefiniowanie reguł klasyfikacji, dzielenia i scalania tokenów.

W pliku konfiguracyjnym, oprócz reguł, mogą się znaleźć następujące polecenia:

- *clean-text* { *yes* | *no* },
- *preliminary-tokenization* { <nazwa metody tokenizacji> },
- *segment* { <parametry wbudowanego systemu segmentującego> },
- *morph* { <parametry analizatora Morfeusz> },
- *output* { <format pliku wynikowego> },
- *abbreviations* { <ścieżka do słownika ze skrótami> },
- *names* { <ścieżka do słownika z nazwami własnymi> },
- *foreign* { <ścieżka do słownika z wyrażeniami obcymi> }.

Wszystkie polecenia są opcjonalne.

Brak polecenia *clean-text* jest równoznaczne z pominięciem etapu wstępnego przygotowania tekstu do dalszej obróbki, o którym była już mowa powyżej.

Polecenie *preliminary-tokenization* służy do wybrania stosownej metody wstępnego podziału tekstu na tokeny. Będą dostępne następujące procedury do wstępnej tokenizacji:

- *whitespaceTokenizer* – dzielący tekst na tokeny w miejscach występowania znaków oznaczających odstęp,
- *letterTokenizer* – dzielący tekst w miejscu występowania znaków nie będących literami,
- *patternTokenizer* – dzielący tekst w miejscach zdefiniowanych przez wyrażenie regularne,
- *standardTokenizer* – dzielący tekst na tokeny w miejscach występowania znaków odstępu lub znaków interpunkcyjnych. Wyjątkami są znaki, które często służą do łączenia części tekstu, np. ‘-’ i ‘\_’.
- *doNothingTokenizer* – agregujący cały tekst jednego zdania. Podział tego tekstu na tokeny zależy będzie tylko od zestawu reguł.

Domyślną metodą wstępnej tokenizacji jest *standardTokenizer*.

Formatem pliku wynikowego może być format xml lub format tekstowy. Formatem domyślnym jest format xml. Oba formaty zostaną omówione w dalszej części.

Poleceniem *morph* możemy określić następujące parametry analizatora morfologicznego Morfeusz:

- *dict* { <ścieżka do słownika> } (ścieżka powinna być ujęta w cudzysłów);
- *aggl* { <reguła aglutynacyjna> } (*isolated* / *permissive* / *strict* – wartością domyślną jest wartość *strict*);
- *praet* { <segmentacja czasu przeszłego> } (*composite* / *split* – wartością domyślną jest wartość *split*);
- *case-handling* { <interpretacja wielkości liter> } (*CONDITIONALLY\_CASE\_SENSITIVE* / *STRICTLY\_CASE\_SENSITIVE* / *IGNORE\_CASE* – wartością domyślną jest *CONDITIONALLY\_CASE\_SENSITIVE*).

W przypadku gdy nie określimy żadnych parametrów wykorzystywanych przez analizator morfologiczny Morfeusz, program użyje domyślnych parametrów i wbudowany słownik *sgjp*.

Jeśli nie chcemy, aby analiza morfologiczna znalazła się w pliku wynikowym, w pliku konfiguracyjnym należy umieścić polecenie *morph { ignore }*.

Poleceniem *segment* możemy określić zestaw reguł SRX służących podziałowi tekstu na zdania oraz metodę podziału tekstu na paragrafy.

- *srx* { <ścieżka do pliku z zestawem reguł SRX> } (ścieżka powinna być ujęta w cudzysłów);
- *par* { <jedna z wartości *LINE\_BREAK* lub *EMPTY\_LINE*> } (wartością domyślną jest *EMPTY\_LINE*).

Brak polecenia *segment* jest jednoznaczne z wykorzystaniem standardowego zestawu reguł SRX dołączonego do programu tokenizera oraz identyfikacją kolejnych paragrafów w tekście przez puste linie.

Etap identyfikacji skrótów, nazw własnych oraz wyrażen obcych może być wspomagany różnymi źródłami danych. Do identyfikacji tych danych służą odpowiednio polecenia *abbreviations*, *names* i *foreign*.

W pliku konfiguracyjnym znajdować się może lista reguł, których zadaniem jest analiza wyników wstępnej tokenizacji i finalny podział tekstu na tokeny. W przypadku braku takiej listy, wstępna tokenizacja jest zarazem finalną.

## Reguły

Reguły są wykorzystywane w dalszym procesie przetwarzania tokenów zidentyfikowanych podczas wstępnej tokenizacji. Proces ten zaczyna się od początkowej klasyfikacji tokenów i ich podziału na typy np. słowa, liczby, adresy URL, itp. Po tym etapie następuje rekurencyjny podział tokenów lub ich łączenie oraz ponowna klasyfikacja. Proces ten kontynuowany jest aż do momentu, w którym żadna reguła (klasyfikująca, dzieląca lub scalająca) nie będzie mogła być zastosowana.

Wynikiem wstępnej tokenizacji jest lista (LST) składająca się z tokenów i rozdzielających je separatorów. Przed przeprowadzeniem klasyfikacji tokenów, elementom tej listy nadaje się typy początkowe – *TOK* dla tokenów i *SEP* dla separatorów. Lista LST jest modyfikowana przez system regułowy.

Każda reguła składa się z dwóch części. W pierwszej z nich określony jest szablon, który system będzie się starał dopasować do jakiegoś fragmentu listy LST. Druga część składa się z deklaracji tokenów, które powinny zastąpić dopasowany wcześniej fragment.

Regułę zapisujemy następująco:

```
rule {
  cond { <template> : <bool-expression> }
  token { <type> [<token-def>] }
  ...
  token { <type> [<token-def>] }
}
```

### Przykład 1.

Weźmy pod uwagę prostą regułę klasyfikującą:

```
rule {
  cond { t : t.matches("\d+") }
  token { number }
}
```

Zastosowanie tej reguły powoduje, że każdemu tokenowi składającemu się z samych cyfr zostanie nadany typ „number”.

### Przykład 2.

Poniższa reguła jest przykładem reguły scalającej. Szablon  $t1(t2)^*$  dopasowuje się do ciągu liczb, z których pierwsza jest liczbą co najwyżej 3 cyfrową, zaś pozostałe posiadają dokładnie 3 cyfry. Dopasowany ciąg liczb jest przez tę regułę scalony w jeden token typu „number”.

```
rule {
  cond { t1(t2)* : t1.is("number") and t1.matches("\d{1,3}") and
                  t2.is("number") and t2.matches("\d{3}")
  }
  token { number }
}
```

### Przykład 3.

```
rule {
  cond { t : t.is("alphanumeric") and t.matches("(\\d+)(\\p{L}+)") }
  token { number t.group(1) }
  token { alpha t.group(2) }
}
```

Zastosowanie tej reguły spowoduje podział tokenu, którego początkowa część składa się z samych cyfr, zaś reszta znaków to litery, na dwa tokeny o typach odpowiednio „number” i „alpha”.

### Przykład 4.

```
rule {
  cond { s(t+)s : s.equals("\"") }
  token { citation group(1) }
}
```

Powyższa reguła zcała grupę tokenów ujętych w cudzysłów w jeden token typu „citation”.

Szablon reguły składa się z nazw tokenów, do których będziemy odwoływać się przy formułowaniu warunków jakie dopasowywane tokeny mają spełniać. Wyrażenia stanowiące szablon konstruowane są w podobny sposób co wyrażenia regularne. W skład szablonu wchodzi więc nazwy tokenów i mogą wchodzić takie znaki jak ‘\*’, ‘+’, ‘?’ a także nawiasy ‘( i ’)’, które oznaczają dokładnie to samo, co w przypadku wyrażeń regularnych.

Warunkiem dopasowania szablonu do listy LST jest spełnienie warunku zdefiniowanego po znaku ‘:’. Warunkiem tym jest wyrażenie logiczne. W wyrażeniach takich występują termy określające warunki jakie spełniać mają poszczególne tokeny oraz spójniki logiczne: and, or i not. Wyrażenia można grupować w podwyrażenia stosując nawiasy ‘( i ’)’. Każdy term składa się z nazwy tokenu, po której następują kolejno: znak ‘:’, opcjonalnie nazwa selektora zakończona znakiem ‘.’ i nazwa funkcji testującej wraz z ewentualnymi parametrami. W obecnej wersji programu są dwa selektory: `str` oznaczający ciąg znaków zawartych w tokenie oraz `type` oznaczający typ tokenu.

Poniższa tabela zawiera wykaz podstawowych funkcji testujących.

Funkcja	Selektor	Parametry	Opis
is	type	String	Bada czy token jest określonego typu.
matches	str	String	Bada czy token dopasowuje się do podanego wyrażenia regularnego. Funkcja zapamiętuje wynik dopasowania. Dla jednego tokenu może być tylko jedno wywołanie tej funkcji.
agreesWith	dowolny	String	Działa podobnie jak matches, ale nie zapamiętuje wyniku dopasowania. Można ją stosować wielokrotnie dla jednego tokenu.
equals	dowolny	String	Sprawdza czy wartość selektora jest danym ciągiem znaków.
startsWith	dowolny	String	Sprawdza czy wartość selektora zaczyna się podanym ciągiem znaków.
endsWith	dowolny	String	Sprawdza czy wartość selektora kończy się podanym ciągiem znaków.
contains	dowolny	String	Sprawdza czy wartość selektora zawiera podany ciąg znaków.

Gdy system dopasuje szablon reguły do listy LST, następuje zastąpienie dopasowanych tokenów tymi, które są zdefiniowane w klauzulach `token { ... }`. Definiując nowe tokeny musimy nadać im typ oraz określić reprezentujące je ciągi znaków. Łącząc tokeny w jeden musimy podać numer grupy w wyrażeniu regularnym definiującym szablon. Dzieląc tokeny na mniejsze składowe musimy podać numer grupy w wyrażeniu regularnym podanym jako parametr funkcji `match`. Podanie jedynie typu nowego tokenu oznacza zastąpienie całego wyniku dopasowania szablonu (`group(0)`) jednym tokenem o tym typie. W przypadku, gdy podział tokenu nie dotyczył wszystkich jego znaków, z fragmentów nieprzyporządkowanych żadnemu tokenowi tworzy się tokeny o typie TOK.

## Formaty plików

Tokenizer przewiduje dwa rodzaje plików wynikowych – XML i TXT.

Głównym elementem pliku XML jest element `<document>` składający się z elementów `<p>` reprezentujących zidentyfikowane w tekście paragrafy. Paragrafy składają się z elementów `<s>` oznaczających poszczególne zdania, te z kolei zawierają elementy `<tok>` odpowiadające tokenom występującym w zdaniu. Elementy `<p>`, `<s>` oraz `<tok>` są identyfikowane przez parametr `id`. Typ tokenu określony jest przez parametr `type` elementu `<tok>`. Element `<tok>` składa się z dwóch części – ciągu znaków zawartych w elemencie `<string>` i identyfikujących token w tekście oraz (opcjonalnie) z morfologicznej analizy. Na opis analizy morfologicznej tokenu składają się elementy `<interp>` oraz `<alt>`. W skład elementu `<interp>` wchodzi element `<item>` odpowiadające krawędziom grafu generowanego przez analizator Morfeusz. Element `<item>` zawiera formę ortograficzną `<orth>` oraz listę alternatywnych opisów `<lex>` określających formę podstawową `<base>` oraz zestaw znaczników morfosyntaktycznych `<tag>`. Elementy `<alt>` odpowiadają rozgałęzieniom występującym w grafie generowanym przez analizator Morfeusz. Składają się one z przynajmniej dwóch elementów `<interp>`.

Na ogół każdemu tokenowi odpowiada dokładnie jeden element `<interp>` lub `<alt>`. W przypadku tokenów złożonych, których poszczególne części oddzielone są separatorem (np. spacją) jest to lista elementów `<interp>` i/lub `<alt>`. W odróżnieniu od tych fragmentów tekstu, które odpowiadają kolejnym elementom `<item>`, fragmenty tekstu odpowiadające kolejnym elementom `<interp>` lub `<alt>` są oddzielone od siebie separatorem.

W przypadku, gdy dwa kolejne tokeny nie są rozdzielone w tekście żadnym separatorem, to pomiędzy kolejne elementy `<tok>`, które odpowiadają tym tokenem umieszcza się element `<ns />`.

Zawartość pliku XML odpowiadająca pojedynczemu zdaniu „Gdzieś był?” wygląda w następujący sposób:

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <p id="1">
    <s id="1">
      <tok id="1" type="TOK">
        <string>Gdzieś</string>
        <alt>
          <interp>
            <item>
              <orth>Gdzie</orth>
              <lex><base>gdzie:d</base><tag>adv</tag></lex>
              <lex><base>gdzie:q</base><tag>qub</tag></lex>
            </item>
          </interp>
        </alt>
      </tok>
    </s>
  </p>
</document>
```

```

        <item>
            <orth>ś</orth>
            <lex><base>być</base>
                <tag>aglt:sg:sec:imperf:nwok</tag></lex>
        </item>
    </interp>
    <interp>
        <item>
            <orth>Gdzieś</orth>
            <lex><base>gdzieś:d</base><tag>adv</tag></lex>
            <lex><base>gdzieś:q</base><tag>qub</tag></lex>
        </item>
    </interp>
</alt>
</tok>
<tok id="2" type="TOK">
    <string>był</string>
    <interp>
        <item>
            <orth>był</orth>
            <lex><base>być</base>
                <tag>praet:sg:m1.m2.m3:imperf</tag></lex>
        </item>
    </interp>
</tok>
<ns />
<tok id="3" type="TOK">
    <string>.</string>
    <interp>
        <item>
            <orth>.</orth>
            <lex><base>.</base><tag>interp</tag></lex>
        </item>
    </interp>
</tok>
</s>
</p>
</document>

```

Innym formatem pliku wynikowego jest format tekstowy. Zawartść pliku tekstowego odpowiadającego plikowi XML opisanemu powyżej jest następująca:

```

PST→Gdzieś→TOK
[
IM→Gdzie
L→gdzie:d→adv
L→gdzie:q→qub
M→ś
L→być→aglt:sg:sec:imperf:nwok
IM→Gdzieś
L→gdzieś:d→adv
L→gdzieś:q→qub
]
T→był→TOK→n

```

IM→był  
 L→być→praet:sg:m1.m2.m3:imperf  
 T→?→TOK  
 IM→?  
 L→?→interp

Pierwsza kolumna składa się z jednej lub kilku znaków i określa jaką informację zawierać będzie dana linia lub następujące po niej linie tekstu:

- P – początek paragrafu,
- S – początek zdania,
- T – kolejny token,
- I – kolejna interpretacja,
- M – kolejny leksem,
- L – opis leksemu,
- [ – początek listy alternatywnych interpretacji,
- ] – koniec listy alternatywnych interpretacji.

Poszczególne kolumny oddzielone są znakiem tabulacji. Ich znaczenie opisane jest poniższą tabelą.

Kolumna 1	Kolumna 2	Kolumna 3	Kolumna 4
P	nie dotyczy	nie dotyczy	nie dotyczy
S	nie dotyczy	nie dotyczy	nie dotyczy
T	ciąg znaków reprezentujących token	typ	znak 'n', jeśli przed następnym tokenem w zdaniu nie było separatora
I	nie dotyczy	nie dotyczy	nie dotyczy
M	forma ortograficzna	nie dotyczy	nie dotyczy
L	forma podstawowa	tag	nie dotyczy
[	nie dotyczy	nie dotyczy	nie dotyczy
]	nie dotyczy	nie dotyczy	nie dotyczy

## Składnia BNF języka definiującego konfigurację tokenizera

```

<tokenizer> ::= ε | <list-of-commands>

<list-of-commands> ::= <command> | <command><list-of-commands>

<command> ::= <clean-text> | <pre-tokenizer> |
               <output> | <srx> | <abbreviations> |
               <names> | <foreign> | <rule>

<clean-text> ::= "clean-text" "{" "yes" | "no" "}"

<pre-tokenizer> ::= "preliminary-tokenization" "{" <method-call> "}"

<output> ::= "output" "{" "xml" | "txt" "}"

<segment> ::= "segment" "{" <seg-options> "}"

<seg-options> ::= <seg-opt><seg-options>

<seg-opt> ::= <srx> | <par>
  
```



```

<srx> ::= "srx" "{" <path> "}"
<par> ::= "par" "{" <par-opt> "}"
<par-opt> ::= "EMPTY_LINE" | "LINE_BREAK"
<morph> ::= "morph" "{" <morph-options> "}"
<morph-options> ::= <morph-opt><morph-options>
<morph-opt> ::= <morph-dict> | <aggl> | <praet> | <case-handling>
<morph-dict> ::= "dict" "{" <path> "}"
<aggl> ::= "aggl" "{" <aggl-option> "}"
<aggl-option> ::= "isolated" | "permissive" | "strict"
<praet> ::= "praet" "{" <praet-option> "}"
<praet-option> ::= "composite" | "split"
<case-handling> ::= "case-handling" "{" <case-handling-option> "}"
<case-handling-option> ::= "CONDITIONALLY_CASE_SENSITIVE" |
                             "STRICTLY_CASE_SENSITIVE" |
                             "IGNORE_CASE"
<abbreviations> ::= "abbreviations" "{" <path> "}"
<names> ::= "names" "{" <path> "}"
<foreign> ::= "foreign" "{" <path> "}"
<rule> ::= "rule" "{" <cond><list-of-tokens> "}"
<cond> ::= "cond" "{" <selector> ":" <bool-expression> "}"
<list-of-tokens> ::= "{" <token> | <token><list-of-tokens> "}"
<cond> ::= "cond" "{" <template> ":" <bool-expression> "}"
<template> ::= <basic-template> | <basic-template><template>
<basic-template> ::= <elementary-template> |
                     <elementary-template> "*" |
                     <elementary-template> "+" |
                     <elementary-template> "?"
<elementary-template> ::= <identifier> | <group>
<group> ::= "(" <template> ")"
<bool-expression> ::= <bool-term> | <bool-term><or-expression>
<bool-term> ::= <not-expression> | <not-expression><and-expression>
<or-expression> ::= "or" <bool-term> |
                    "or" <bool-term><or-expression>
<and-expression> ::= "and" <not-expression> |

```

```

    "and" <not-expression><and-expression>

<not-expression> ::= <bool-factor> | "not" <bool-factor>

<bool-factor> ::= <token-factor> | "(" <bool-expression> ")"

<token-factor> ::= <identifier> "." <method-call> |
    <identifier> "." <identifier> "." <method-call>

<token> ::= "token" "{" <token-type> | <token-type><token-def> "}"

<token-type> ::= <identifier>

<token-def> ::= "group" |
    "group" "(" <integer> ")" |
    <identifier> "." "group" |
    <token-id> "." "group" "(" <integer> ")"

<method-call> ::= <method-name> | <method-name><args>

<method-name> ::= <identifier>

<args> ::= "(" <argument-list> ")"

<argument-list> ::= <argument> | <argument> "," <argument-list>

```