|  |
| --- |
|  |

Biblioteka do analizy leksykalnej

|  |
| --- |
|  |

Spis treści

[2 Działanie leksera 2](#_Toc94136644)

[2.1 Wyrażenia regularne 2](#_Toc94136645)

[2.2 Automaty skończone 4](#_Toc94136646)

[3 Interfejs użytkownika 6](#_Toc94136647)

[3.1 Dodatki 6](#_Toc94136648)

[3.1.1 Logger 6](#_Toc94136649)

[3.1.2 UUID I IdCreator 7](#_Toc94136650)

[3.2 tworzenie leksera 8](#_Toc94136651)

[3.3 Odczyt reguł 9](#_Toc94136652)

[3.4 analiza ciągów 10](#_Toc94136653)

[4 przykłady użycia 11](#_Toc94136654)

[4.1 Zadanie 1 11](#_Toc94136655)

[4.2 przykład na jezyk 11](#_Toc94136656)

[5 Budowanie drzewa składniowego 12](#_Toc94136657)

[5.1 Algorytm tworzenia drzewa składniowego 13](#_Toc94136658)

[5.2 Funkcje przeszukujące drzewo składniowe 13](#_Toc94136659)

[5.3 Tworzenie drzewa dla przykladu 13](#_Toc94136660)

[6 Budowanie automatu 13](#_Toc94136661)

[6.1 Tworzenie das 13](#_Toc94136662)

[6.2 Mergowanie dasow 13](#_Toc94136663)

[6.3 Tworzenie dla przykladu 13](#_Toc94136664)

[7 Bibliografia 14](#_Toc94136665)

# Działanie leksera

Zadaniem leksera jest sprawdzenie, czy podany teks należy do zdefiniowanego języka i zwrócenie rozpoznanych słów w formie żetonów (**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.

Dla ułatwienia wyrażenia regularne są analizowane przez bibliotekę od lewej strony, wszystkie operacje mają ten sam priorytet 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

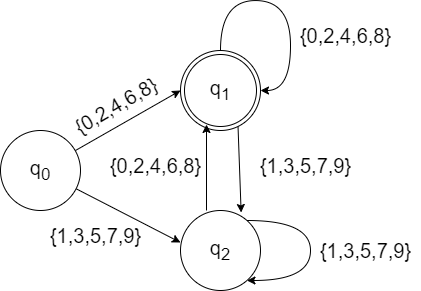
δ – funkcje przejść odwzorowujące Q x ∑ w Q

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.



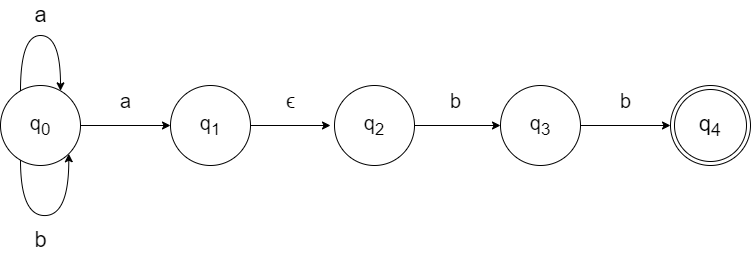
Rysunek . Automat akceptujący liczby podzielne przez 2

„*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**

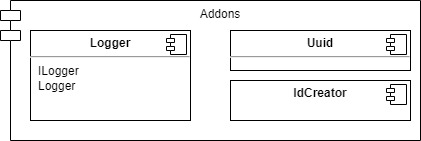


Rysunek . NAS dla języka (a|b)\*abb

# Interfejs użytkownika

## Dodatki

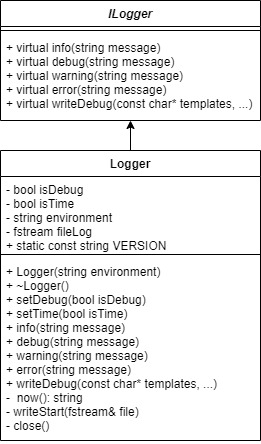
Biblioteka korzysta z kilku dodatkowych funkcjonalności nie będących typowym działaniem leksera. Jednak pozwoliły one na usprawnienie algorytmów oraz ułatwiły debugowanie jej. Klasy te wydzielone są do osobnej przestrzeni nazw i są umieszczone w folderze addons.



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

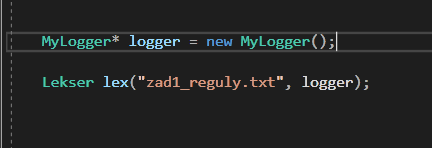
Domyślna implementacja *loggera* zapisuje informacje do pliku log.txt. Tworzy, go w folderze Lekser. Ilość logowanych informacji możemy zmieniać, modyfikując flagę *isDebug*. Możemy do logowanej wiadomości dodać czas, w którym zdarzenie miało miejsce, przez sterowanie flaga *isTime.* Funkcja logująca *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.

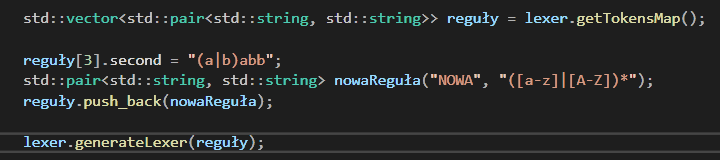
## tworzenie leksera

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 ze 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 żetonów. 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.



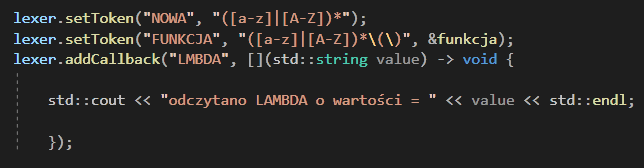
Rysunek . Przykład konstruowania leksera

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

Rozszerzenie 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 ze 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.

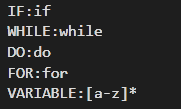
## Odczyt reguł

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:

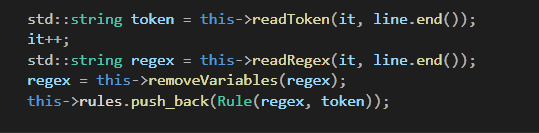
1. Każdą z zasad piszemy w osobnej linii.
2. Każda zasada dzieli się na 2 części, które dzieli znak „ : ”.
3. Pierwszą część stanowi nazwa tokenu.
4. Druga część to wyrażenie regularne definiujące podamy token.

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

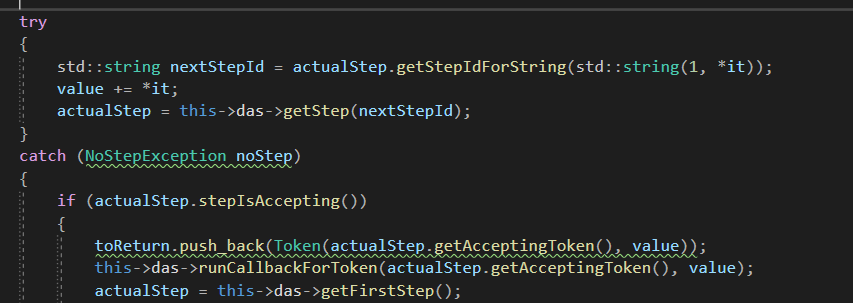
Plik z regułami jest analizowany linia po linii, na początku odczytujemy etykietę tokenu, a następnie regułę w postaci wyrażenia regularnego. Ostatnim krokiem jest sprawdzenie, czy w podanym regexie występują zmienne. Rozpoznajemy je poprzez ciąg, będący nazwą wcześniej zdefiniowanego tokenu, otoczony znakiem „$” na początku i końcu ciągu. Szukamy wyrażenia regularnego kryjącego się pod zmienną i umieszczamy go zamiast niej. 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, moduł zwraca wszystkie trzymane reguły.



Rysunek . Odczytywanie reguły z pliku

## analiza ciągów

Za analizę odpowiada klasa *LekserAnalizer*, której metody przyjmują ciąg znaków lub nazwę pliku. Tworząc instancje klasy, przekazujemy wskaźnik do obiektu odpowiadającego za symulacje automatu. Dostarczone dane analizowane są linia po linii. Analizator odczytane znaki dostarcza do automatu, który przechodzi przez kolejne kroki. Jeżeli dla odczytanego znaku automat nie posiada przejścia, biblioteka sprawdza czy automat znajduje się w stanie akceptującym. Dla stanu akceptującego zwracamy odczytany token zwierający informacje o etykiecie żetonu i odczytanym ciągu. Dla rozpoznanego tokenu wywoływana jest funkcja, o ile została zdefiniowana. Następnie resetujemy automat do stanu początkowego i zaczynamy analizę od znaku, dla którego nie odnaleziono przejścia. Zatrzymanie analizy w stanie nieakceptującym jest błędem, komunikowanym przez bibliotekę. Błąd ten kończy analizę.



Rysunek . Analiza ciągu

Jeżeli pomiędzy odczytywanymi tokenami występowały białe znaki, są one ignorowane.

# przykłady użycia

Działanie biblioteki przedstawię na podstawie dwóch zadań. Wykorzystamy je podczas analizy poszczególnych kroków tworzenia automatów skończonych.

## Zadanie 1

W pierwszym zadaniu musimy stworzyć analizator leksykalny rozpoznający poprawnie zapisane wyryżenia arytmetyczne. Przygotowany program powinien rozpoznawać liczby całkowite w zapisie dziesiętnym i szesnastkowym (dodatnie i ujemne), liczby rzeczywiste w zapisie tradycyjnym i wykładniczym, nawiasy zamykające i otwierające oraz podstawowe operatory: +, -, \*, /. Po zakończeniu działania program powinien wydrukować na standardowym wyjściu podsumowanie zawierające liczbę użytych operatorów, liczbę i łączną sumę znalezionych liczb całkowitych oraz liczbę i łączną sumę znalezionych liczb rzeczywistych.

Zaczniemy od zdefiniowania wyrażeń regularnych w pliku tekstowym. Przyjrzymy się regułą odpowiedzialnym za rozpoznawanie liczb dziesiętnych. Pierwsza reguła *digit:[0-9]* odpowiada za rozpoznawanie pojedynczej cyfry. Aby rozpoznać liczbę tworzony regułę która wymaga przynajmniej 1 cyfry *digits:$digit$+*. W tej regule wykorzystaliśmy zmienną *digit*. Token *sign\_opt* odpowiada za znak ujemnej liczby. Teraz możemy zbudować docelową regułę rozpoznającą liczby dziesiętne tworząc wyrażenie *dec\_num:$sign\_opt$?$digits$*.Stworzone wyrażenie jest równoważne z zapisem:

-?[0-9]+, jednak dzięki wykorzystaniu zmiennych zwiększamy czytelność wyrażenia.

Wszystkie reguły:

Digit : [0-9]

hex\_digit: $digit$ | [a-f] | [A-F]

hex\_digits: $hex\_digit$+

digits: $digit$+

hex\_pref: 0 (X|x)

fractional: .$digits$

sign\_opt: \-

exp\_opt: (e|E) $sign\_opt$? $digits$

plus\_op: \+

minus\_op: \-

mult\_op: \\*

div\_op: \/

operator: $plus\_op$|$minus\_op$|$mult\_op$|$div\_op$

op\_bracket: \(

cl\_bracket: \)

dec\_num: $sign\_opt$?$digits$

hex\_num: $hex\_pref$$hex\_digits$

double\_num: $sign\_opt$?$digits$$fractional$

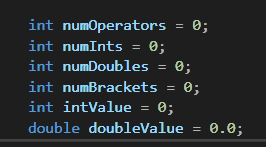
double\_exp\_num: $sign\_opt$?$digits$$fractional$?$exp\_opt$

z tak przygotowanym plikiem możemy utworzyć obiekt leksera podając nazwę pliku.



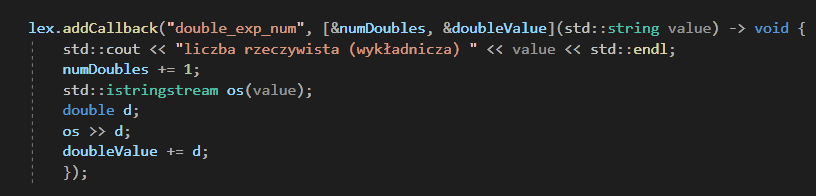
Rysunek . Tworzenie leksera

Następnie inicjalizujemy zmienne służące do zliczenia wyników.



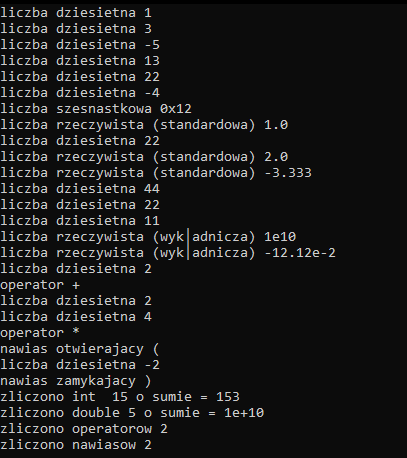
Rysunek . Inicjalizacja zmiennych

Ostatnim elementem przygotowania analizatora jest zdefiniowanie wywołań zwrotnych, które będą aktualizować zmienne podczas rozpoznania liczby. Skorzystamy z funkcji *addCallback* gdzie jako argumenty przekażemy etykietę żetonu oraz wyrażenie labda.



Rysunek . Dodanie wywołania zwrotnego dla liczby rzeczywistej

Na koniec wywołujemy analizę pliku i wypisujemy na standardowe wyjście wyniki analizy.



Rysunek . Wynik przykładowej analizy

Zadanie to posiada sporą ilość zasad jednak nie są one skomplikowane. W zadaniu wykorzystano funkcje zwrotne posiadające logikę. Zadanie pozwoliło na sprawdzenie biblioteki w działaniu i pozwoliło mi wykryć kilka błędów w generowaniu automatu.

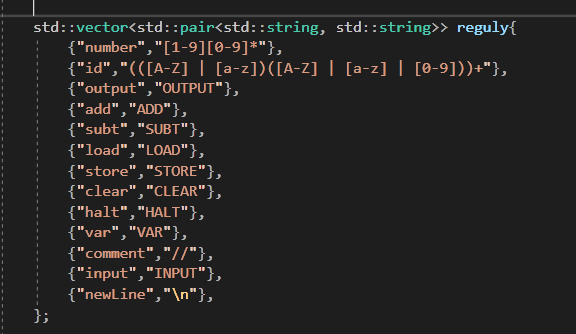
## Zadanie 2

W zadaniu 2 zbudujemy analizator leksykalny rozpoznający składnie prostego pseudoassemblera. Lekser będzie rozpoznawał następujące słowa kluczowe:

* INPUT – wczytanie zawartości rejestru ze standardowego wejścia
* OUTPUT – wydrukowanie zawartości rejestru na standardowym wyjściu
* LOAD x – wczytanie do rejestru wartości z pamięci pod adresem x
* STORE x – zapisanie wartości z rejestru w pamięci pod adresem x
* ADD x – dodanie do zawartości rejestru zawartości pamięci pod adresem x
* SUBT x – odjęcie od zawartości rejestru zawartości pamięci pod adresem x
* VAR x y – umieszczenie wartości y w pamięci pod adresem x
* CLEAR – wyzerowanie rejestru
* HALT – zatrzymanie działania programu.

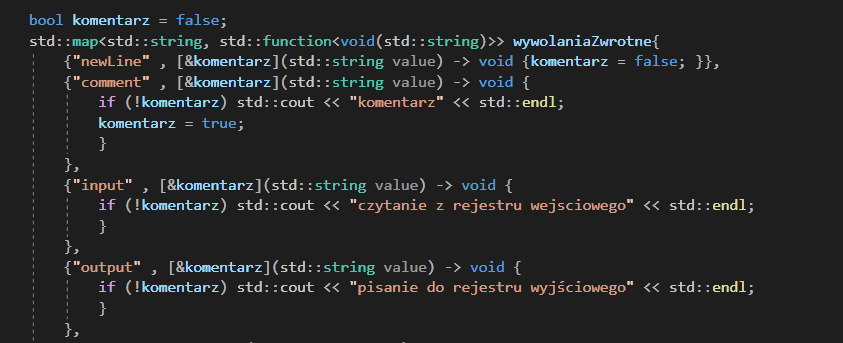
Adresy podawane będą za pomocą liczb całkowitych lub w postaci zmiennych. Nazwa zmiennych może być dowolnym ciągiem alfanumerycznym rozpoczynający się literą.

W tym zadaniu wyrażenia regularne dostarczymy za pomocą vectora.



Rysunek . Reguły leksera w zadaniu 2

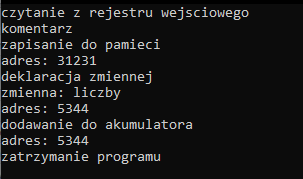
Mapę wywołania zwrotnych również utworzymy przed konstruowaniem Leksera. Wywołania będą wypisywać na standardowe wyjście odczytaną instrukcje o ile nie jest ona w komentarzu.



Rysunek . Mapa wywołań zwrotnych

Z tak przygotowanymi zmiennymi możemy utworzyć nasz analizator oraz rozpocząć analizę pseudoassemblera.

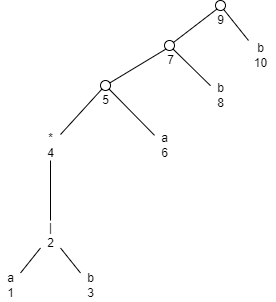
Zadanie to sprawdza lekser w roli części kompilatora rozpoznającego jakiś język. Wyrażenia regularne są proste i większość z nich rozpoznaje tylko jedno słowo w zadanym alfabecie. Obawiałem się czy biblioteka poradzi sobie z rozpoznawaniem nowej lini, jednak moje obawy okazały się niesłuszne i w bardzo prosty sposób udało się uzyskać tą funkcjonalność.



Rysunek . Przykładowy wynik działania programu

# Budowanie drzewa składniowego

Pierwszym etapem utworzenia automatu z wyrażenia regularnego jest zmiana postaci regexa z tekstu na drzewo rozkładu. Drzewo stanowi strukturę drzewa binarnego gdzie liście opowiadają operandom a węzły operatorom.



Rysunek . Drzewo składniowe dla (a|b)\*abb

Za odwzorowania drzewa przez bibliotece odpowiada specjalna struktura *regexNode*. Odpowiada ona za pojedynczy węzeł lub liść w drzewie rozkładu. Struktura posiada wskaźniki na swojego rodzica oraz lewe i prawe dziecko. Klasa posiada również pola mówiące o typie węzła lub jego etykiecie. Każdy z elementów drzewa jest numerowany tak że elementy w lewym poddrzewie mają numery mniejsze od rodzica a elementy prawego poddrzewa większe. Korzystając z tej właściwości możemy przeglądać drzewo w dół za pomocą operatora []. Ogranicza nas to w poruszaniu się po strukturze poprzez indeks, jeżeli nie posiadamy referencji do korzenia drzewa, Jednak nadal możemy poruszać się po drzewie w górę poprzez wskaźnik na rodzica.

## Algorytm tworzenia drzewa składniowego

**Wejście:** wyrażenie regularne oraz numer od którego zaczyna numerować elementy drzewa **Wyjście:** wskaźnik na korzeń będący obiektem klasy *regexNode.*

**Metoda:**

1. **Wybierz pierwszy element regexa.**
2. **Sprawdź typ odczytanego elementu i dobierz odpowiednia operacje.**
3. **Wykonaj operacje dodającą węzły do drzewa.**
4. **Jeżeli wybrany element nie jest ostatni wybierz następny element i przejdź do 2.**
5. **Zwróć wygenerowane drzewo.**

Implementacja algorytmu znajduje się w klasie *RegexService,* która dziedziczy po klasie *RegexConstructorSyntaxTree.* Jest to algorytm pierwszy algorytm implementowany podczas tworzenia biblioteki. Liczne zmiany oraz sytuacje które wyniknęły podczas dalszego tworzenia biblioteki spowodowały, duży spadek jakości kodu. Największym mankamentem implementacji jest sposób obsługi stringu odpowiedzialnego za wyrażenie arytmetyczne. Bardzo pomocna okazała się duża liczba testów dzięki której pomimo częstej zmiany kodu miałem pewność że pozostałe elementy implementacji nie ulegały degradacji.

Analizę zaczynamy od utworzenia pierwszego elementu na podstawie pierwszego znaku dostarczonego wyrażenia, gdzie następuje walidacja czy nie zaczynamy wyrażenia znakiem specjalnym. Kolejnym etapem jest pętla po wszystkich znakach wyrażenia. Pierwszym Krokiem w pętli jest sprawdzenie czy poprzednio dodanym węzłem nie jest operacja sumy. Jeżeli poprzednią operacją jest suma walidujemy czy aktualnie przetwarzany znak nie jest symbolem specjalnym i dodajemy go jako prawe dziecko operacji sumy. W każdym innym przypadku korzystamy z funkcji *checkAction*, która zwraca nam wskaźnik do funkcji. Metoda *checkAction* na podstawie dostarczonego symbolu wyrażenia dobiera odpowiednią funkcje która dodaje węzeł odpowiadający symbolowi. Wywołanie takiej funkcji zawsze powoduje zwrócenie nowego obiektu będącego korzeniem generowanego drzewa. Dla operacji w nawiasach wywołujemy rekurencyjnie budowę drzewa z wyrażenia w nawiasie i dołączamy je do naszego głównego drzewa.



Rysunek . Funkcja checkAction

Po wyjściu z pętli metoda dodaje do wygenerowanego drzewa węzeł oznaczający koniec wyrażenia z etykietą #. Węzeł ten wykorzystamy w następnych etapach tworzenia automatu.

## Przykład tworzenia drzewa rozkładu

Przeanalizujemy tworzenie drzewa rozkładu dla wyrażenia regularnego definiującego żeton *hex\_num* z zadania pierwszego opisanego w rozdziale 4. Wyrażenie po zastąpieniu zmiennych wygląda następująco: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Funkcje przeszukujące drzewo składniowe

Do konstruowania deterministycznego automatu skończonego bezpośrednio z drzewa rozkładu wykorzystamy trzy funkcje analizujące nasze drzewo:

1. *nullable*
2. *firstPos*
3. *followPos*

Funkcja nullable zwraca informacje czy wyrażenie reprezentowane przez drzewo które otrzyma funkcja może zwrócić słowo puste. Operacja *firstPos* zwraca vector z numerami węzłów otrzymanego drzewa, które odpowiadają pierwszemu symbolowi co najmniej jednemu słowu tworzonego z wyrażenia. Funkcja przechodzi rekurencyjnie przez drzewo w poszukiwaniu węzłów. Dla węzła odpowiadającego za sumę zwraca *firstPos* dla lewego i prawego poddrzewa. W wypadku operacji konkatenacji zwracamy *firstPos* dla lewego poddrzewa i sprawdzamy czy może one tworzyć pusty ciąg, jeżeli tak dodajemy do zwracanego zbioru *firstPos* prawego drzewa. Dla drzewa z rysunku 5.1 funkcja *firstPos* zwróci zbiór{1, 3, 6}. Najbardziej skomplikowana funkcja *followPos* zwraca zbiór pozycji mogących nastąpić po otrzymanym węźle. Metoda ta przeszukuje rekurencyjnie drzewo w górę korzystając z wskaźnika na rodzica. Operacja sprawdza typ operacji reprezentowany przez węzeł i buduje zbiór który zostanie zwrócony przez funkcje.

# Budowanie automatu

Budując automat dla leksera zaczynamy od zbudowania Deterministycznego automatu dla każdego żetonu osobno. Aby uzyskać jeden automat tworzymy NAS, w którym z pierwszego stanu mamy puste przejścia do stanów będących pierwszym stanem uprzednio wygenerowanych automatów deterministycznych. Z takiego niedeterministycznego automatu tworzymy automat deterministyczny, z którego będziemy korzystać nasz Lekser.

## Tworzenie das

## Mergowanie dasow

## Tworzenie dla przykladu

# 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.