|  |
| --- |
|  |

Biblioteka do analizy leksykalnej

|  |
| --- |
|  |

Spis treści

[2 Działanie leksera 2](#_Toc93779145)

[2.1 Wyrażenia regularne 3](#_Toc93779146)

[2.2 Automaty skończone 5](#_Toc93779147)

[2.2.1 NAS 6](#_Toc93779148)

[2.2.2 DAS 6](#_Toc93779149)

[3 Interfejs użytkownika 6](#_Toc93779150)

[3.1 Dodatki 6](#_Toc93779151)

[3.1.1 Logger 7](#_Toc93779152)

[3.1.2 UUID I IdCreator 8](#_Toc93779153)

[3.2 tworzenie leksera 8](#_Toc93779154)

[3.2.1 konstruktory 8](#_Toc93779155)

[3.2.2 czytanie reguł 8](#_Toc93779156)

[3.3 sprawdznie plikow 8](#_Toc93779157)

[4 przyklady uzycia 8](#_Toc93779158)

[4.1 przykład na kalkulator 8](#_Toc93779159)

[4.2 przykład na jezyk 8](#_Toc93779160)

[5 Budowanie drzewa rozkładu 9](#_Toc93779161)

[5.1 Funkcje nullable i folowpos 9](#_Toc93779162)

[5.2 Tworzenie wezla dla danej akcji 9](#_Toc93779163)

[5.3 Tworzenie drzewa dla przykladu 9](#_Toc93779164)

[6 Budowanie automatu 9](#_Toc93779165)

[6.1 Tworzenie das 9](#_Toc93779166)

[6.2 Mergowanie dasow 9](#_Toc93779167)

[6.3 Tworzenie dla przykladu 9](#_Toc93779168)

[7 Bibliografia 10](#_Toc93779169)

# Działanie leksera

Zadaniem leksera jest sprawdzenie, czy podany teks należy do zdefiniowanego języka i zwrócenie rozpoznanych słów w formie tokenów. Do zdefiniowania czym dla leksera jest język, musimy zdefiniować elementy budujące język.

**„Alfabetem** nazywamy dowolny skończony zbiór symboli”. (1) Symbolem może być dowolny element, który będziemy wykorzystywać do budowy naszego języka. Mogą być to cyfry litery znaki interpunkcyjne czy nawet emotikony. Jednym z przykładów alfabetu jest *alfabet binarny* będący zbiorem {0,1}.

„**Ciąg** […] nad alfabetem jest skończoną sekwencją symboli wybranych z tego alfabetu.” (1) *Słowo*, *zdanie* czy *ciąg* w teorii języków są synonimami. Do naszego języka może należeć puste słowo nieposiadające żadnego symbolu, oznaczamy je znakiem ϵ, a jego wielkość równa się zero.

Teraz możemy zdefiniować język jako „dowolny przeliczalny zbiór ciągów nad pewnym ustalonym alfabetem”. (1)

## Wyrażenia regularne

Do analizy plików lekser potrzebuje reguł opisujących nasz język. Jedną z notacji definiujących słowa w naszym języku są wyrażenia regularne (**regex**), które opisują zasady tworzenia słów w danym języku. Podstawy wyrażeń regularnych stanowią dwie reguły:

1. „ϵ jest wyrażeniem regularnym, a L(ϵ) to {ϵ}, czyli język, którego jedynym elementem jest pusty ciąg”. (1)
2. „Jeżeli a jest symbolem alfabetu ∑, wówczas a jest wyrażeniem regularnym, a L(a) = {a}, czyli jest to język z jednym ciągiem o długości jeden”. (1)

Tworząc ciągi, wykonujemy operacje na symbolach lub językach, które mają odzwierciedlenie w wyrażeniach regularnych. Mamy trzy główne operacje *sumy*, *złożenia* i *domknięcia*, które pokrótce opisze, na przykładzie języków L i M, gdzie L = {001, 10, 111} a M = {ϵ, 001}.

„**Złożenie** lub **konkatenacja** języków L i M – to zbiór łańcuchów, które można utworzyć przez wzięcie dowolnego łańcucha z L i złożenie go z dowolnym innym łańcuchem z M” (2) złożenie zapisujemy poprzez zapis składanych elementów obok siebie. Elementem neutralnym złożenia jest ciąg pusty, „inaczej mówiąc, dla dowolnego ciągu s, ϵs = sϵ = s”. (1) Dla naszego przykładu złożenie LM = {001, 10, 111, 001001, 10001, 111001}. Wyrażenie regularne ab, gdzie a ∈ L i b ∈ M, opisuje język L(a)L(b)

„**Suma teoriomnogościowa** dwóch języków L i M oznaczona przez L ∪ M to zbiór łańcuchów, które należą do L, do M lub do obu z nich”. (2) A więc L ∪ M = {ϵ, 10, 001, 111}. Sumę w wyrażeniach regularnych oznaczamy poprzez symbol „|”. Wyrażenie regularne a | b, gdzie a ∈ L i b ∈ M, opisuje język L(a) ∪ L(b). Wykorzystując tą operacji, możemy przedstawić język R składający się z słów {b, c} poprzez wyrażenie: b | c.

„**Domknięcie** (bądź **gwiazdka** lub **domkniecie Kleenego**) języka L oznaczone jest przez L\* i reprezentuje zbiór tych łańcuchów, które można utworzyć przez wzięcie dowolnej liczby łańcuchów z L, być może z powtórzeniami (tzn. ten sam łańcuch można wybrać więcej niż raz) i złożenie ich wszystkich”. (2) „Jeżeli pomyślimy o konkatenacji jako „iloczynie”, możemy zdefiniować „potęgowanie” ciągów następująco. Definiujemy s0 jako równe ϵ, a dla dowolnego i > 0 definiujemy si jako si-1s”. (1) Domkniecie jest takim „potęgowaniem” dla każdego i ≥ 0 i tworzy nieskończony zbiór dla dowolnego niepustego języka. Dla naszego przykładu M\* = {ϵ, 001,001001,001001001,..} zatem wyrażenie a\* dla a ∈ L opisuje język L(a)\*. Formalnie domknięcie kleenego możemy zapisać jako suma nieskończona.

L\* =

Aby ułatwić definiowanie wyrażeń regularnych mamy dodatkowe operatory powstałe z trzech podstawowych operacji:

**Dodatnie domknięcie Kleenego** języka L oznaczane jest przez L+ i reprezentuje zbiór łańcuchów, które można utworzyć przez złożenie dodatniej liczby łańcuchów z L. Operacja ta jest równoważna z operacją LL\*, a wyrażenie a+ dla a ∈ L oznacza L(a)L(a)\*.

**Jedno lub zero wystąpień** oznaczamy przez operator „?”. Jest on równoważny z wyrażeniem a | ϵ i dla a ∈ L oznacza L(a) | L(ϵ).

**Minimalna lub maksymalna liczba wystąpień** podawana jest w formie {n, m} gdzie n i m ∈ ℕ i n oznacza minimalną a m maksymalną liczbę wystąpień. Możemy pominąć jedną z cyfr i zdeklarować tylko minimalną liczbę wystąpień stosując operator {n,} lub {,m} deklarując maksymalną liczbę wystąpień.

Wyrażenie regularne a1 | a2 | … | an, gdzie każdy ai jest kolejnym symbolem z ASCI, możemy zastąpić zapisem pierwszego i ostatniego symbolu, oddzielonych znakiem „-’’, w nawiasach kwadratowych.

Wyrażenia regularne są analizowane przez bibliotekę od lewej strony i chcąc nadać priorytet pewnym operacją, należy umieścić je w nawiasie.

## Automaty skończone

„***Automat skończony*** *jest modelem matematycznym systemu o dyskretnych wejściach i wyjściach. System taki w danej chwili może znajdować się w jednym ze skończonej liczby stanów, który to stan jest ściśle uzależniony od stanu poprzedniego. Jeden ze stanów pełni rolę stanu początkowego, od którego dany automat rozpoczyna działanie, z drugiej strony niektóre stany pełnią rolę stanów końcowych kończąc pracę automatu. Praca automatu oparta jest na analizie symboli wejściowych ze skończonego alfabetu.[…] Po przeanalizowaniu wszystkich symboli automat skończony może przyjąć jeden z dwu stanów: akceptacji lub nieakceptacji*”.(3) Celem biblioteki jest zbudowanie takiego automatu na podstawie dostarczonych wyrażeń regularnych i implementacja algorytmu, symulującego przechodzenie przez różne stany w zależności od analizowanego tekstu, który zwróci informacje czy analizowany ciąg spełnia zdeklarowane reguły.

Często automaty są przedstawiane w formie grafów skierowanych, gdzie wierzchołki obrazują stany automatu, a krawędzie możliwe przejście między stanami. Wierzchołki symbolizujące stan akceptujący oznaczamy podwójnym kółkiem. Formalnym zapisem Automatu jest uporządkowana piątka < Q, ∑, δ, q0, F> gdzie:

Q – skończony zbiór stanów

∑ – skończony alfabet symboli wejściowych

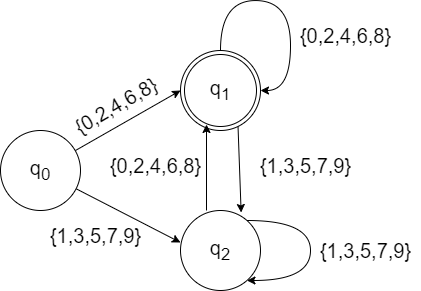
δ – funkcje przejść odwzorowujące Q x ∑ w Q

q0 – stan początkowy należący do Q

F – zbiór stanów końcowych należących do Q

**Przykład 2.1: Automat skończony akceptujący liczby podzielne przez 2.**

Wejściem automatu będą cyfry od 0 do 9, czyli ∑ = {0,1,2,3,4,5,6,7,8,9}, Q = {q0, q1, q2}. q0 jest stanem początkowym, a q1 jest stanem akceptującym. Funkcje przejść prowadzą dla liczb parzystych z q0,q1, q1 do q2 a dla liczb nieparzystych prowadzą do q2. Podany automat możemy zapisać w postaci grafowej.



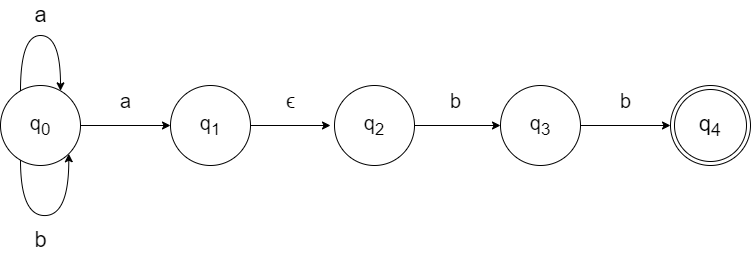
Rysunek . Automat akceptujący liczby podzielne przez 2

„*Automaty skończone występują w dwóch odmianach:*

1. ***Niedeterministyczne automaty skończone*** *(NAS) nie mają ograniczeń co do etykiet występujących ba krawędziach grafu. Pewien symbol może występować na kilku krawędziach grafu. Pewien symbol może występować na kilku krawędziach wychodzących z tego samego stanu, a ponadto dopuszczalną etykietą jest ϵ, pusty ciąg.*
2. ***Deterministyczne automaty skończone*** *(DAS) mają dla każdego stanu i każdego symbolu alfabetu wejściowego dokładnie jedną krawędź wychodzącą z tego stanu opatrzoną tym symbolem*”. (1)

Funkcje przejść w niedeterministycznych automatach skończonych zwracają zbiór stanów należących do Q

**Przykład 2.2: Graf przejść dla NAS akceptującego język (a|b)\*abb**

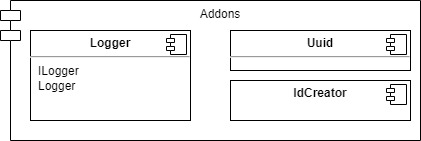


Rysunek . NAS dla języka (a|b)\*abb

# Interfejs użytkownika

## Dodatki

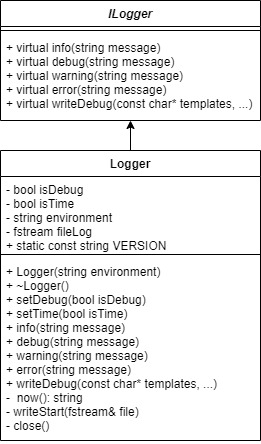
Biblioteka korzysta z kilku dodatkowych funkcjonalności nie będących typowym działaniem leksera. Jednak pozwoliły one na usprawnienie algorytmów oraz ułatwiły debugowanie jej. Klasy te wydzielone są do osobnej przestrzeni nazw i są umieszczone w folderze addons.



Rysunek . Elementy w module Addons.

### Logger

Do łatwiejszego śledzenia i debuggowania w bibliotece logowane są wykonywane akcje oraz ich rezultaty. Mamy możliwość przekazania do konstruktora *leksera* własny obiekt spełniający interfejs deklarowany przez abstrakcyjną klasę *ILekser* w przeciwnym przypadku wykorzystana domyślna implementacja.



Rysunek . Schemat Loggera

Domyślna implementacja *loggera* zapisuje informacje do pliku log.txt. Tworzy, go w folderze Lekser. Ilość logowanych informacji możemy zmieniać, modyfikując flagę *isDebug*. Możemy do logowanej wiadomości dodać czas, w którym zdarzenie miało miejsce, przez sterowanie flaga *isTime.* Funkcja logująca *error*, oprócz zapisywania w pliku wiadomości oznaczonej słowem [ERROR], wypisuje podaną wiadomość na standardowe wyjście, nadając jej czerwony kolor.

### UUID I IdCreator

Dodatek *UUID* pozwala na generowanie identyfikatora. Klasa generuje stu dwudziesto ośmio bitową unikalną etykietę. Identyfikator jest ciągiem 36 znaków gdzie ósmy, trzynasty, osiemnasty i dwudziesty trzeci znak są myślnikami. Twórcą wykorzystanego algorytmu jest Fernando Moreno Valles (4). Metoda ta nie jest najlepszym sposobem otrzymywania UUID jednak w nie dużych ilościach ma bardzo duże prawdopodobieństwo unikalności a jego niewątpliwą zaletą jest szybkość i prostota. Również bardzo istotną zaletą jest brak dodatkowych zależności. Wygenerowane identyfikatory służą przede wszystkim do identyfikacji poszczególnych kroków automatu. Za brak duplikowania się kroków odpowiada klasa *IdCreator,* którana podstawie otrzymanych informacji potrafi zwrócić identyfikator kroku dla którego został wygenerowany już identyfikator, w przeciwnym razie generuje nowy UUID.

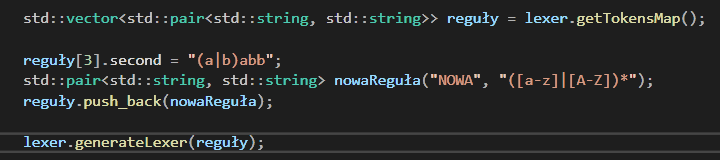
## tworzenie leksera

Aby korzystać z Leksera należy utworzyć instancje obiektu klasy Lekser. Tworząc obiekt możemy opcjonalnie podać trzy argumenty do konstruktora. Pierwszym argumentem jest źródło reguł w postaci ścieżki do pliku z zdefiniowanymi regułami lub vector par <string, string> gdzie pierwszym elementem jest etykieta tokenu a drugim elementem wyrażenie regularne definiujące regułę. Jako następny argument możemy podać implementacje klasy abstrakcyjne ILoger. Ostatnim z argumentów jest mapa wywołań zwrotnych( callbacks ), które zostaną wywołane podczas odczytania tokenów. Kluczem mapy jest etykieta tokenu dla którego definiujemy funkcje, a wartością funkcja która zostanie wywołana po odczytaniu zdefiniowanego tokenu. Funkcja musi spełniać interfejs std::function< void( std::string ) >, gdzie argumentem funkcji jest ciąg odczytany podczas identyfikacji tokenu.

### 

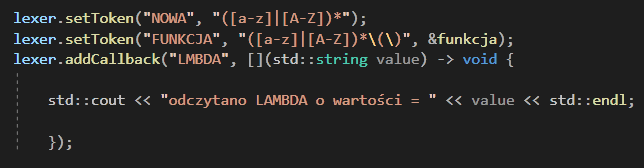
Rysunek . Przykład konstruowania leksera

Obiekt leksera możemy modyfikować w czasie podmieniając reguły, dodając nowe tokeny lub wywołania zwrotne. Mamy możliwość zmiany źródła reguł podając nową nazwę pliku lub nowy vector z regułami. Biblioteka daje możliwość pobrania aktualnych reguł i ich modyfikacji.



Rysunek . Przykład modyfikacji reguł

Biblioteka pozwala dodawać pojedyncze zasady oraz wywołania zwrotne bezpośrednio na obiekcie naszego leksera. Wywołania zwrotne możemy definiować w wygodnej postaci wyrażeń lamba.



Rysunek . Pojedyncze modyfikowanie leksera

Ostatnią a zarazem najważniejszą funkcją biblioteki jest analiza leksykalna. Analizie podany może być ciąg znaków w formie stringu lub plik którego nazwę wraz z ścieżką podamy jako argument do funkcji analizeFile. Podczas analizy wywoływane będą zdeklarowane wywołania zwrotne ,a sama funkcja zwróci vektor obiektów Token. Obiekt ten zawiera dwie propercje:

1. name mówiący o nazwie odczytanego tokenu.
2. value zawiera informacje o ciągu który odpowiadał za odczytanie tokenu.

Tokeny zwrócone są w kolejności odczytanej podczas analizy.

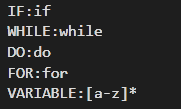
## Odczyt reguł

Za odczyt reguł odpowiedzialny jest obiekt klasy LekserRuleReader. Odpowiada on za Dostarczanie reguł utworzonych z dostarczonego vektora lub pliku.

Jednym z dwóch dostępnych źródeł reguł jest plik tekstowy. Przygotowując plik z regułami musimy stosować się do kilku zasad:

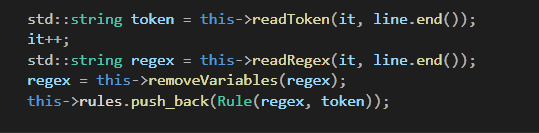
1. Każdą z zasad piszemy w osobnej linii.
2. Każda zasada dzieli się na 2 części które dzieli znak „:”.
3. Pierwszą część stanowi nazwa tokenu.
4. Druga część to wyrażenie regularne definiujące podamy token.

Dla ułatwienia definiowania reguł mamy możliwość wykorzystania tokenu wcześniej zdefiniowanego w pliku jako zmiennej umieszczanej w innym wyrażeniu regularnym. Jeśli analizując ciąg będzie spełniał on reguły więcej niż jednego tokenu zostanie odczytany jako ten token ,który jest później zdefiniowany w pliku lub vektorze reguł.



Rysunek . przykład definicji zasad w pliku

plik z regułami jest analizowany linia po lini na początku odczytujemy etykiete tokenu a następnie regułę w postaci wyrażenia regularnego. Ostatnim krokiem jest sprawdzenie czy w podanym regexie występują zmienne. Rozpoznajemy je poprzez ciąg, będący nazwą wcześniej zdefiniowanego tokenu, otoczony znakiem „$” na początku i końcu ciągu. Szukamy wyrażenia regularnego definiującego podany token i umieszczamy go zamiast zmiennej. Dla nadania priorytetu regule kryjącej się za zmienną umieszczona zostaje ona w nawiasach. Podczas odczytania zmiennej której nie odczytała jeszcze biblioteka rzucany jest wyjątek. Tak odczytana reguła jest dodawana do reguł trzymanych przez *LekserRuleReader.* Kończąc generowanie reguł moduł zwraca wszystkie trzymane reguły.



Rysunek . Odczytywanie reguły z pliku

## analiza plików

\*\*\*\*\*\*\*\*\*\*\*10\*\*\*\*\*\*\*\*\*\*\*

# przyklady uzycia

## przykład na kalkulator

## przykład na jezyk

\*\*\*\*\*\*\*\*\*\*\*14\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Budowanie drzewa rozkładu

## Funkcje nullable i folowpos

## Tworzenie wezla dla danej akcji

## Tworzenie drzewa dla przykladu

\*\*\*\*\*\*\*\*\*\*\*\*\*20\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Budowanie automatu

## Tworzenie das

## Mergowanie dasow

## Tworzenie dla przykladu

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*30\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Bibliografia

1. **Alfred V. Aho Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman.** *Kompilatory: reguły, metody i narzędzia .* Warszawa : Wydawnictwo Naukowe PWN SA, 2019.

2. **J.E. Hopcroft R. Motwani, J.D. Ullman.** *Wprowadzenie do teorii automatów, języków i obliczeń.* Warszawa : Wydawnictwo naukowe PWN, 2005. 83-01-14502-1.

3. **Boryczka Urszula.** prac.us.edu.pl. [Online] [Zacytowano: 20 styczeń 2022.] http://prac.us.edu.pl/~uboryczk/wdi2/pliki/pdf/wyklad5pdf.pdf.

4. **Valles Fernando Moreno.** github. [Online] [Zacytowano: 20 12 2020.] https://gist.github.com/fernandomv3/46a6d7656f50ee8d39dc#file-uuid-hpp.

5. **Ben-Ari Mordechai.** *Understanding Programming Languages.* Chichester : John Wiley & Sons, 1996.