Wstępna wersja języka zapytań w nowym Poliqarpie

Dominika Pawlik, Aleksander Zabłocki
MIM UW, IPI PAN
Bartosz Zaborowski
IPI PAN

21 października 2013

Plan

1. Co chcemy zrobić
2. Model danych
3. Przegląd języka

Plan

1. Co chcemy zrobić
   - Zarys możliwości
   - Trochę szczegółów
2. Model danych
3. Przegląd języka

Wejście

Poliqarp 1.0 & 2.0	Poliqarp 2.0

• KIPI (NKJP)
  • morfoskładnia
  • segmentacja
  • metadane
	• całe NKJP
	  • sł. składniowe, jedn. nazewn.
	  • sensy słów
	• inne projekty IPI
	  • Składnica: grafy składnikowe
	  • grafy zależnościowe
	  • LFG
	• możliwe rozszerzenia
	  • korpusy historyczne
	    • atrybuty czcionki
	    • jakość OCR, transkrypcje
	    • atrybuty pozycyjne

Funkcjonalność

(klamerki = świat wyszukiwarek drzewiastych)

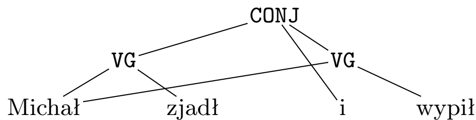
	Poliqarp 1.0 & 2.0	Poliqarp 2.0	
	• wieloznaczność • interpretacji	• grafu składnikowego • segmentacji • transkrypcji • ...	

nietypowe	• wyrażenia regularne
	  • dla znaków
	  • dla segmentów
		• relacje „drzewiaste”
		  • ojciec, przodek
		  • lewy brat, lewy syn
			standard

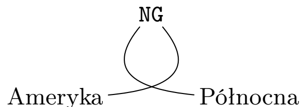
	• zmienne • lukier składniowy dla typów skończonych	• kwantyfikacja • negacja	ambitne

Grafy składniowe

- współdzielone węzły
- nieciągłość poddrzew

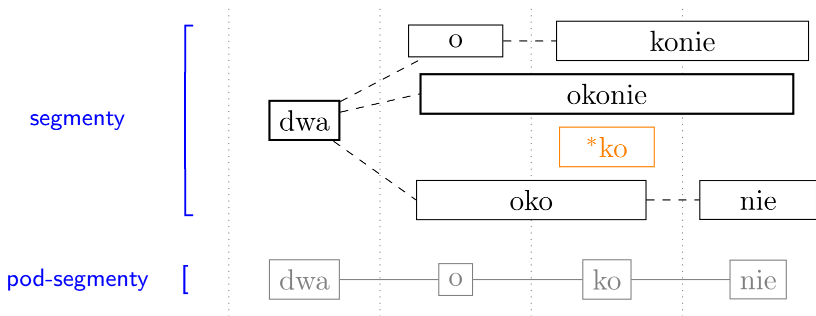


- zaburzenia kolejności:



  - porządek segmentowy — dla wszystkich segmentów
  - porządki krawędziowe — dla dzieci dowolnego węzła

Wieloznaczna segmentacja



- segmenty = DAG z rzutowaniem do ciągu pod-segmentów
- tylko jedna ścieżka (np. dwa okonie) podlega pełnej anotacji

Plan

1. Co chcemy zrobić

2. Model danych
   - Typy danych
   - Modelowanie NKJP
   - Założenia

3. Przegląd języka

Typy danych

- Typ prosty
  - integer, float
  - Boolean
  - string, enum (np. wartość kategorii gramatycznej)
- Struktura atrybutowa
  - np. węzeł; krawędź; interpretacja; struktura LFG; czcionka
  - przykładowa specyfikacja: `[orth = bym && !space]`
- Lista
  - np. segmentów w zdaniu; dzieci danego węzła; `{ gen, acc }`
- Lista wieloznaczności: elementy wybrane oraz niewybrane
  - np. interpretacje; transkrypcje; *zestawy* dzieci węzła
  - przykładowe użycie: `[msd ~ [pos = subst]]`
  - nie jest listą, raczej parą list (wybrane, niewybrane)

Rodzaje struktur

- wyróżnione typy struktur: węzły i krawędzie

węzły	**zwykłe struktury** (np. LFG)
• tworzą DAG • połączone krawędziami • $\rightsquigarrow$ fragmenty wejścia	• tworzą dowolny graf • połączone atrybutami • $\not\rightsquigarrow$ fragmenty wejścia

Rodzaje struktur

- wyróżnione typy struktur: węzły i krawędzie

węzły	**zwykłe struktury** (np. LFG)
• tworzą DAG • połączone krawędziami • $\rightsquigarrow$ fragmenty wejścia	• tworzą dowolny graf • połączone atrybutami • $\not\rightsquigarrow$ fragmenty wejścia

- dwa rodzaje krawędzi (jak np. w ANNIS)

krawędzie pierwotne	**krawędzie wtórne**
zastosowanie: składnia	zastosowanie: koreferencje
• tworzą DAG • lokalne • gęste	• tworzą dowolny graf • nieograniczone • rzadkie (?)

Atrybut typu

Specjalny atrybut dla dowolnych struktur

- konfigurowalna nazwa — u nas `type`
- wartości: `node`, `edge`, `interp`, `fstr`, ...

Atrybut typu

Specjalny atrybut dla dowolnych struktur

- konfigurowalna nazwa — u nas `type`
- wartości: `node`, `edge`, `interp`, `fstr`, ...
- zagnieżdżanie
  - ciąg podtypów: `node:syngr:NG:PrepNG`
  - dowolna grupa składniowa: `[type = ".*:syngr:.*"]`
  - lukier składniowy: `[type = syngr/t]`, `[syngr]`

Typy danych w praktyce – przykład NKJP

Uwagi na początek:

- to tylko przykład reprezentacji NKJP za pomocą dostępnych typów,
- nie należy się zbytnio przywiązywać do nazw (np. atrybutów),
- decyzje o sposobie reprezentacji należą do osoby przygotowującej korpus dla Poliqarpa,
- w zależności od potrzeb ten sam korpus może być różnie zamodelowany!

Typy danych w praktyce – przykład NKJP

Uwagi na początek:

- to tylko przykład reprezentacji NKJP za pomocą dostępnych typów,
- nie należy się zbytnio przywiązywać do nazw (np. atrybutów),
- decyzje o sposobie reprezentacji należą do osoby przygotowującej korpus dla Poliqarpa,
- w zależności od potrzeb ten sam korpus może być różnie zamodelowany!

Segmentacja

- Segmenty reprezentujemy przez węzły z type = `node:seg`.
- Reprezentacja DAGu segmentacji w NKJP uboga, ale można ją zamodelować:
  - węzły z type = `node:sub` – najdłuższe niepodzielne ciągi znaków (podsegmenty),
  - atrybuty text, newword, newline, endword itp,
  - krawędzie wtórne* z segmentów do podsegmentów.
- W segmentach również atrybuty text, newword, endword (na podstawie podsegmentów).

Segmentacja

- Segmenty reprezentujemy przez węzły z type = `node:seg`.
- Reprezentacja DAGu segmentacji w NKJP uboga, ale można ją zamodelować:
  - węzły z type = `node:sub` – najdłuższe niepodzielne ciągi znaków (podsegmenty),
  - atrybuty text, newword, newline, endword itp,
  - krawędzie wtórne* z segmentów do podsegmentów.
- W segmentach również atrybuty text, newword, endword (na podstawie podsegmentów).

Segmentacja

- Segmenty reprezentujemy przez węzły z type = `node:seg`.
- Reprezentacja DAGu segmentacji w NKJP uboga, ale można ją zamodelować:
  - węzły z type = `node:sub` – najdłuższe niepodzielne ciągi znaków (podsegmenty),
  - atrybuty text, newword, newline, endword itp,
  - krawędzie wtórne* z segmentów do podsegmentów.
- W segmentach również atrybuty text, newword, endword (na podstawie podsegmentów).

Segmentacja

- Segmenty reprezentujemy przez węzły z type = `node:seg`.
- Reprezentacja DAGu segmentacji w NKJP uboga, ale można ją zamodelować:
  - węzły z type = `node:sub` – najdłuższe niepodzielne ciągi znaków (podsegmenty),
  - atrybuty text, newword, newline, endword itp,
  - krawędzie wtórne* z segmentów do podsegmentów.
- W segmentach również atrybuty text, newword, endword (na podstawie podsegmentów).

Morfoskładnia

- Dodatkowe atrybuty węzłów `node:seg`:
- orth - jak w NKJP,
- msd - lista wieloznaczności, w niej struktury atrybutowe:
  - type = `morph`,
  - base,
  - tag zakodowany pozycyjnie (pos, case, ...)
- na liście msd dokładnie jedna wartość wybrana.

Morfoskładnia

- Dodatkowe atrybuty węzłów `node:seg`:
- orth - jak w NKJP,
- msd - lista wieloznaczności, w niej struktury atrybutowe:
  - type = `morph`,
  - base,
  - tag zakodowany pozycyjnie (pos, case, ...)
- na liście msd dokładnie jedna wartość wybrana.

Morfoskładnia

- Dodatkowe atrybuty węzłów `node:seg`:
- orth - jak w NKJP,
- msd - lista wieloznaczności, w niej struktury atrybutowe:
  - type = `morph`,
  - base,
  - tag zakodowany pozycyjnie (pos, case, ...)
- na liście msd dokładnie jedna wartość wybrana.

Morfoskładnia

- Dodatkowe atrybuty węzłów `node:seg`:
- orth - jak w NKJP,
- msd - lista wieloznaczności, w niej struktury atrybutowe:
  - type = `morph`,
  - base,
  - tag zakodowany pozycyjnie (pos, case, ...)
- na liście msd dokładnie jedna wartość wybrana.

Słowa składniowe

Węzły o typie `node:synw`.

- Atrybuty: orth, newword itp. - syntezowane ze składowych, msd - jak w segmentach (1-elementowa lista wieloznaczności),
- krawędzie pierwotne do składników (segmentów lub słów skł.).

Słowa składniowe

Węzły o typie `node:synw`.

- Atrybuty: orth, newword itp. - syntezowane ze składowych, msd - jak w segmentach (1-elementowa lista wieloznaczności),
- krawędzie pierwotne do składników (segmentów lub słów skł.).

Słowa składniowe

Węzły o typie `node:synw`.

- Atrybuty: orth, newword itp. - syntezowane ze składowych, msd - jak w segmentach (1-elementowa lista wieloznaczności),
- krawędzie pierwotne do składników (segmentów lub słów skł.).

Grupy składniowe

Również węzły...

- type = `node:syngr:<typ_grupy>` ,
- syntezowany orth,
- krawędzie pierwotne do składników (słów skł. lub grup):
  - type dla krawędzi: `head:synh` , `head:semh` , `nonhead` ;
  - `head:synh` i `head:semh` mogą wskazywać na ten sam węzeł.

Grupy składniowe

Również węzły…

- type = `node:syngr:<typ_grupy>`,
- syntezowany orth,
- krawędzie pierwotne do składników (słów skł. lub grup):
  - type dla krawędzi: `head:synh`, `head:semh`, `nonhead`;
  - `head:synh` i `head:semh` mogą wskazywać na ten sam węzeł.

Sensy słów

- Dodatkowy atrybut węzłów `node:seg` – sense,
- troche podobny do `msd`:
  - lista wieloznaczności zawierająca potencjalne tagi sensów (np. `{zostac.1...zostac.5}`),
  - jeden element wybrany,
  - lista pusta dla słów nieanotowanych.

Sensy słów

- Dodatkowy atrybut węzłów `node:seg` – sense,
- troche podobny do `msd`:
  - lista wieloznaczności zawierająca potencjalne tagi sensów (np. `{zostac.1...zostac.5}`),
  - jeden element wybrany,
  - lista pusta dla słów nieanotowanych.

Sensy słów

- Dodatkowy atrybut węzłów `node:seg` – sense,
- troche podobny do `msd`:
    - lista wieloznaczności zawierająca potencjalne tagi sensów (np. `{zostac.1...zostac.5}`),
    - jeden element wybrany,
    - lista pusta dla słów nieanotowanych.

Jednostki nazewnicze

- Struktura składniowa, węzły typu `node:named:typ_jednostki:podtyp_jednostki`,
- pozostałe atrybuty jednostek jak w korpusie (base, certainty...),
- struktura odrębna od słów i grup skł., zbudowana nad segmentami.

Jednostki nazewnicze

- Struktura składniowa, węzły typu `node:named:typ_jednostki:podtyp_jednostki`,
- pozostałe atrybuty jednostek jak w korpusie (base, certainty. . .),
- struktura odrębna od słów i grup skł., zbudowana nad segmentami.

Jednostki nazewnicze

- Struktura składniowa, węzły typu `node:named:typ_jednostki:podtyp_jednostki`,
- pozostałe atrybuty jednostek jak w korpusie (base, certainty...),
- struktura odrębna od słów i grup skł., zbudowana nad segmentami.

Segmentacja wielkoskalowa

- Zdania: węzły typu `node:s`, krawędzie do grup i/lub słów skł. oraz, niezależnie, do jednostek nazewniczych,
- akapity: węzły z type = `node:p`,
- dokumenty: najwyższy poziom grafu składni; węzły typu `node:doc`,
- dawne metadane są zwykłymi atrybutami dokumentów.

Segmentacja wielkoskalowa

- Zdania: węzły typu `node:s`, krawędzie do grup i/lub słów skł. oraz, niezależnie, do jednostek nazewniczych,
- akapity: węzły z type = `node:p`,
- dokumenty: najwyższy poziom grafu składni; węzły typu `node:doc`,
- dawne metadane są zwykłymi atrybutami dokumentów.

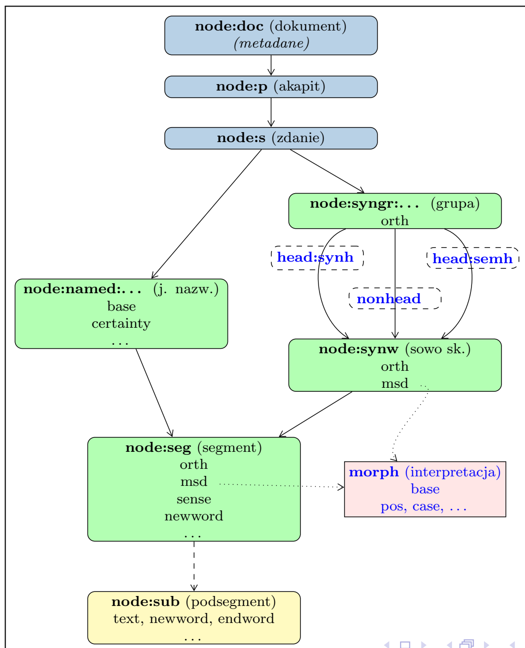
Segmentacja wielkoskalowa

- Zdania: węzły typu `node:s`, krawędzie do grup i/lub słów skł. oraz, niezależnie, do jednostek nazewniczych,
- akapity: węzły z type = `node:p`,
- dokumenty: najwyższy poziom grafu składni; węzły typu `node:doc`,
- dawne metadane są zwykłymi atrybutami dokumentów.

Segmentacja wielkoskalowa

- Zdania: węzły typu `node:s`, krawędzie do grup i/lub słów skł. oraz, niezależnie, do jednostek nazewniczych,
- akapity: węzły z type = `node:p`,
- dokumenty: najwyższy poziom grafu składni; węzły typu `node:doc`,
- dawne metadane są zwykłymi atrybutami dokumentów.

Podsumowanie obrazkowe



Ogólne założenia modelu danych

- Wyróżniamy segmenty = liście w grafie składniowym (patrzymy na krawędzie pierwotne),
- 1 wyróżniony rodzaj segmentacji wielkoskalowej (np. zdania),
- rozróżniamy porządek krawędziowy (dzieci węzła) i segmentowy (wg segmentów, niekoniecznie liniowy),
- struktury atrybutowe mogą być współdzielone (np. f-struktury w LFG).

Ogólne założenia modelu danych

- Wyróżniamy segmenty = liście w grafie składniowym (patrzymy na krawędzie pierwotne),
- 1 wyróżniony rodzaj segmentacji wielkoskalowej (np. zdania),
- rozróżniamy porządek krawędziowy (dzieci węzła) i segmentowy (wg segmentów, niekoniecznie liniowy),
- struktury atrybutowe mogą być współdzielone (np. f-struktury w LFG).

Ogólne założenia modelu danych

- Wyróżniamy segmenty = liście w grafie składniowym (patrzymy na krawędzie pierwotne),
- 1 wyróżniony rodzaj segmentacji wielkoskalowej (np. zdania),
- rozróżniamy porządek krawędziowy (dzieci węzła) i segmentowy (wg segmentów, niekoniecznie liniowy),
- struktury atrybutowe mogą być współdzielone (np. f-struktury w LFG).

Plan

1. Co chcemy zrobić
2. Model danych
3. Przegląd języka
   - Budowa zapytania
   - Obiekty, relacje, atrybuty
   - Zmienne
   - Inne

Budowa zapytania (część I)

Od szczegółu do ogółu

- Zapytanie egzystencjalne o wartość
  - węzeł `[orth = pies]`
  - lista węzłów `[pos = subst]*`

sukces $\Longleftrightarrow$ znaleziono specyfikowany(-e) obiekt(-y)

Budowa zapytania (część I)

Od szczegółu do ogółu

- Zapytanie egzystencjalne o wartość
  - węzeł `[orth = pies]`
  - lista węzłów `[pos = subst]*`

sukces $\Longleftrightarrow$ znaleziono specyfikowany(-e) obiekt(-y)

- Zapytanie relacyjne `[type = syngr:NG] > [orth = pies]`

sukces $\Longleftrightarrow$ sukces w podzapytaniach + zachodzi relacja

Budowa zapytania (część I)

Od szczegółu do ogółu

- Zapytanie egzystencjalne o wartość
  - węzeł `[orth = pies]`
  - lista węzłów `[pos = subst]*`

  sukces $\Longleftrightarrow$ znaleziono specyfikowany(-e) obiekt(-y)
- Zapytanie relacyjne `[type = syngr:NG] > [orth = pies]`

  sukces $\Longleftrightarrow$ sukces w podzapytaniach + zachodzi relacja
- Kombinacja logiczna zapytań

  ```
  $X:=[pos = subst] && (nie [ ] $X => $X.case = gen)
  ```

  sukces $\Longleftrightarrow$ wyrażenie obliczone na wynikach podzapytań daje `true`

Budowa zapytania (część I)

Od szczegółu do ogółu

- Zapytanie egzystencjalne o wartość
  - węzeł `[orth = pies]`
  - lista węzłów `[pos = subst]*`

  sukces ⟺ znaleziono specyfikowany(-e) obiekt(-y)
- Zapytanie relacyjne `[type = syngr:NG] > [orth = pies]`

  sukces ⟺ sukces w podzapytaniach + zachodzi relacja
- Kombinacja logiczna zapytań

  `$X:=[pos = subst] && (nie [ ] $X => $X.case = gen)`

  sukces ⟺ wyrażenie obliczone na wynikach podzapytań daje `true`

Morał: zapytanie = wyrażenie konwertowalne do booleana

(konwertowalne typy: węzeł; lista węzłów; boolean)

Opis obiektów

- Konstruktor (⇝ wartość) lub specyfikacja

typ	konstruktor	specyfikacja

liczba
boolean
string, enum
	`1` `2.5`
	`true` `false`
	`pies` `"\?"`
		—
		—
		`"p(ie)?s(a|u|ie)?"`
		(wyrażenie regularne)

struktura	—	`[orth = kot && !space]` (wyrażenie logiczne)
lista l. wielozn.	`{ nom, gen }` —	`{| [orth = kot]* |}` (wyrażenie regularne) —

- Aplikacja atrybutu do danego obiektu (⇝ wartość)

```
[pos = subst].msd.all.size
```

Relacje, operacje (podstawowe)

- Równość

`==`	ten sam obiekt	niekoniecznie wygodne
`=`	takie same obiekty	w konkretnej sytuacji ⇝ lukier
`!=`	nierówność	
`/=`	bez niejawnych konwersji (obiekt ⇝ lista 1-elem.; lista węzłów ⇝ boolean)	

Relacje, operacje (podstawowe)

- Równość

`==`	ten sam obiekt	niekoniecznie wygodne w konkretnej sytuacji ⇝ lukier
`=`	takie same obiekty	
`!=`	nierówność	
`/=`	bez niejawnych konwersji (obiekt ⇝ lista 1-elem.; lista węzłów ⇝ boolean)	

- Typy proste

`<` `<=`	porównywanie liczb (brak arytmetyki!)
`&&` `||` `!` `=>`	operacje logiczne
`@`	konkatenacja napisów

Relacje, operacje (na listach)

- Listy (działające też jako zbiory)

`[n]`	n-ty element	`contains`	zawieranie elementu
`[-n]`	n-ty od końca	`<=` `_<=_`	podciąg (?), podzbiór
`@`	konkatenacja	`_|_` `_&_` `__`	operacje na zbiorach
`.uniq`	$\rightsquigarrow$ zbiór		
`.size`	rozmiar		

Relacje, operacje (na listach)

- Listy (działające też jako zbiory)

`[n]`	n-ty element	`contains`	zawieranie elementu
`[-n]`	n-ty od końca	`<=` `_<=_`	podciąg (?), podzbiór
`@`	konkatenacja	`_|_` `_&_` `__`	operacje na zbiorach
`.uniq`	⇝ zbiór		
`.size`	rozmiar		

- Listy wieloznaczności

`.all`	lista wszystkich wartości	`~` `~~`	po staremu
`.sel`	lista wybranych wartości	`=` `==`	
`.val`	jedyna wybrana wartość, albo `null`		

Relacje, operacje (na listach)

- Listy (działające też jako zbiory)

`[n]`	n-ty element	`contains`	zawieranie elementu
`[-n]`	n-ty od końca	`<=` `_<=_`	podciąg (?), podzbiór
`@`	konkatenacja	`_|_` `_&_` `__`	operacje na zbiorach
`.uniq`	⇝ zbiór		
`.size`	rozmiar		

- Listy wieloznaczności

`.all`	lista wszystkich wartości	`~` `~~`	po staremu
`.sel`	lista wybranych wartości	`=` `==`	
`.val`	jedyna wybrana wartość, albo `null`		

przykład: `[msd.sel = {| [case=gen]* |}]` ⇝ `[msd==[case=gen]]` ⇝ `[msd.case==gen]` ⇝ `[case==gen]`

nowość: `[msd.sel = {| [case=gen && numb=sg]* |}]` ⇝ `[{case,numb}=={gen,sg}]`

Relacje (międzywęzłowe, część I)

- Dominacja

`>`	krawędź pierwotna
`->`	krawędź wtórna
`>[`*cond*`]`	krawędź spełnia *cond*
`>*`	ciąg krawędzi
`>{3,5}`	od 3 do 5 krawędzi
`>*{:`*spec*`:}`	węzły po drodze pasują do *spec*
`>[-1]`	krawędź do prawego syna

Relacje (międzywęzłowe, część I)

- Dominacja

`>`	krawędź pierwotna
`->`	krawędź wtórna
`>[`*cond*`]`	krawędź spełnia *cond*
`>*`	ciąg krawędzi
`>{3,5}`	od 3 do 5 krawędzi
`>*{:`*spec*`:}`	węzły po drodze pasują do *spec*
`>[-1]`	krawędź do prawego syna
X : *cond*	X taki, że *cond*

```
[type=node:s] >{,8}{: $X : $X !>* [orth=nie] :} [pos=fin]
```

ŹLE →
```
[type=node:s] >{,8}{: $X && $X !>* [orth=nie] :} [pos=fin]
```

Relacje (międzywęzłowe, część II)

- Precedencja
  - wg porządku krawędziowego (na dzieciach danego węzła)

`.sons`	lista dzieci węzła
`{| regex |}`	specyfikacja zawartości listy

```
   [type=NG].sons  = {| []* [pos=adj]  [pos=subst] []* |}
⇝  [type=NG].sons >= {      [pos=adj], [pos=subst]      }
```

Relacje (międzywęzłowe, część II)

- Precedencja
  - wg porządku krawędziowego (na dzieciach danego węzła)

`.sons`	lista dzieci węzła
`{| regex |}`	specyfikacja zawartości listy

```
  [type=NG].sons  = {| []* [pos=adj]  [pos=subst] []* |}
⇝ [type=NG].sons >= {     [pos=adj],  [pos=subst]      }
```

  - wg porządku segmentowego

`.span`	lista segmentów pod węzłem
regex **(bez kontekstu)**	specyfikacja ciągu segmentów

```
[type=NG]        > (      [pos=adj]  [pos=subst]      )
```

Relacje (międzywęzłowe, część II)

- Precedencja
  - wg porządku krawędziowego (na dzieciach danego węzła)

`.sons`	lista dzieci węzła
`{| ` *regex* ` |}`	specyfikacja zawartości listy

```
[type=NG].sons  = {| []* [pos=adj]  [pos=subst] []* |}
[type=NG].sons >= {      [pos=adj], [pos=subst]       }
```

  - wg porządku segmentowego

`.span`	lista segmentów pod węzłem
regex **(bez kontekstu)**	specyfikacja ciągu segmentów

```
[type=NG]       > (      [pos=adj]  [pos=subst]       )
```

Podsumowanie: trzy warianty wyrażeń regularnych

`{| tak+ |}`   `tak+`   `"tak+"`

Zmienne

A priori — dowolnego typu (w tym lista)

`$X`	bieżąca wartość
`$X:=`*wyr*	przypisz wyrażenie do `$X`
`$X+=`*wyr*	dopisz wyrażenie do `$X`

Przykłady:

nieciągłość (/\\)	`$A []* $B []* $C && $X > $A && $X !> $B && $X > $C`
dopełniacz negacji	`$X:=nie? [] [case=$C] && !$X.empty => $C=gen && ... ])`
częste „y" w NG	`[type=NG].span = {| ($Y+=y | [])* |} && $Y.size >= 2`

Kwantyfikatory (pierwszego rzędu)

- Składnia

`#X:` *wyr*	istnieje `$X` takie, że *wyr*
`##X:` *wyr*	dla każdego `$X` zachodzi *wyr*
`#3X:` *wyr*	istnieją dokładnie 3 wartości `$X` t. że *wyr*
`#3,10X:` *wyr*	istnieje pomiędzy 3 a 10 wartości `$X` t. że *wyr*

- Przykład — najniższy wspólny przodek:

```
$X >* {$Y, $Z} && ##W: ($W >* {$Y, $Z} => $W >* $X)
```

Kwantyfikatory (pierwszego rzędu)

- Składnia

`#X:` *wyr*	istnieje `$X` takie, że *wyr*
`##X:` *wyr*	dla każdego `$X` zachodzi *wyr*
`#3X:` *wyr*	istnieją dokładnie 3 wartości `$X` t. że *wyr*
`#3,10X:` *wyr*	istnieje pomiędzy 3 a 10 wartości `$X` t. że *wyr*

- Przykład — najniższy wspólny przodek:

```
$X >* {$Y, $Z} && ##W: ($W >* {$Y, $Z} => $W >* $X)
```

- Zasięg:
  - domyślnie — chunk (np. zdanie)
  - zmieniany przez `within`

  Przykład: `#5,X: $X.base=abażur within [type=p]`

Różności semantyczne

- specyfikacja / niezainicjowana zmienna $\rightsquigarrow$ kwantyfikator $\exists$
  w szczególności, obowiązuje zasięg kwantyfikatorów

przykład:
```
$X ->[type=coref] [] within [type=p] && ...
```

Różności semantyczne

- specyfikacja / niezainicjowana zmienna $\rightsquigarrow$ kwantyfikator $\exists$
  w szczególności, obowiązuje zasięg kwantyfikatorów

  przykład:
  ```
  $X -> [type=coref] [] within [type=p] && ...
  ```
- dopasowanie = wartościowanie, dla którego wychodzi `true`

Różności semantyczne

- specyfikacja / niezainicjowana zmienna $\rightsquigarrow$ kwantyfikator $\exists$
  w szczególności, obowiązuje zasięg kwantyfikatorów

  przykład: `$X ->[type=coref] [] within [type=p] && ...`

- dopasowanie = wartościowanie, dla którego wychodzi `true`
  a co z modyfikacjami zmiennych? dublujemy

  przykład: `$X:=la* && !$X.empty && $X:="a*" && $X.pos=qub`

ale modyfikacje to nie lukier: `($Z+=[orth!=y] | y)+`

Różności semantyczne

- specyfikacja / niezainicjowana zmienna ⇝ kwantyfikator ∃
  w szczególności, obowiązuje zasięg kwantyfikatorów

przykład: `$X -> [type=coref] [] within [type=p] && ...`

- dopasowanie = wartościowanie, dla którego wychodzi `true`
  a co z modyfikacjami zmiennych? dublujemy

przykład: `$X:=la* && !$X.empty && $X:="a*" && $X.pos=qub`

ale modyfikacje to nie lukier: `( $Z+=[orth!=y] | y)+`

- negacja
  niby proste, ale np. co ma znaczyć `!=` ?

string != *string-spec*	*string* nie spełnia *string-spec*
struct != *struct-spec*	*struct* nie równa się pewnemu *struct-spec*
string-spec$_1$!= *string-spec*$_2$	bez sensu
struct-spec$_1$!= *struct-spec*$_2$	istnieją i są różne

Flagi

Składnia: *wyr*/*flaga*

- dotyczące stringów

`/I` `/i`	uwzględnij/ignoruj wielkość liter
`/a(`*loc*`)`	użyj ustawień lokale *loc*
`/X` `/x` `/xl` `/xr`	toleruj podciągi (z żadnej/lewej/prawej/obu stron)

- dotyczące zmiennych

`/E` `/e`	zmienne mają różne/dowolne wartości (Tiger: `/e` , PML-TQ: `/E`)

- dotyczące segmentacji

`/S` `/s`	zabroń/dozwól rozsegmentowanie zap. prostych (bez rozspacjowania: np. `byłby` $\overset{?}{\leadsto}$ [był][by])
`/H` `/h`	zabroń/dozwól przeniesienie słowa do nowej linii

Post-processing

- łańcuchy zapytań

zap_1 ;; zap_2	wykonaj zap_1, a na każdym jego wyniku zap_2 (zap_2 może korzystać ze zmiennych z zap_1)

Przykłady:

przykrój wynik	`[type=VG] > $X:=[type=NG]3 ;; $X[2]`
AG NG AG na raty	`$X:=[type=NG] $Y:=[type=AG] ;; [type=AG] $X && $Y`

Post-processing

- łańcuchy zapytań

  zap_1 ;; zap_2 — wykonaj zap_1, a na każdym jego wyniku zap_2 (zap_2 może korzystać ze zmiennych z zap_1)

  Przykłady:

  przykrój wynik `[type=VG] > $X:=[type=NG]3 ;; $X[2]`

  AG NG AG na raty `$X:=[type=NG] $Y:=[type=AG] ;; [type=AG] $X && $Y`

- funkcje statystyczne (do konsultacji)

  zap ;; $niby\text{-}sql$ — wykonaj zap i na wszystkich jego wynikach $niby\text{-}sql$

  - siła wyrazu prostego SQL, ale składnia raczej inna (graficznie?)
  - wyr. logiczne SQL $\rightsquigarrow$ zapytania

  Wstępny przykład:

  z czym często jem? `jem [case=$C] ;; $C grpby $C ordby count() desc`

Dziękujemy!